# Table of Contents

What is AWS IoT Greengrass? ..................................................................................................................... 1  
   For first-time users .......................................................................................................................... 1  
   For existing users .......................................................................................................................... 1  
How AWS IoT Greengrass works .............................................................................................................. 2  
   Key concepts .................................................................................................................................... 2  
   Features of AWS IoT Greengrass .................................................................................................... 4  
Move from Version 1 ............................................................................................................................ 5  
   Differences between V1 and V2 ..................................................................................................... 5  
   Run V1 applications on V2 ............................................................................................................. 7  
What's new in Version 2 ....................................................................................................................... 10  
   AWS IoT Greengrass Core v2.4.0 software update ......................................................................... 11  
      Public component updates ........................................................................................................ 11  
   AWS IoT Greengrass Core v2.3.0 software update ......................................................................... 13  
      Public component updates ........................................................................................................ 13  
   AWS IoT Greengrass Core v2.2.0 software update ......................................................................... 14  
      Public component updates ........................................................................................................ 14  
   AWS IoT Greengrass Core v2.1.0 software update ......................................................................... 16  
      Platform support updates ........................................................................................................... 16  
      Public component updates ........................................................................................................ 17  
   AWS IoT Greengrass Core v2.0.5 software update ......................................................................... 21  
      Public component updates ........................................................................................................ 21  
   AWS IoT Greengrass Core v2.0.4 software update ......................................................................... 22  
      Public component updates ........................................................................................................ 22  
Getting started tutorial ....................................................................................................................... 24  
   Prerequisites .................................................................................................................................... 24  
   Set up your environment .................................................................................................................. 25  
   Install the AWS IoT Greengrass Core software ........................................................................... 26  
   Create your first component .......................................................................................................... 30  
   Upload your component ................................................................................................................. 35  
   Deploy your component .................................................................................................................. 39  
Setting up core devices ....................................................................................................................... 41  
   Supported platforms ....................................................................................................................... 41  
   Requirements .................................................................................................................................... 42  
      Requirements to run Lambda functions ....................................................................................... 43  
      Set up an AWS account ................................................................................................................. 44  
   Install the AWS IoT Greengrass Core software ........................................................................... 45  
      Install with automatic provisioning ........................................................................................... 46  
      Install with manual provisioning ............................................................................................... 50  
      Install with fleet provisioning ..................................................................................................... 62  
      Install with custom provisioning ................................................................................................. 83  
      Installer arguments .................................................................................................................... 89  
   Run the AWS IoT Greengrass Core software ............................................................................... 91  
      Check if the AWS IoT Greengrass Core software runs as a system service ............................ 92  
      Run the AWS IoT Greengrass Core software as a system service ........................................... 92  
      Run the AWS IoT Greengrass Core software without a system service .................................. 93  
Run AWS IoT Greengrass in Docker ................................................................................................. 95  
   Supported platforms and requirements ....................................................................................... 95  
   Software downloads ....................................................................................................................... 94  
   Choose how to provision AWS resources ...................................................................................... 95  
   Build the AWS IoT Greengrass image from a Dockerfile ............................................................. 95  
   Run AWS IoT Greengrass in Docker with automatic provisioning .............................................. 99  
   Run AWS IoT Greengrass in Docker with manual provisioning ................................................... 104  
   Troubleshooting AWS IoT Greengrass in a Docker container ..................................................... 117  
Configure the AWS IoT Greengrass Core software ...................................................................... 119  

AWS IoT Greengrass Developer Guide, Version 2
<table>
<thead>
<tr>
<th>Component</th>
<th>Type</th>
<th>Versions</th>
<th>Configuration</th>
<th>Dependencies</th>
<th>Input data</th>
<th>Output data</th>
<th>Local log file</th>
<th>Licenses</th>
<th>Changelog</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS IoT Greengrass Developer Guide, Version 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local log file</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Changelog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device Defender</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Versions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local log file</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Changelog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IP detector</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Versions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local log file</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Changelog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kinesis Data Firehose</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Versions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local log file</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Changelog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See also</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lambda launcher</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Versions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local log file</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Changelog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lambda manager</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Versions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local log file</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Changelog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lambda runtimes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Versions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local log file</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Changelog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Token exchange service</td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure tunneling</td>
<td>289</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secret manager</td>
<td>285</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS IoT Greengrass Developer Guide, Version 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Versions</td>
<td>283</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td>284</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td>284</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>284</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local log file</td>
<td>285</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Changelog</td>
<td>285</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Secure tunneling</strong></td>
<td>289</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Secret manager</strong></td>
<td>285</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Shadow manager</strong></td>
<td>294</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Amazon SNS</strong></td>
<td>298</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Stream manager</strong></td>
<td>306</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Token exchange service</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Versions</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Requirements</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Local log file</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Changelog</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Licenses</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Requirements</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Versions</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Local log file</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Changelog</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Licenses</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Requirements</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Versions</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Local log file</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Changelog</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Licenses</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Requirements</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Versions</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Local log file</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Changelog</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Licenses</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Requirements</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Versions</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Local log file</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Changelog</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Licenses</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Requirements</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Versions</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Local log file</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Changelog</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Licenses</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Requirements</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Versions</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Local log file</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Changelog</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Licenses</strong></td>
<td>311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Use interprocess communication (IPC) ............................................................................................... 396
Deploy components to devices ......................................................................................................... 383
Interact with local shadows ..................................................................................................... 462
Interact with component lifecycle ............................................................................................. 446
Publish/subscribe AWS IoT Core MQTT messages ....................................................................... 433
Publish/subscribe local messages ............................................................................................. 410
Subscribe to IPC event streams ................................................................................................ 405
Authorize components to perform IPC operations ....................................................................... 403
Supported SDKs ..................................................................................................................... 396
Check deployment status ........................................................................................................ 395
Create deployments ................................................................................................................ 384
Check device deployment status .......................................................................................... 395
Update component configurations ..................................................................................... 390
Revise deployments ............................................................................................................. 393
Cancel deployments ............................................................................................................. 394
Check deployment status ...................................................................................................... 395
Import a Lambda function (CLI) ............................................................................................... 368
Step 1: Define the Lambda function configuration .............................................................. 368
Step 2: Create the Lambda function component .................................................................... 381
Import a Lambda function as a component (console) ............................................................... 364
Step 1: Choose a Lambda function to import ........................................................................ 364
Step 2: Configure Lambda function parameters ............................................................... 365
Step 3: (Optional) Specify supported platforms for the Lambda function ......................... 366
Step 4: (Optional) Specify component dependencies for the Lambda function ............... 366
Step 5: (Optional) Run the Lambda function in a container .............................................. 367
Step 6: Create the Lambda function component .................................................................... 368
Deploy components to devices ............................................................................................... 383
Create deployments ................................................................................................................ 384
Update component configurations ..................................................................................... 390
Revise deployments ............................................................................................................. 393
Cancel deployments ............................................................................................................. 394
Check deployment status ...................................................................................................... 395
Import a Lambda function as a component (console) ............................................................... 364
GetThingShadow ............................................................................................................ 465
Authorization ................................................................................................................. 462
Examples ....................................................................................................................... 458
GetSecretValue ............................................................................................................... 456
Authorization ................................................................................................................. 455
SendConfigurationValidityReport ...................................................................................... 454
UpdateConfiguration ....................................................................................................... 452
SubscribeToComponentUpdates ....................................................................................... 448
ResumeComponent ........................................................................................................... 450
GetConfiguration ............................................................................................................. 451
UpdateConfiguration ....................................................................................................... 452
SubscribeToConfigurationUpdate .................................................................................... 453
SubscribeToValidateConfigurationUpdates ........................................................................ 454
SendConfigurationValidityReport .................................................................................... 454
Retrieve secret values ........................................................................................................ 455
Authorization ................................................................................................................. 455
Examples ....................................................................................................................... 458
Interact with local shadows ............................................................................................... 462
Minimum SDK versions ...................................................................................................... 462
Authorization ................................................................................................................. 462
GetThingShadow ............................................................................................................. 465
Perform machine learning inference ................................................................. 558
How AWS IoT Greengrass ML inference works .................................................. 558
What's different in Version 2? ........................................................................ 559
Requirements ................................................................................................. 559
Supported model sources ............................................................................... 559
Supported runtimes ....................................................................................... 560
Machine learning components ......................................................................... 560

Perform sample image classification inference ................................................. 564
Prerequisites ................................................................................................. 564
Step 1: Subscribe to the default notifications topic ........................................... 564
Step 2: Deploy the TensorFlow Lite image classification component ............. 565
Step 3: View inference results ....................................................................... 566
Next steps ....................................................................................................... 567

Perform sample image classification inference on images from a camera ...... 568
Prerequisites ................................................................................................. 568
Step 1: Configure the camera module on your device ....................................... 569
Step 2: Verify your subscription to the default notifications topic .................. 571
Step 3: Modify the TensorFlow Lite image classification component configuration and deploy it . 571
Step 4: View inference results ....................................................................... 572
Next steps ....................................................................................................... 573

Use SageMaker Edge Manager ........................................................................ 573
How it works ................................................................................................. 574
Requirements ............................................................................................... 574
Get started with SageMaker Edge Manager .................................................... 575

Customize your machine learning components .............................................. 581
Modify the configuration of a public inference component ............................. 582
Use a custom model with the sample inference component .......................... 583
Create custom machine learning components .............................................. 585
Create a custom inference component ......................................................... 587

Troubleshooting ......................................................................................... 592
Failed to fetch library .................................................................................. 592
Cannot open shared object file ..................................................................... 592
<library> not found ..................................................................................... 593
No CUDA-capable device is detected ............................................................ 593
No such file or directory ............................................................................. 593
Memory errors ............................................................................................ 594
Disk space errors ....................................................................................... 594
Timeout errors ........................................................................................... 594

Greengrass Command Line Interface ............................................................ 595
Install the Greengrass CLI ........................................................................... 595
Prerequisites ............................................................................................... 595
Deploy the Greengrass CLI component ......................................................... 595

Greengrass CLI commands ........................................................................... 597
component ................................................................................................. 598
deployment ................................................................................................. 600
logs .............................................................................................................. 602
get-debug-password .................................................................................... 606

Security ....................................................................................................... 607
Data protection ............................................................................................ 607
Data encryption .......................................................................................... 608
Device authentication and authorization ....................................................... 610
X.509 certificates ......................................................................................... 610
AWS IoT policies ....................................................................................... 611
Update a core device’s AWS IoT policy ....................................................... 613
Minimal AWS IoT policy ............................................................................ 614
Troubleshooting IDT for AWS IoT Greengrass V2

Where to look for errors

Support policy for AWS IoT Device Tester for AWS IoT Greengrass

Troubleshooting

View AWS IoT Greengrass Core software logs

View component logs

AWS IoT Greengrass Core software issues

Unable to set up core device

Unable to connect to AWS IoT Core

Out of memory error

Unable to install Greengrass CLI

User root is not allowed to execute

Failed to map segment from shared object: operation not permitted

AWS IoT Greengrass cloud issues

An error occurred (AccessDeniedException) when calling the CreateComponentVersion operation: User: arn:aws:iam::<username> is not authorized to perform:

null

Core device deployment issues

Error: com.aws.greengrass.componentmanager.exceptions.PackageDownloadException: Failed to download artifact

Error:


Error:

com.aws.greengrass.componentmanager.exceptions.NoAvailableComponentVersionException: Failed to negotiate component <name> version with cloud and no local applicable version satisfying requirement <requirements>

software.amazon.awssdk.services.secretsmanager.model.SecretsManagerException: User: <user> is not authorized to perform: secretsmanager:GetSecretValue on resource: <arn>

Info:

com.aws.greengrass.deployment.exceptions.RetryableDeploymentDocumentDownloadException: Greengrass Cloud Service returned an error when getting full deployment configuration

Info: com.aws.greengrass.deployment.DeploymentDocumentDownloader: Calling Greengrass cloud to get full deployment configuration

Core device component issues

Python script doesn't log messages

AWS CLI issues

Error: Invalid choice: 'greengrassv2'

Tag your resources

Using tags in AWS IoT Greengrass V2
What is AWS IoT Greengrass?

AWS IoT Greengrass is an open source Internet of Things (IoT) edge runtime and cloud service that helps you build, deploy and manage IoT applications on your devices. You can use AWS IoT Greengrass to build software that enables your devices to act locally on the data that they generate, run predictions based on machine learning models, and filter and aggregate device data. AWS IoT Greengrass enables your devices to collect and analyze data closer to where that data is generated, react autonomously to local events, and communicate securely with other devices on the local network. Greengrass devices can also communicate securely with AWS IoT Core and export IoT data to the AWS Cloud. You can use AWS IoT Greengrass to build edge applications using pre-built software modules, called components, that can connect your edge devices to AWS services or third-party services. You can also use AWS IoT Greengrass to package and run your software using Lambda functions, Docker containers, native operating system processes, or custom runtimes of your choice.

The following example shows how an AWS IoT Greengrass device interacts with the AWS Cloud.

For first-time users of AWS IoT Greengrass

If you're new to AWS IoT Greengrass, we recommend that you review the following section:

- How AWS IoT Greengrass works (p. 2)

Next, follow the getting started tutorial (p. 24) to try out the basic features of AWS IoT Greengrass. In this tutorial, you install the AWS IoT Greengrass Core software on a device, develop a Hello World component, and package that component for deployment.

For existing users of AWS IoT Greengrass

For current users of AWS IoT Greengrass, we recommend the following topics to help you understand the new features in AWS IoT Greengrass Version 2, and learn how to move from version 1 to version 2:

- What's new in AWS IoT Greengrass Version 2 (p. 10)
- Move from AWS IoT Greengrass Version 1 (p. 5)
How AWS IoT Greengrass works

The AWS IoT Greengrass client software, also called AWS IoT Greengrass Core software, runs on Linux-based distributions, such as Ubuntu or Raspberry Pi OS, for devices with ARM or x86 architectures. With AWS IoT Greengrass, you can program devices to act locally on the data they generate, run predictions based on machine learning models, and filter and aggregate device data. AWS IoT Greengrass enables local execution of AWS Lambda functions, Docker containers, native OS processes, or custom runtimes of your choice.

AWS IoT Greengrass provides pre-built software modules called components that let you easily extend edge device functionality. AWS IoT Greengrass components enable you to connect to AWS services and third-party applications at the edge. After you develop your IoT applications, AWS IoT Greengrass enables you to remotely deploy, configure, and manage those applications on your fleet of devices in the field.

The following example shows how an AWS IoT Greengrass device interacts with the AWS IoT Greengrass cloud service and other AWS services in the AWS Cloud.

Key concepts for AWS IoT Greengrass

The following are essential concepts for understanding and using AWS IoT Greengrass:

**Greengrass core device**

A device that runs the AWS IoT Greengrass Core software. A Greengrass core device is an AWS IoT thing. You can add multiple core devices to AWS IoT thing groups to create and manage groups of Greengrass core devices. For more information, see Setting up AWS IoT Greengrass core devices (p. 41).

**Greengrass client device**

A device that connects to and communicates with a Greengrass core device over MQTT. A Greengrass client device is an AWS IoT thing. The core device can process, filter, and aggregate data from client
devices that connect to it. You can configure the core device to relay MQTT messages between client devices, the AWS IoT Core cloud service, and Greengrass components. For more information, see Interact with local IoT devices (p. 477).

Client devices can run FreeRTOS or use the AWS IoT Device SDK or Greengrass discovery API (p. 502) to get information about core devices to which they can connect.

**Greengrass component**

A software module that is deployed to and runs on a Greengrass core device. All software that is developed and deployed with AWS IoT Greengrass is modeled as a component. AWS IoT Greengrass provides pre-built public components that provide features and functionality that you can use in your applications. You can also develop your own custom components, on your local device or in the cloud. After you develop a custom component, you can use the AWS IoT Greengrass cloud service to deploy it to single or multiple core devices. You can create a custom component and deploy that component to a core device. When you do, the core device downloads the following resources to run the component:

- **Recipe**: A JSON or YAML file that describes the software module by defining component details, configuration, and parameters.
- **Artifact**: The source code, binaries, or scripts that define the software that will run on your device. You can create artifacts from scratch, or you can create a component using a Lambda function, a Docker container, or a custom runtime.
- **Dependency**: The relationship between components that enables you to enforce automatic updates or restarts of dependent components. For example, you can have a secure message processing component dependent on an encryption component. This ensures that any updates to the encryption component automatically update and restart the message processing component.

For more information, see AWS-provided components (p. 132) and Develop AWS IoT Greengrass components (p. 322).

**Deployment**

The process to send components and apply the desired component configuration to a destination target device, which can be a single Greengrass core device or a group of Greengrass core devices. Deployments automatically apply any updated component configurations to the target and include any other components that are defined as dependencies. You can also clone an existing deployment to create a new deployment that uses the same components but is deployed to a different target. Deployments are continuous, which means that any updates you make to the components or the component configuration of a deployment automatically get sent to all destination targets. For more information, see Deploy AWS IoT Greengrass components to devices (p. 383).

**AWS IoT Greengrass Core software**

The set of all AWS IoT Greengrass software that you install on a core device. AWS IoT Greengrass Core software comprises the following:

- **Nucleus**: This required component provides the minimum functionality of the AWS IoT Greengrass Core software. The nucleus manages deployments, orchestration, and lifecycle management of other components. It also facilitates communication between AWS IoT Greengrass components locally on an individual device. For more information, see Greengrass nucleus (p. 136).
- **Optional components**: These configurable components are provided by AWS IoT Greengrass and enable additional features on your edge devices. Depending on your requirements, you can choose the optional components that you want to deploy to your device, such as data streaming, local machine learning inference, or a local command line interface. For more information, see AWS-provided components (p. 132).

You can upgrade your AWS IoT Greengrass Core software by deploying new versions of your components to your device.
Features of AWS IoT Greengrass

AWS IoT Greengrass Version 2 consists of the following elements:

- **Software distributions**
  - The Greengrass nucleus component, which is the minimum installation of the AWS IoT Greengrass Core software. This component manages deployments, orchestration, and lifecycle management of Greengrass components.
  - Additional optional components provided by AWS that integrate with services, protocols, and software.
  - The AWS IoT Device SDK, which contains the interprocess communication (IPC) SDK and the Greengrass discovery SDK for client devices.
  - The Stream Manager SDK.

- **Cloud service**
  - AWS IoT Greengrass V2 API
  - AWS IoT Greengrass V2 console

**AWS IoT Greengrass Core software**

You can use the AWS IoT Greengrass Core software that runs on your edge devices to do the following:

- Process data streams on the local device with automatic exports to the AWS Cloud. For more information, see Manage data streams on the AWS IoT Greengrass Core (p. 519).
- Support MQTT messaging between AWS IoT and components. For more information, see Publish/subscribe AWS IoT Core MQTT messages (p. 433).
- Interact with local devices that connect and communicate over MQTT. For more information, see Interact with local IoT devices (p. 477).
- Support local publish and subscribe messaging between components. For more information, see Publish/subscribe local messages (p. 410).
- Deploy and invoke components and Lambda functions. For more information, see Deploy AWS IoT Greengrass components to devices (p. 383).
- Manage component lifecycles, such as with support for install and run scripts. For more information, see AWS IoT Greengrass component recipe reference (p. 341).
- Perform secure, over-the-air (OTA) software updates of the AWS IoT Greengrass Core software and custom components. For more information, see Update the AWS IoT Greengrass Core software (OTA) (p. 129) and Deploy AWS IoT Greengrass components to devices (p. 383).
- Provide secure, encrypted storage of local secrets and controlled access by components. For more information, see Secret manager (p. 285).
- Secure connections between devices and the AWS Cloud with device authentication and authorization. For more information, see Device authentication and authorization for AWS IoT Greengrass (p. 610).

You configure and manage Greengrass core devices through AWS IoT Greengrass APIs where you create continuous software deployments. For more information, see Deploy AWS IoT Greengrass components to devices (p. 383).

For more information about supported platforms, requirements, and downloads, see Setting up AWS IoT Greengrass core devices (p. 41).

By downloading this software, you agree to the Greengrass Core Software License Agreement.
Move from AWS IoT Greengrass Version 1

AWS IoT Greengrass Version 2 is a new major version release of the AWS IoT Greengrass Core software, APIs, and console. You can’t use the AWS IoT Greengrass Core software v1.x with the V2 APIs. Likewise, you can’t use the AWS IoT Greengrass Core software v2.0 with the V1 APIs. However, by using some modifications, you can run your V1 applications on AWS IoT Greengrass V2.

Topics

• Differences between V1 and V2 (p. 5)
• Run AWS IoT Greengrass V1 applications on AWS IoT Greengrass V2 (p. 7)

Differences between V1 and V2

AWS IoT Greengrass V2 introduces new fundamental concepts for devices, fleets, and deployable software. This section describes the V1 concepts that are different in V2.

• AWS IoT Greengrass groups and deployments

In AWS IoT Greengrass V1, a group defines a core device, the settings and software for that core device, and the list of AWS IoT things that connect to that core device.

In AWS IoT Greengrass V2, you use deployments to define the software components and configurations that run on core devices. Each deployment targets a single core device or an AWS IoT thing group that can contain multiple core devices. Deployments to thing groups are continuous, so when you add a core device to a thing group, it receives the software configuration for that fleet. For more information, see Deploy AWS IoT Greengrass components to devices (p. 383).

You can also create local deployments to develop and test custom software components. For more information, see Create local AWS IoT Greengrass components (p. 324).

• AWS IoT Greengrass Core software and connectors

In AWS IoT Greengrass V1, the AWS IoT Greengrass Core software is a single package that contains the software and all of its features. AWS IoT Greengrass connectors are modules that you deploy to AWS IoT Greengrass V1 core devices.

In AWS IoT Greengrass V2, the AWS IoT Greengrass Core software is modular, so that you can choose what to install to control the memory footprint. The Greengrass nucleus component (p. 136) is the minimum required installation of the AWS IoT Greengrass Core software that handles deployments, orchestration, and lifecycle management of other components. Features such as stream manager, secret manager, and log manager are components that you deploy only when you need those features. AWS IoT Greengrass V2 also provides some AWS IoT Greengrass V1 connectors as components. For more information, see AWS-provided components (p. 132).

• AWS Lambda functions

In AWS IoT Greengrass V1, Lambda functions define the software that runs on core devices. In each Greengrass group, you define subscriptions and local resources that the function uses. You also define the container parameters for functions that the AWS IoT Greengrass Core software runs in a containerized Lambda runtime environment.

In AWS IoT Greengrass V2, components are the software that run on core devices. Components can consist of any software applications, and each component has a recipe that defines the component’s metadata, parameters, dependencies, and scripts to run at each step in the component lifecycle. The recipe also defines the component’s artifacts, which are binary files such as scripts, compiled code, and static resources. When you deploy a component to a core device, the core device downloads the...
component recipe and artifacts to run the component. For more information, see Develop AWS IoT Greengrass components (p. 322).

You can import Lambda functions as components that run in a Lambda runtime environment in AWS IoT Greengrass V2. When you import the Lambda function, you specify the subscriptions, local resources, and container parameters for the function. For more information, see Run AWS IoT Greengrass V1 applications on AWS IoT Greengrass V2 (p. 7).

- **Subscriptions**

In AWS IoT Greengrass V1, subscriptions specify where Lambda functions receive event messages to consume as function payloads. Functions subscribe to local publish/subscribe messages and AWS IoT Core MQTT messages.

In AWS IoT Greengrass V2, components manage their own subscriptions to local publish/subscribe and AWS IoT Core MQTT messages. In the component recipe, you define authorization policies to specify which topics the component can use to communicate. In component code, you can use interprocess communication (IPC) for local publish/subscribe messaging and AWS IoT Core MQTT messaging. For more information, see Use the AWS IoT Device SDK for interprocess communication (IPC) (p. 396).

- **Local resources**

In AWS IoT Greengrass V1, Lambda functions run in containers that you configure to access volumes and devices on the core device's file system.

In AWS IoT Greengrass V2, components run outside containers, so you don't need to specify which local resources the component can access. You can develop components that work directly with local resources on core devices. You can also develop components that run Docker containers. For more information, see Run a Docker container (p. 331).

  **Note**

When you import a containerized Lambda function as a component, you specify the local resources that the function uses.

- **Greengrass devices (connected devices)**

In AWS IoT Greengrass V1, Greengrass devices are AWS IoT things that you add to a Greengrass group to connect to the core device in that group and communicate over MQTT. You must deploy that group each time that you add or remove a connected device. You use subscriptions to relay messages between connected devices, AWS IoT Core, and applications on the core device.

In AWS IoT Greengrass V2, connected devices are called Greengrass client devices, and you associate client devices to core devices to connect them and communicate over MQTT. You can define authorization policies that apply to groups of client devices, so you don't need to create a deployment to add or remove a client device. To relay messages between client devices, AWS IoT Core, and Greengrass components, you can configure an optional MQTT bridge component. For more information, see Interact with local IoT devices (p. 477).

In both AWS IoT Greengrass V1 and AWS IoT Greengrass V2, devices can run FreeRTOS or use the AWS IoT Device SDK or Greengrass discovery API (p. 502) to get information about core devices to which they can connect.

- **Local shadow service**

In AWS IoT Greengrass V1, the local shadow service is enabled by default, and supports only unnamed classic shadows. You use the Greengrass Core SDK in your Lambda functions to interact with shadows on your devices.

In AWS IoT Greengrass V2, you enable the local shadow service by deploying the shadow manager component. You can then use the AWS IoT Device SDK V2 in Lambda functions, or in custom components, to interact with shadows on your devices.
In both AWS IoT Greengrass V1 and AWS IoT Greengrass V2, you can sync local shadow states with cloud shadows in AWS IoT Core. For more information, see Interact with device shadows (p. 515).

Run AWS IoT Greengrass V1 applications on AWS IoT Greengrass V2

You can run most AWS IoT Greengrass V1 applications on AWS IoT Greengrass V2. You can use AWS-provided components that offer the same functionality as AWS IoT Greengrass connectors, and you can import Lambda functions as components that run on AWS IoT Greengrass V2.

Topics
- Can I run my Greengrass v1.x applications on Greengrass v2.0? (p. 7)
- Run V1 Lambda functions (p. 7)
- Run AWS IoT Greengrass connectors (p. 8)
- Run machine learning inference (p. 9)
- Connect V1 Greengrass devices (p. 9)

Can I run my Greengrass v1.x applications on Greengrass v2.0?

AWS IoT Greengrass provides features that you can use to run your AWS IoT Greengrass Core software v1.x applications on the AWS IoT Greengrass Core software v2.0. However, if your v1.x applications use any of the following listed features, you won’t be able to run them on the v2.0 software yet.

- Hardware security integration
- Stream manager telemetry metrics
- The following languages in the AWS IoT Greengrass Core SDK:
  - Node.js
  - C
  - The C and C++ Lambda function runtimes.

You can develop custom components (p. 322) to build any feature or runtime to run on Greengrass core devices.

Run V1 Lambda functions

You can import Lambda functions as AWS IoT Greengrass V2 components. If your components use features such as stream manager or local secrets, you must define dependencies on the AWS-provided components that package these features' functionality. When you deploy a component, the deployment includes the component dependencies that you specify. You configure these dependent features when you deploy your Lambda function component.

If your Lambda function uses features such as stream manager or local secrets, you must define dependencies on the AWS-provided components that package these features. When you deploy the Lambda function component, the deployment also includes the component for each feature that you define as a dependency. In the deployment, you can configure parameters such as which secrets to deploy to the core device. Not all V1 features require a component dependency for your Lambda function on V2. The following list describes how to use V1 features on V2 in your Lambda function component.

- Stream manager
If your Lambda function uses stream manager, specify `aws.greengrass.StreamManager` as a component dependency when you import the function. When you deploy the stream manager component, specify the stream manager parameters to set for the target core devices. For more information, see Stream manager (p. 306).

- **Local secrets**

  If your Lambda function uses local secrets, specify `aws.greengrass.SecretManager` as a component dependency when you import the function. When you deploy the secret manager component, specify the secret resources to deploy to the target core devices. The core device's role alias must point to an IAM role that allows the core device to retrieve the secret resources to deploy. For more information, see Secret manager (p. 285).

- **Subscriptions**

  If your Lambda function publishes messages to the local publish/subscribe broker or to AWS IoT Core, specify `aws.greengrass.LegacySubscriptionRouter` as a component dependency when you import the function. When you deploy the legacy subscription router component, specify the subscriptions that the Lambda function uses. For more information, see Legacy subscription router (p. 194).

  **Note**

  This component is required only if your Lambda function uses the `publish()` function in the AWS IoT Greengrass Core SDK. If you update your Lambda function code to use the interprocess communication (IPC) interface in the V2 AWS IoT Device SDK, you don't need to deploy the legacy subscription router component. For more information, see the following interprocess communication (p. 396) services:

  - Publish/subscribe local messages (p. 410)
  - Publish/subscribe AWS IoT Core MQTT messages (p. 433)

- **Local volumes and devices**

  If your containerized Lambda function accesses local volumes or devices, specify those volumes and devices when you import the Lambda function. This feature doesn't require a component dependency.

- **Local shadows**

  If your Lambda function interacts with local shadows, you must update the Lambda function code to use the AWS IoT Device SDK V2. You must also specify `aws.greengrass.ShadowManager` as a component dependency when you import the function.

- **Access other AWS services**

  If your Lambda function uses AWS credentials to make requests to other AWS services, specify `aws.greengrass.TokenExchangeService` as a component dependency when you import the function. The core device's role alias must point to an IAM role that allows the core device to perform the AWS operations that the Lambda function uses. For more information, see Token exchange service (p. 311) and Authorize core devices to interact with AWS services (p. 627).

For more information, see Run AWS Lambda functions (p. 361).

**Run AWS IoT Greengrass connectors**

You can deploy AWS-provided components that offer the same functionality of AWS IoT Greengrass connectors. When you create the deployment, you can configure the connectors' parameters. For more information, see the following AWS IoT Greengrass V2 components that provide Greengrass connectors:

- CloudWatch metrics component (p. 153)
- AWS IoT Device Defender component (p. 170)
- Kinesis Data Firehose component (p. 180)
AWS IoT Greengrass Developer Guide, Version 2
Run V1 applications on V2

- Modbus-RTU protocol adapter component (p. 264)
- Amazon SNS component (p. 298)

AWS IoT Greengrass V2 doesn’t provide a component to replace the Docker application deployment connector, but you can create components that run Docker containers from images. For more information, see Run a Docker container (p. 331).

Run machine learning inference

AWS IoT Greengrass V2 provides sample Amazon SageMaker Neo DLR machine learning components and models. You can use these features for image classification and object detection. To use other machine learning frameworks, such as MXNet and TensorFlow, you can develop your own custom components that use these frameworks.

Connect V1 Greengrass devices

In AWS IoT Greengrass V2, Greengrass devices (or connected devices) are called client devices. AWS IoT Greengrass V2 support for client devices is backward-compatible with AWS IoT Greengrass V1, so you can connect V1 core devices to V2 core devices without changing their application code. To enable client devices to connect to a V2 core device, deploy Greengrass components that enable client device support, and associate the client devices to the core device. To relay messages between client devices, the AWS IoT Core cloud service, and Greengrass components (including Lambda functions), deploy and configure the MQTT bridge component (p. 280). You can deploy the IP detector component (p. 177) to automatically detect connectivity information, or you can manually manage endpoints. For more information, see Interact with local IoT devices (p. 477).
What's new in AWS IoT Greengrass Version 2

AWS IoT Greengrass Version 2 is a new major version release of AWS IoT Greengrass that introduces the following features:

- **Open source edge runtime**—The edge runtime is now open source and distributed under the Apache 2.0 license and available on GitHub. You can now view the AWS IoT Greengrass edge runtime code, which allows you to troubleshoot interactions with your application and helps you build more reliable and performant applications running on AWS IoT Greengrass. You can also customize and extend the AWS IoT Greengrass edge runtime to meet your specific hardware and software needs. For more information, see Open source AWS IoT Greengrass Core software (p. 761).

- **Improved modularity**—You can add or remove pre-built software components based on your use cases, and your device CPU and memory resources. For example, you can choose to include only pre-built AWS IoT Greengrass components, such as stream manager, when you need to process data streams with your application. Or, you can include only machine learning components when you want to perform machine learning inference locally on your devices. For more information, see Develop AWS IoT Greengrass components (p. 322) and AWS-provided components (p. 132).

- **New local development tools**—AWS IoT Greengrass includes a new command line interface (CLI) that enables you to locally develop and debug applications on your device. In addition, the new local debug console helps you visually debug applications on your device. With these new capabilities, you can develop and debug code on a test device before using the cloud to deploy to your production devices. For more information, see Greengrass CLI (p. 166) and Local debug console (p. 199).

- **Improved fleet deployment features**—AWS IoT Greengrass is now integrated with AWS IoT thing groups. This enables you to organize your devices in groups and manage application deployments across your devices with features that control rollout rates, timeouts, and rollbacks. For more information, see Deploy AWS IoT Greengrass components to devices (p. 383).

AWS IoT Greengrass release notes provide details about AWS IoT Greengrass releases—new features, updates and improvements, and general fixes. AWS IoT Greengrass has the following types of releases:

- New feature releases for AWS IoT Greengrass
- AWS IoT Greengrass Core software updates

This section contains all of the AWS IoT Greengrass V2 release notes, latest first, and includes major feature changes and significant bug fixes. For information about additional minor fixes, see the aws-greengrass organization on GitHub.

**Release notes**

- Release: AWS IoT Greengrass Core v2.4.0 software update on August 3, 2021 (p. 11)
- Release: AWS IoT Greengrass Core v2.3.0 software update on June 29, 2021 (p. 13)
- Release: AWS IoT Greengrass Core v2.2.0 software update on June 18, 2021 (p. 14)
- Release: AWS IoT Greengrass Core v2.1.0 software update on April 26, 2021 (p. 16)
- Release: AWS IoT Greengrass Core v2.0.5 software update on March 09, 2021 (p. 21)
- Release: AWS IoT Greengrass Core v2.0.4 software update on February 04, 2021 (p. 22)
Release: AWS IoT Greengrass Core v2.4.0 software update on August 3, 2021

This release provides version 2.4.0 of the Greengrass nucleus component, new AWS-provided components, and updates to AWS-provided components.

Release date: August 3, 2021

Release highlights

- **System resource limits**—The Greengrass nucleus component now supports system resource limits. You can configure the maximum amount of CPU and RAM usage that each component’s processes can use on the core device. For more information, see Configure system resource limits for components (p. 124).

- **Pause/resume components**—The Greengrass nucleus now supports pausing and resuming components. You can use the interprocess communication (IPC) library to develop custom components that pause and resume other components' processes. For more information, see PauseComponent (p. 449) and ResumeComponent (p. 450).

- **Install with AWS IoT fleet provisioning**—Use the new AWS IoT fleet provisioning plugin to install the AWS IoT Greengrass Core software on devices that connect to AWS IoT to provision required AWS resources. Devices use a claim certificate to provision. You can embed the claim certificate on devices during manufacturing, so each device can provision as soon as it comes online. For more information, see Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning (p. 62).

- **Install with custom provisioning**—Develop a custom provisioning plugin to provision required AWS resources when you install the AWS IoT Greengrass Core software on devices. You can create a Java application that runs during installation to set up Greengrass core devices for your custom use case. For more information, see Install AWS IoT Greengrass Core software with custom resource provisioning (p. 83).

Release details

- Public component updates (p. 11)

Public component updates

The following table lists AWS-provided components that include new and updated features.

**Important**

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all component dependencies for that component. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly. To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment (p. 384). For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA) (p. 129).

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus</td>
<td>Version 2.4.0 of the Greengrass nucleus (p. 136) is available.</td>
</tr>
</tbody>
</table>
## Public component updates

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>New features</strong></td>
<td></td>
</tr>
<tr>
<td>- Adds support for system resource limits. You can configure the maximum amount of CPU and RAM usage that each component's processes can use on the core device. For more information, see [Configure system resource limits for components](p. 124).</td>
<td></td>
</tr>
<tr>
<td>- Adds IPC operations to pause and resume components. For more information, see [PauseComponent](p. 449) and [ResumeComponent](p. 450).</td>
<td></td>
</tr>
<tr>
<td>- Adds support for provisioning plugins. You can specify a JAR file to run during installation to provision required AWS resources for a Greengrass core device. The Greengrass nucleus includes an interface that you can implement to develop custom provisioning plugins. For more information, see [Install AWS IoT Greengrass Core software with custom resource provisioning](p. 83).</td>
<td></td>
</tr>
<tr>
<td>- Adds the optional <code>thing-name-policy</code> argument to the AWS IoT Greengrass Core software installer. You can use this option to specify an existing or custom AWS IoT policy when you install the AWS IoT Greengrass Core software with automatic resource provisioning (p. 46).</td>
<td></td>
</tr>
<tr>
<td><strong>Bug fixes and improvements</strong></td>
<td></td>
</tr>
<tr>
<td>- Updates logging configuration on startup. This fixes an issue where the logging configuration wasn't applied on startup.</td>
<td></td>
</tr>
<tr>
<td>- Updates the nucleus loader symlink to point to the component store in the Greengrass root folder during installation. This update enables you to delete the JAR file and other nucleus artifacts that you download when you install the AWS IoT Greengrass Core software.</td>
<td></td>
</tr>
<tr>
<td>- Additional minor fixes and improvements. For more information, see the changelog on GitHub.</td>
<td></td>
</tr>
</tbody>
</table>

**Greengrass CLI**  
Version 2.4.0 of the [Greengrass CLI](p. 166) is available.

**New features**  
- Adds support for system resource limits. When you create a local deployment, you can configure the maximum amount of CPU and RAM usage that each component's processes can use on the core device. For more information, see [Configure system resource limits for components](p. 124) and the [deployment create command](p. 600).  

**AWS IoT fleet provisioning by claim**  
The AWS IoT fleet provisioning by claim plugin is now available. For more information, see [Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning](p. 62).

**New features**  
- Adds support to install the AWS IoT Greengrass Core software with AWS IoT fleet provisioning. During installation, devices connect to AWS IoT to provision required AWS resources and download device certificates to use for regular operations.
Release: AWS IoT Greengrass Core v2.3.0 software update on June 29, 2021

This release provides version 2.3.0 of the Greengrass nucleus component.

Release date: June 29, 2021

Release highlights

• Large configuration support—The Greengrass nucleus component now supports deployment documents up to 10 MB. You can now deploy larger configuration updates to Greengrass components.

  Note
  To use this feature, a core device's AWS IoT policy must allow the greengrass:GetDeploymentConfiguration permission. If you used the AWS IoT Greengrass Core software installer to provision resources (p. 46), your core device's AWS IoT policy allows greengrass:* , which includes this permission. For more information, see Device authentication and authorization for AWS IoT Greengrass (p. 610).

Release details

• Public component updates (p. 13)

Public component updates

The following table lists AWS-provided components that include new and updated features.

Important
When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all component dependencies for that component. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly. To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment (p. 384). For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA) (p. 129).

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus</td>
<td>Version 2.3.0 of the Greengrass nucleus (p. 136) is available.</td>
</tr>
<tr>
<td></td>
<td>New features</td>
</tr>
<tr>
<td></td>
<td>• Adds support for deployment configuration documents up to 10 MB, up from 7 KB (for deployments that target things) or 31 KB (for deployments that target thing groups).</td>
</tr>
<tr>
<td></td>
<td>To use this feature, a core device's AWS IoT policy must allow the greengrass:GetDeploymentConfiguration permission. If you used the AWS IoT Greengrass Core software installer to provision resources (p. 46), your core device's AWS IoT policy allows greengrass:* , which includes this permission. For more information, see Device authentication and authorization for AWS IoT Greengrass (p. 610).</td>
</tr>
</tbody>
</table>
Component | Details
--- | ---
• Adds the `iot:thingName` recipe variable. You can use this recipe variable to get the name of the core device's AWS IoT thing in a recipe. For more information, see Recipe variables (p. 354).

### Bug fixes and improvements
• Additional minor fixes and improvements. For more information, see the changelog on GitHub.

---

**Release: AWS IoT Greengrass Core v2.2.0 software update on June 18, 2021**

This release provides version 2.2.0 of the Greengrass nucleus component, new AWS-provided components, and updates to AWS-provided components.

**Release date:** June 18, 2021

**Release highlights**

- **Client device support**—The new AWS-provided client device components enable you to connect client devices to your core devices using cloud discovery. You can sync client devices with AWS IoT Core and interact with client devices in Greengrass components. For more information, see Interact with local IoT devices (p. 477).

- **Local shadow service**—The new shadow manager component enables the local shadow service on your core devices. You can use this shadow service to interact with local shadows while offline using the Greengrass interprocess communication (IPC) libraries in the AWS IoT Device SDK. You can also use the shadow manager component to synchronize local shadow states with AWS IoT Core. For more information, see Interact with device shadows (p. 515).

**Release details**

- Public component updates (p. 14)

### Public component updates

The following table lists AWS-provided components that include new and updated features.

**Important**

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all component dependencies for that component. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly. To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment (p. 384). For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA) (p. 129).

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus</td>
<td>Version 2.2.0 of the Greengrass nucleus (p. 136) is available.</td>
</tr>
<tr>
<td>Component</td>
<td>Details</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>New features</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Adds IPC operations for local shadow management.</td>
</tr>
<tr>
<td>Bug fixes and improvements</td>
<td>• Reduces the size of the JAR file.</td>
</tr>
<tr>
<td></td>
<td>• Reduces memory usage.</td>
</tr>
<tr>
<td></td>
<td>• Fixes issues where the log configuration wasn't updated in certain cases.</td>
</tr>
<tr>
<td></td>
<td>• Additional minor fixes and improvements. For more information, see the changelog on GitHub.</td>
</tr>
<tr>
<td><strong>Shadow manager</strong></td>
<td>Version 2.0.0 of the new shadow manager component (p. 294) is available.</td>
</tr>
<tr>
<td>New features</td>
<td>• Adds support for classic and named shadows.</td>
</tr>
<tr>
<td></td>
<td>• Adds support for local shadow management using IPC.</td>
</tr>
<tr>
<td></td>
<td>• Adds support for shadow synchronization with AWS IoT Core.</td>
</tr>
<tr>
<td><strong>Client device auth</strong></td>
<td>Version 2.0.0 of the new client device auth component (p. 146) is available.</td>
</tr>
<tr>
<td>New features</td>
<td>• Adds support for Greengrass client devices, which are local IoT devices that connect to a core device over MQTT.</td>
</tr>
<tr>
<td></td>
<td>• Adds support for authentication and authorization of client devices and their MQTT actions.</td>
</tr>
<tr>
<td><strong>Moquette MQTT broker</strong></td>
<td>Version 2.0.0 of the new Moquette MQTT broker component (p. 283) is available.</td>
</tr>
<tr>
<td>New features</td>
<td>• Adds support for a local Moquette MQTT broker that handles communication with client devices.</td>
</tr>
<tr>
<td><strong>MQTT bridge</strong></td>
<td>Version 2.0.0 of the new MQTT bridge component (p. 280) is available.</td>
</tr>
<tr>
<td>New features</td>
<td>• Adds support to relay messages between the local MQTT broker, the local Greengrass publish/subscribe broker, and the AWS IoT Core MQTT broker.</td>
</tr>
<tr>
<td><strong>IP detector</strong></td>
<td>Version 2.0.0 of the new IP detector component (p. 177) is available.</td>
</tr>
<tr>
<td>New features</td>
<td>• Adds support to report a core device's local MQTT broker endpoints to the AWS IoT Greengrass cloud service for client devices to connect.</td>
</tr>
<tr>
<td><strong>Log manager</strong></td>
<td>Version 2.1.1 of the log manager component (p. 206) is available.</td>
</tr>
<tr>
<td>Bug fixes and improvements</td>
<td>• Fixes an issue where the system log configuration wasn't updated in certain cases.</td>
</tr>
</tbody>
</table>
Component | Details
--- | ---
**DLR object detection** | Version 2.1.2 of the DLR object detection (p. 232) is available.
- Bug fixes and improvements
  - Fixes an image scaling issue that resulted in inaccurate bounding boxes in the sample DLR object detection inference results.

**TensorFlow Lite object detection** | Version 2.1.1 of the TensorFlow Lite object detection (p. 252) is available.
- Bug fixes and improvements
  - Fixes an image scaling issue that resulted in inaccurate bounding boxes in the sample TensorFlow Lite object detection inference results.

---

**Release: AWS IoT Greengrass Core v2.1.0 software update on April 26, 2021**

This release provides version 2.1.0 of the Greengrass nucleus component and updates AWS-provided components.

**Release date:** April 26, 2021

**Release highlights**

- **Docker Hub and Amazon Elastic Container Registry (Amazon ECR) integration**—The new Docker application manager component enables you to download public or private images from Amazon ECR. You can also use this component to download public images from Docker Hub and AWS Marketplace. For more information, see Run a Docker container (p. 331).

- **Dockerfile and Docker images for AWS IoT Greengrass Core software**—You can use the Greengrass Docker image to run AWS IoT Greengrass in a Docker container that uses Amazon Linux 2 as the base operating system. You can also use the AWS IoT Greengrass Dockerfile to build your own Greengrass image. For more information, see Run AWS IoT Greengrass Core software in a Docker container (p. 93).

- **Support for additional machine learning frameworks and platforms**—You can deploy sample machine learning inference components that use pre-trained models to perform sample image classification and object detection using TensorFlow Lite 2.5.0 and DLR 1.6.0. This release also extends sample machine learning support for Armv8 (AArch64) devices. For more information, see Perform machine learning inference (p. 558).

**Release details**

- Platform support updates (p. 16)
- Public component updates (p. 17)

**Platform support updates**

<table>
<thead>
<tr>
<th>Platform</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docker</td>
<td>A Dockerfile and Docker image for AWS IoT Greengrass are now available.</td>
</tr>
</tbody>
</table>
Dockerfile

AWS IoT Greengrass provides a Dockerfile to build a container image that has AWS IoT Greengrass Core software and dependencies installed on an Amazon Linux 2 (x86_64) base image. You can modify the base image in the Dockerfile to run AWS IoT Greengrass on a different platform architecture.

Docker image

AWS IoT Greengrass provides a pre-built Docker image that has AWS IoT Greengrass Core software and dependencies installed on an Amazon Linux 2 (x86_64) base image.

For more information, see Run AWS IoT Greengrass Core software in a Docker container (p. 93).

Public component updates

The following table lists AWS-provided components that include new and updated features.

Important
When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all component dependencies for that component. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly. To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment (p. 384). For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA) (p. 129).

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus</td>
<td>Version 2.1.0 of the Greengrass nucleus (p. 136) is available. New features</td>
</tr>
<tr>
<td></td>
<td>• Supports downloading Docker images from private repositories in Amazon ECR.</td>
</tr>
<tr>
<td></td>
<td>• Adds the following parameters to customize the MQTT configuration on core devices:</td>
</tr>
<tr>
<td></td>
<td>• maxInFlightPublishes – The maximum number of unacknowledged MQTT QoS 1 messages that can be in flight at the same time.</td>
</tr>
<tr>
<td></td>
<td>• maxPublishRetry – The maximum number of times to retry a message that fails to publish.</td>
</tr>
<tr>
<td></td>
<td>• Adds the fleetstatusservice configuration parameter to configure the interval at which the core device publishes device status to the AWS Cloud.</td>
</tr>
<tr>
<td></td>
<td>• Additional minor fixes and improvements. For more information, see the changelog on GitHub.</td>
</tr>
<tr>
<td></td>
<td>Bug fixes and improvements</td>
</tr>
<tr>
<td></td>
<td>• Fixes an issue that caused shadow deployments to be duplicated when the nucleus restarts.</td>
</tr>
</tbody>
</table>
Public component updates

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Fixes an issue that caused the nucleus to crash when it encountered a service load exception.</td>
</tr>
<tr>
<td></td>
<td>• Improves component dependency resolution to fail a deployment that includes a circular dependency.</td>
</tr>
<tr>
<td></td>
<td>• Fixes an issue that prevented a plugin component from being redeployed if that component had been previously removed from the core device.</td>
</tr>
<tr>
<td></td>
<td>• Fix an issue that caused the HOME environment variable to be set to the /greengrass/v2/work directory for Lambda components or for components that run as root. The HOME variable is now correctly set to the home directory for the user that runs the component.</td>
</tr>
<tr>
<td></td>
<td>• Additional minor fixes and improvements. For more information, see the changelog on GitHub.</td>
</tr>
<tr>
<td>Docker application manager</td>
<td>Version 2.0.0 of the new Docker application manager component (p. 162) is available.</td>
</tr>
<tr>
<td></td>
<td>New features</td>
</tr>
<tr>
<td></td>
<td>• Manages credentials to download images from private repositories in Amazon ECR.</td>
</tr>
<tr>
<td></td>
<td>• Downloads public images from Amazon ECR, Docker Hub, and AWS Marketplace.</td>
</tr>
<tr>
<td>Lambda launcher</td>
<td>Version 2.0.4 of the Lambda launcher component (p. 188) is available.</td>
</tr>
<tr>
<td></td>
<td>Bug fixes and improvements</td>
</tr>
<tr>
<td></td>
<td>• Fixes an issue where the component doesn't correctly pass AddGroupOwner to the Lambda function container.</td>
</tr>
<tr>
<td>Legacy subscription router</td>
<td>Version 2.1.0 of the legacy subscription router component (p. 194) is available.</td>
</tr>
<tr>
<td></td>
<td>Bug fixes and improvements</td>
</tr>
<tr>
<td></td>
<td>• Adds support to specify component names instead of ARNs for source and target. If you specify a component name for a subscription, you don't need to reconfigure the subscription each time the version of the Lambda function changes.</td>
</tr>
<tr>
<td>Local debug console</td>
<td>Version 2.1.0 of the local debug console component (p. 199) is available.</td>
</tr>
<tr>
<td></td>
<td>New features</td>
</tr>
<tr>
<td></td>
<td>• Uses HTTPS to secure your connection to the local debug console. HTTPS is enabled by default.</td>
</tr>
<tr>
<td></td>
<td>Bug fixes and improvements</td>
</tr>
<tr>
<td></td>
<td>• You can dismiss flashbar messages in the configuration editor.</td>
</tr>
<tr>
<td>Component</td>
<td>Details</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Log manager</strong></td>
<td>Version 2.1.0 of the log manager component (p. 206) is available.</td>
</tr>
<tr>
<td>Bug fixes and improvements</td>
<td>Use defaults for logFileDirectoryPath and logFileRegex that work for Greengrass components that print to standard output (stdout) and standard error (stderr).</td>
</tr>
<tr>
<td></td>
<td>Correctly route traffic through a configured network proxy when uploading logs to CloudWatch Logs.</td>
</tr>
<tr>
<td></td>
<td>Correctly handle colon characters (:) in log stream names. CloudWatch Logs log stream names don't support colons.</td>
</tr>
<tr>
<td></td>
<td>Simplify log stream names by removing thing group names from the log stream.</td>
</tr>
<tr>
<td></td>
<td>Remove an error log message that prints during normal behavior.</td>
</tr>
<tr>
<td><strong>DLR image classification</strong></td>
<td>Version 2.1.1 of the DLR image classification (p. 226) component is available.</td>
</tr>
<tr>
<td>New features</td>
<td>Use Deep Learning Runtime v1.6.0.</td>
</tr>
<tr>
<td></td>
<td>Add support for sample image classification on Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano.</td>
</tr>
<tr>
<td></td>
<td>Enable camera integration for sample inference. Use the new UseCamera configuration parameter to enable the sample inference code to access the camera on your Greengrass core device and run inference locally on the captured image.</td>
</tr>
<tr>
<td></td>
<td>Add support for publishing inference results to the AWS Cloud. Use the new PublishResultsOnTopic configuration parameter to specify the topic on which you want to publish results.</td>
</tr>
<tr>
<td></td>
<td>Add the new ImageDirectory configuration parameter that enables you to specify a custom directory for the image on which you want to perform inference.</td>
</tr>
<tr>
<td>Bug fixes and improvements</td>
<td>Write inference results to the component log file instead of a separate inference file.</td>
</tr>
<tr>
<td></td>
<td>Use the AWS IoT Greengrass Core software logging module to log component output.</td>
</tr>
<tr>
<td></td>
<td>Use the AWS IoT Device SDK to read the component configuration and apply configuration changes.</td>
</tr>
</tbody>
</table>
## Public component updates

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DLR object detection</strong></td>
<td>Version 2.1.1 of the DLR object detection (p. 232) component is available.</td>
</tr>
<tr>
<td></td>
<td><strong>New features</strong></td>
</tr>
<tr>
<td></td>
<td>• Use Deep Learning Runtime v1.6.0.</td>
</tr>
<tr>
<td></td>
<td>• Add support for sample object detection on Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano.</td>
</tr>
<tr>
<td></td>
<td>• Enable camera integration for sample inference. Use the new UseCamera configuration parameter to enable the sample inference code to access the camera on your Greengrass core device and run inference locally on the captured image.</td>
</tr>
<tr>
<td></td>
<td>• Add support for publishing inference results to the AWS Cloud. Use the new PublishResultsOnTopic configuration parameter to specify the topic on which you want to publish results.</td>
</tr>
<tr>
<td></td>
<td>• Add the new ImageDirectory configuration parameter that enables you to specify a custom directory for the image on which you want to perform inference.</td>
</tr>
<tr>
<td></td>
<td><strong>Bug fixes and improvements</strong></td>
</tr>
<tr>
<td></td>
<td>• Write inference results to the component log file instead of a separate inference file.</td>
</tr>
<tr>
<td></td>
<td>• Use the AWS IoT Greengrass Core software logging module to log component output.</td>
</tr>
<tr>
<td></td>
<td>• Use the AWS IoT Device SDK to read the component configuration and apply configuration changes.</td>
</tr>
<tr>
<td><strong>DLR image classification model store</strong></td>
<td>Version 2.1.1 of the DLR image classification model store (p. 238) component is available.</td>
</tr>
<tr>
<td></td>
<td><strong>New features</strong></td>
</tr>
<tr>
<td></td>
<td>• Add a sample ResNet-50 image classification model for Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano.</td>
</tr>
<tr>
<td><strong>DLR object detection model store</strong></td>
<td>Version 2.1.1 of the DLR object detection model store (p. 241) component is available.</td>
</tr>
<tr>
<td></td>
<td><strong>New features</strong></td>
</tr>
<tr>
<td></td>
<td>• Add a sample YOLOv3 object detection model for Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano.</td>
</tr>
<tr>
<td><strong>DLR installer</strong></td>
<td>Version 1.6.1 of the DLR (p. 243) component is available.</td>
</tr>
<tr>
<td></td>
<td><strong>New features</strong></td>
</tr>
<tr>
<td></td>
<td>• Install Deep Learning Runtime v1.6.0 and its dependencies.</td>
</tr>
<tr>
<td></td>
<td>• Add support for installing DLR on Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano.</td>
</tr>
<tr>
<td></td>
<td><strong>Bug fixes and improvements</strong></td>
</tr>
<tr>
<td></td>
<td>• Install the AWS IoT Device SDK in the virtual environment to read the component configuration and apply configuration changes.</td>
</tr>
<tr>
<td></td>
<td>• Additional minor bug fixes and improvements.</td>
</tr>
</tbody>
</table>
Release: AWS IoT Greengrass Core v2.0.5 software update on March 09, 2021

This release provides version 2.0.5 of the Greengrass nucleus component and updates AWS-provided components. It fixes an issue with network proxy support and an issue with the Greengrass data plane endpoint in AWS China Regions.

Release date: March 09, 2021

Public component updates

The following table lists AWS-provided components that include new and updated features.

Important
When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all component dependencies for that component. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add...
new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly. To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment (p. 384). For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA) (p. 129).

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus</td>
<td>Version 2.0.5 of the Greengrass nucleus (p. 136) is available.</td>
</tr>
<tr>
<td></td>
<td>Bug fixes and improvements</td>
</tr>
<tr>
<td></td>
<td>• Correctly routes traffic through a configured network proxy when</td>
</tr>
<tr>
<td></td>
<td>downloading AWS-provided components.</td>
</tr>
<tr>
<td></td>
<td>• Use the correct Greengrass data plane endpoint in AWS China Regions.</td>
</tr>
</tbody>
</table>

Release: AWS IoT Greengrass Core v2.0.4 software update on February 04, 2021

This release provides version 2.0.4 of the Greengrass nucleus component. It includes the new greengrassDataPlanePort parameter to configure HTTPS communication over port 443 and fixes bugs. The minimal IAM policy now requires the iam:GetPolicy and sts:GetCallerIdentity when the AWS IoT Greengrass Core software installer is run with --provision true.

Release date: February 04, 2021

Public component updates

The following table lists AWS-provided components that include new and updated features.

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus</td>
<td>Version 2.0.4 of the Greengrass nucleus (p. 136) is available.</td>
</tr>
<tr>
<td></td>
<td>New features</td>
</tr>
<tr>
<td></td>
<td>• Enables HTTPS traffic over port 443. You can use the new greengrassDataPlanePort configuration parameter for version 2.0.4 of the nucleus component to configure HTTPS communication to travel over port 443 instead of the default port 8443. For more information, see Configure HTTPS over port 443 (p. 126).</td>
</tr>
</tbody>
</table>
Public component updates

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Adds the work path recipe variable. You can use this recipe variable to get the path to components' work folders, which you can use to share files between components and their dependencies. For more information, see the work path recipe variable (p. 355).</td>
</tr>
</tbody>
</table>

Bug fixes and improvements

- Prevents the creation of the token exchange AWS Identity and Access Management (IAM) role policy if a role policy already exists.

  As a result of this change, the installer now requires the `iam:GetPolicy` and `sts:GetCallerIdentity` when run with `--provision true`. For more information, see Minimal IAM policy for installer to provision resources (p. 629).

- Correctly handles the cancellation of a deployment that has not yet been registered successfully.

- Updates the configuration to remove older entries with newer timestamps when rolling back a deployment.

- Additional minor fixes and improvements. For more information, see the changelog on GitHub.
Getting started with AWS IoT Greengrass V2

You can complete this getting started tutorial to learn about the basic features of AWS IoT Greengrass V2. In this tutorial, you do the following:

- Install and configure the AWS IoT Greengrass Core software on a Linux device, such as a Raspberry Pi. This device is an AWS IoT Greengrass core device.
- Develop a Hello World component on your Greengrass core device. Components are software modules that run on Greengrass core devices.
- Upload that component to AWS IoT Greengrass V2 in the AWS Cloud.
- Deploy that component from the AWS Cloud to your Greengrass core device.

**Note**
This tutorial describes how to set up a development environment and explore the features of AWS IoT Greengrass. For more information about how to set up and configure production devices, see the following:

- Setting up AWS IoT Greengrass core devices (p. 41)
- Install the AWS IoT Greengrass Core software (p. 45)

You can expect to spend 20-30 minutes on this tutorial.

**Prerequisites**

To complete this getting started tutorial, you need the following:

- An AWS account. If you don't have one, see Set up an AWS account (p. 44).
- The use of an AWS Region that supports AWS IoT Greengrass V2. For the list of supported Regions, see AWS IoT Greengrass V2 endpoints and quotas in the AWS General Reference.
- An AWS Identity and Access Management (IAM) user with administrator permissions.
- A Windows, macOS, or Unix-like development computer with an internet connection.
- A device that meets the Requirements to install and run the AWS IoT Greengrass Core software v2.0 (p. 42) and has an internet connection to the same network as your development computer. We recommend that you use a Raspberry Pi with Raspberry Pi OS (previously called Raspbian).
- Python 3.5 or later installed on the device.
- AWS Command Line Interface (AWS CLI) installed and configured with credentials on your development computer and on your device. Make sure you use the same AWS Region to configure the AWS CLI on your development computer and on your device. To use AWS IoT Greengrass V2 with the AWS CLI, you must have one of the following versions or later:
  - Minimum AWS CLI V1 version: v1.18.197
  - Minimum AWS CLI V2 version: v2.1.11

  **Tip**
  You can run the following command to check the version of the AWS CLI that you have.
Set up your environment

Follow the steps in this section to set up a Linux device to use as your AWS IoT Greengrass core device. These steps assume that you use a Raspberry Pi with Raspberry Pi OS. If you use a different device or operating system, consult the relevant documentation for your device.

To set up a Raspberry Pi for AWS IoT Greengrass V2

1. Enable SSH on your Raspberry Pi to remotely connect to it. For more information, see SSH (Secure shell) in the Raspberry Pi Documentation.
2. Find the IP address of your Raspberry Pi to connect to it with SSH. To do so, you can run the following command on your Raspberry Pi.

```
hostname -I
```
3. Connect to your Raspberry Pi with SSH. On your development computer, run the following command. Replace `username` with the name of the user to sign in, and replace `pi-ip-address` with the IP address that you found in the previous step.

```
ssh username@pi-ip-address
```

The default Raspberry Pi user name and password are `pi` and `raspberry`, respectively.

**Important**

If your development computer uses an earlier version of Windows, you might not have the `ssh` command, or you might have `ssh` but can't connect to your Raspberry Pi. To connect to your Raspberry Pi, you can install and configure PuTTY, which is a no-cost, open source SSH client. Consult the PuTTY documentation to connect to your Raspberry Pi.

4. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. On your Raspberry Pi, use the following commands to install Java 11.

```
sudo apt install default-jdk
```
When the installation completes, run the following command to verify that Java runs on your Raspberry Pi.

```
java --version
```

The command prints the version of Java that runs on the device. The output might look similar to the following example.

```
openjdk version "11.0.9.1" 2020-11-04
```
Install the AWS IoT Greengrass Core software

Follow the steps in this section to set up your Raspberry Pi as a AWS IoT Greengrass core device that you can use for local development. In this section, you download and run an installer that does the following to configure the AWS IoT Greengrass Core software for your device:

- Installs the Greengrass nucleus component, which is the only mandatory component and the minimum requirement to run the AWS IoT Greengrass Core software on a device. For more information, see Greengrass nucleus component (p. 136).
- Registers your device as an AWS IoT thing and downloads a digital certificate that allows your device to connect to AWS. For more information, see Device authentication and authorization for AWS IoT Greengrass (p. 610).
- Adds the device’s AWS IoT thing to a thing group, which is a group or fleet of AWS IoT things. Thing groups enable you to manage fleets of Greengrass core devices. When you deploy software components to your devices, you can choose to deploy to individual devices or to groups of devices. For more information, see Managing devices with AWS IoT in the AWS IoT Core Developer Guide.
- Creates the IAM role that allows your Greengrass core device to interact with AWS services. By default, this role allows your device to interact with AWS IoT and send logs to Amazon CloudWatch Logs. For more information, see Authorize core devices to interact with AWS services (p. 627).
- Installs the AWS IoT Greengrass command line interface (greengrass-cli), which you can use to test custom components that you develop on the core device. For more information, see Greengrass Command Line Interface (p. 595).

To install and configure the AWS IoT Greengrass Core software

1. On your AWS IoT Greengrass core device (your Raspberry Pi), run the following command to switch to the home directory.

   cd ~

2. Run the following command to download the AWS IoT Greengrass Core software.

   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip > greengrass-nucleus-latest.zip

   By downloading this software, you agree to the Greengrass Core Software License Agreement.

3. Run the following command to unzip the software and remove the ZIP file. Replace GreengrassInstaller with the name of the destination folder.

   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-nucleus-latest.zip

4. Provide your AWS credentials so that the installer can provision the AWS IoT and IAM resources for your core device. To increase security, you can get credentials for an IAM role that allows only the minimum permissions necessary to provision. For more information, see Minimal IAM policy for installer to provision resources (p. 629).

   Do one of the following to retrieve credentials and provide them to the AWS IoT Greengrass Core software:
• Use long-term credentials from an IAM user:
  a. Provide the access key ID and secret access key for your IAM user. For more information about how to retrieve long-term credentials, see Managing access keys for IAM users in the IAM User Guide.
  b. Run the following commands to provide the credentials to the AWS IoT Greengrass Core software.

```bash
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtznFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

• (Recommended) Use temporary security credentials from an IAM role:
  a. Provide the access key ID, secret access key, and session token from an IAM role that you assume. For more information about how to retrieve these credentials, see Using temporary security credentials with the AWS CLI in the IAM User Guide.
  b. Run the following commands to provide the credentials to the AWS IoT Greengrass Core software.

```bash
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtznFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

The AWS IoT Greengrass Core software doesn't save or store your credentials.

5. Run the following command to launch the AWS IoT Greengrass Core software installer. If your core device has systemd, this command installs the software as a system service that runs on boot. This command also specifies to use the `ggc_user` system user and `ggc_group` system group to run software components on the core device. The installer creates this default user and group for you. Replace argument values in your command as follows.

a. `/greengrass/v2`: The path to the root folder to use to install the AWS IoT Greengrass Core software.

b. `GreengrassInstaller`. The path to the folder where you unpacked the AWS IoT Greengrass Core software installer.

c. `region`. The AWS Region in which to find or create resources.

d. `MyGreengrassCore`. The name of the AWS IoT thing for your Greengrass core device. If the thing doesn't exist, the installer creates it. The installer downloads the certificates to authenticate as the AWS IoT thing. For more information, see Device authentication and authorization for AWS IoT Greengrass (p. 610).

  **Note**
  The thing name can't contain colon (:) characters.

e. `MyGreengrassCoreGroup`. The name of AWS IoT thing group for your Greengrass core device. If the thing group doesn't exist, the installer creates it and adds the thing to it. If the thing group exists and has an active deployment, the core device downloads and runs the software that the deployment specifies.

  **Note**
  The thing group name can't contain colon (:) characters.

f. `GreengrassV2IoTThingPolicy`. The name of the AWS IoT policy that allows the Greengrass core devices to communicate with AWS IoT and AWS IoT Greengrass. If the AWS IoT policy doesn't exist, the installer creates a permissive AWS IoT policy with this name. You can restrict this policy's permissions for your use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices (p. 614).

g. `GreengrassV2TokenExchangeRole`. The name of the IAM role that allows the Greengrass core device to get temporary AWS credentials. If the role doesn't exist, the installer creates it.
and creates and attaches a policy named `GreengrassV2TokenExchangeRoleAccess`. For more information, see Authorize core devices to interact with AWS services (p. 627).

h. `GreengrassCoreTokenExchangeRoleAlias`. The alias to the IAM role that allows the Greengrass core device to get temporary credentials later. If the role alias doesn't exist, the installer creates it and points it to the IAM role that you specify. For more information, see Authorize core devices to interact with AWS services (p. 627).

```bash
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
-jar ./GreengrassInstaller/lib/Greengrass.jar \
--aws-region region \
--thing-name MyGreengrassCore \
--thing-group-name MyGreengrassCoreGroup \
--thing-policy-name GreengrassV2IoTThingPolicy \
--tes-role-name GreengrassV2TokenExchangeRole \
--tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias \
--component-default-user ggc_user:ggc_group \
--provision true \
--setup-system-service true \
--deploy-dev-tools true
```

**Note**
If you are running AWS IoT Greengrass on a device with limited memory, you can control the amount of memory that AWS IoT Greengrass Core software uses. To control memory allocation, you can set JVM heap size options in the `jvmOptions` configuration parameter in your nucleus component. For more information, see Control memory allocation with JVM options (p. 121).

When you run this command, you should see the following messages to indicate that the installer succeeded.

```
Successfully configured Nucleus with provisioned resource details!
Configured Nucleus to deploy aws.greengrass.Cli component
Successfully set up Nucleus as a system service
```

**Note**
If your system doesn't have systemd, the installer won't set up the software as a system service, and you won't see that success message.

6. The local development tools can take up to a minute to deploy. You can run the following command to check the status of the deployment. Replace `MyGreengrassCore` with the name of your core device.

```bash
aws greengrassv2 list-effective-deployments --core-device-thing-name MyGreengrassCore
```

The `coreDeviceExecutionStatus` indicates the status of the deployment to the core device. When the status is `SUCCEEDED`, run the following command to verify that the Greengrass CLI is installed and runs. Replace `/greengrass/v2` with the path to the root folder.

```
/greengrass/v2/bin/greengrass-cli help
```

The command outputs help information for the Greengrass CLI. If the `greengrass-cli` isn't found, the deployment might have failed to install the Greengrass CLI. For more information, see Troubleshooting AWS IoT Greengrass V2 (p. 751).

You can also run the following command to manually deploy the AWS IoT Greengrass CLI to your device.

---

28
Install the AWS IoT Greengrass Core software

- Replace `region` with the AWS Region that you use. Make sure that you use the same AWS Region that you used to configure the AWS CLI on your device.
- Replace `account-id` with your AWS account ID.
- Replace `MyGreengrassCore` with the name of your core device.

**Linux, macOS, or Unix**

```bash
aws greengrassv2 create-deployment \
   --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" \
   --components '{
     "aws.greengrass.Cli": {
       "componentVersion": "2.4.0"
     }
   }'
```

**Windows (CMD)**

```bash
aws greengrassv2 create-deployment ^
   --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" ^
   --components "{"aws.greengrass.Cli":{""componentVersion":""2.4.0""}}"
```

**Windows (PowerShell)**

```bash
aws greengrassv2 create-deployment `^
   --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" `^
   --components '{"aws.greengrass.Cli":{""componentVersion":""2.4.0""}}'
```

**Tip**

You can add `/greengrass/v2/bin` to your PATH environment variable to run greengrass-cli without its absolute path.

7. If you installed the software as a system service, the installer runs the software for you. Otherwise, you must run the software. To see if the installer set up the software as a system service, look for the following line in the installer output.

```
Successfully set up Nucleus as a system service
```

If you don't see this message, do the following to run the software:

a. Run the following command to run the software.

```bash
sudo /greengrass/v2/alts/current/distro/bin/loader
```

The software prints the following message if it launches successfully.

```
Launched Nucleus successfully.
```

b. You must leave the current terminal session open to keep the AWS IoT Greengrass Core software running. On your development computer, run the following command to open a second SSH session that you can use to run additional commands on the core device. Replace `username` with the name of the user to sign in, and replace `pi-ip-address` with the IP address of the device.

```
```
Create your first component

A component is a software module that runs on AWS IoT Greengrass core devices. Components enable you to create and manage complex applications as discrete building blocks that you can reuse from one Greengrass core device to another. Every component is composed of a recipe and artifacts.

- **Recipes**

  Every component contains a recipe file, which defines its metadata. The recipe also specifies the component's configuration parameters, component dependencies, lifecycle, and platform compatibility. The component lifecycle defines the commands that install, run, and shut down the component. For more information, see AWS IoT Greengrass component recipe reference (p. 341).

  You can define recipes in JSON or YAML format.

- **Artifacts**

  Components can have any number of artifacts, which are component binaries. Artifacts can include scripts, compiled code, static resources, and any other files that a component consumes. Components can also consume artifacts from component dependencies.

With AWS IoT Greengrass, you can develop and test components on your Greengrass core device without interaction with the AWS Cloud. When you complete your component, you can upload it to AWS IoT Greengrass to deploy it to other devices. For more information, see Develop AWS IoT Greengrass components (p. 322).

In this section, you learn how to create and run a basic Hello World component.

**To create a Hello World component**

1. On your Greengrass core device, run the following command to change to the AWS IoT Greengrass installation folder to use for local development. Replace `~/GreengrassInstaller` with the path to the installation folder.

   ```
   cd ~/GreengrassInstaller
   ```

2. AWS IoT Greengrass expects that component recipes and artifacts are in separate folders.

   Run the following command to create a folder for the component recipe.

   ```
   mkdir recipes
   ```

3. Run the following command to create the recipe file and open it in a text editor.

   ```
   ssh username@pi-ip-address
   ```

   For more information about how to interact with the Greengrass system service, see Configure AWS IoT Greengrass as a system service (p. 119).
Create your first component

Paste the following recipe into the file.

**JSON**

```json
{
"RecipeFormatVersion": "2020-01-25",
"ComponentName": "com.example.HelloWorld",
"ComponentVersion": "1.0.0",
"ComponentDescription": "My first AWS IoT Greengrass component.",
"ComponentPublisher": "Amazon",
"ComponentConfiguration": {
"DefaultConfiguration": {
"Message": "world"
}
},
"Manifests": [
{
"Platform": {
  "os": "linux"
},
"Lifecycle": {
"Run": "python3 -u {artifacts:path}/hello_world.py '{configuration:/Message}'"
}
}
]
}
```

**YAML**

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: My first AWS IoT Greengrass component.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    Message: world
Manifests:
- Platform:
  os: linux
Lifecycle:
  Run: |
    python3 -u {artifacts:path}/hello_world.py '{configuration:/Message}'
```

This recipe's ComponentConfiguration section defines a parameter, Message, that defaults to world. The Manifests section defines a manifest, which is a set of lifecycle instructions and artifacts for a platform. You can define multiple manifests to specify different install instructions.
for various platforms, for example. In the manifest, the Lifecycle section instructs the Greengrass core device to run the Hello World script with the Message parameter value as an argument.

4. Run the following command to create a folder for the component artifacts.

```bash
mkdir -p artifacts/com.example.HelloWorld/1.0.0
```

**Important**
You must use the following format for the artifact folder path. Include the component name and version that you specify in the recipe.

```
artifacts/componentName/componentVersion/
```

5. Run the following command to create a Python script artifact file and open it in a text editor.

```bash
nano artifacts/com.example.HelloWorld/1.0.0/hello_world.py
```

Paste the following Python script into the file.

```python
import sys
import datetime

message = "Hello, %s! Current time: %s.\n" % (sys.argv[1], datetime.datetime.now())

# Print the message to stdout.
print(message)

# Append the message to the log file.
with open('/tmp/Greengrass_HelloWorld.log', 'a') as f:
    print(message, file=f)
```

This Python script logs a hello message and the current time to /tmp/Greengrass_HelloWorld.log.

6. Use the local AWS IoT Greengrass CLI to manage components on your Greengrass core device.

Run the following command to deploy the component to the AWS IoT Greengrass core. Replace `~/GreengrassInstaller` with your AWS IoT Greengrass V2 installation folder.

```bash
sudo /greengrass/v2/bin/greengrass-cli deployment create \
--recipeDir -~/GreengrassInstaller/recipes \
--artifactDir -~/GreengrassInstaller/artifacts \
--merge "com.example.HelloWorld=1.0.0"
```

This command adds the component that uses the recipe in recipes and the Python script in artifacts. The --merge option adds or updates the component and version that you specify.

7. Run the following command to verify that the Hello World component runs and logs messages.

```bash
tail -f /tmp/Greengrass_HelloWorld.log
```

You should see messages similar to the following example.

```
```
**Note**
If the `tail` command tells you that the file doesn't exist, the local deployment may not be complete yet. If the file doesn't exist within 15 seconds, the deployment likely failed. This can occur if your recipe isn't valid, for example. Run the following command to view the AWS IoT Greengrass core log file. This file includes logs from the Greengrass core device's deployment service.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

You can also view the log file for your Hello World component.

```
sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
```

For more information, see Troubleshooting AWS IoT Greengrass V2 (p. 751).

8. Modify the local component to iterate and test your code.

Run the following command to edit `hello_world.py`.

```
nano artifacts/com.example.HelloWorld/1.0.0/hello_world.py
```

9. Add the following code at line 5 to edit the message that the AWS IoT Greengrass core logs.

```python
message += " Greetings from your first Greengrass component."
```

The `hello_world.py` script should now have the following contents.

```python
import sys
import datetime

message = "Hello, %s! Current time: %s.\n" % (sys.argv[1], datetime.datetime.now())
message += " Greetings from your first Greengrass component."
# Print the message to stdout.
print(message)
# Append the message to the log file.
with open('/tmp/Greengrass_HelloWorld.log', 'a') as f:
    print(message, file=f)
```

10. Run the following command to update the component with your changes.

```
sudo /greengrass/v2/bin/greengrass-cli deployment create \
    --recipeDir ~/GreengrassInstaller/recipes \
    --artifactDir ~/GreengrassInstaller/artifacts \
    --merge "com.example.HelloWorld=1.0.0"
```

This command applies the latest Hello World artifact to the AWS IoT Greengrass core runtime.

11. Run the following command to restart the component. When you restart a component, the core device uses the latest changes.

```
sudo /greengrass/v2/bin/greengrass-cli component restart \
    --names "com.example.HelloWorld"
```

12. Check the log again to verify that the Hello World component outputs the new message.

```
tail -f /tmp/Greengrass_HelloWorld.log
```
You should see messages similar to the following example.

```
```

13. You can update the component's configuration parameters to test different configurations. When you deploy a component, you can specify a configuration update, which defines how to modify the component's configuration on the core device. You can specify which configuration values to reset to default values and the new configuration values to merge onto the core device. For more information, see Update component configurations (p. 390).

Do the following:

a. Create a file called `hello-world-config-update.json` to contain the configuration update.

```
nano hello-world-config-update.json
```

b. Copy the following JSON object into the file. This JSON object defines a configuration update that merges the value `friend` to the `Message` parameter to update its value. This configuration update doesn't specify any values to reset. You don't need to reset the `Message` parameter because the merge update replaces the existing value.

```
{
  "com.example.HelloWorld": {
    "MERGE": {
      "Message": "friend"
    }
  }
}
```

c. Run the following command to deploy the configuration update to the Hello World component.

```
sudo /greengrass/v2/bin/greengrass-cli deployment create --merge "com.example.HelloWorld=1.0.0" --update-config hello-world-config-update.json
```

d. Check the log again to verify that the Hello World component outputs the new message.

```
tail -f /tmp/Greengrass_HelloWorld.log
```

You should see messages similar to the following example.

```
```

14. After you finish testing your component, remove it from your core device. Run the following command.

```
sudo /greengrass/v2/bin/greengrass-cli deployment create --remove="com.example.HelloWorld"
```

**Important**

This step is required for you to deploy the component back to the core device after you upload it to AWS IoT Greengrass. Otherwise, the deployment fails with a version compatibility error because the local deployment specifies a different version of the component.
Run the following command and verify that the `com.example.HelloWorld` component doesn't appear in the list of components on your device.

```
sudo /greengrass/v2/bin/greengrass-cli component list
```

Your Hello World component is complete, and you can now upload it to the AWS IoT Greengrass service. Then, you can deploy the component to AWS IoT Greengrass core devices.

### Upload your component

When you finish a component, you can upload it to the AWS IoT Greengrass service in the AWS Cloud. AWS IoT Greengrass provides a component management service that hosts your components so that you can deploy them to individual devices or fleets of devices. To upload a component to AWS IoT Greengrass, you complete the following steps:

- Upload component artifacts to an S3 bucket.
- Add each artifact's Amazon Simple Storage Service (Amazon S3) URI to the component recipe.
- Create a component in AWS IoT Greengrass from the component recipe.

In this section, you complete these steps on your AWS IoT Greengrass core device to upload your Hello World component to AWS IoT Greengrass.

#### To upload your Hello World component

1. Use an S3 bucket in your AWS account to host AWS IoT Greengrass component artifacts. When you deploy the component to a core device, the device downloads the component's artifacts from the bucket.

   You can use an existing S3 bucket, or run the following command to create a bucket. This command creates a bucket with your AWS account ID and AWS Region to form a unique bucket name. Replace `123456789012` with your AWS account ID and `region` with the AWS Region that you use for this tutorial.

   ```
   aws s3 mb s3://greengrass-component-artifacts-123456789012-region
   ```

   The command outputs the following information if the request succeeds.

   ```
   make_bucket: greengrass-component-artifacts-123456789012-region
   ```

2. Allow the core device to access component artifacts in the S3 bucket. Each core device has a core device IAM role (p. 627) that allows it to interact with AWS IoT and send logs to the AWS Cloud. This device role doesn't allow access to S3 buckets by default, so you must create and attach a policy that allows the core device to retrieve component artifacts from the S3 bucket.

   If your device's role already allows access to the S3 bucket, you can skip this step. Otherwise, create an IAM policy that allows access and attach it to the role, as follows:

   a. Create a file called `component-artifact-policy.json` and copy the following JSON into the file. This policy allows access to all files in the S3 bucket that you created in the previous step. Replace `DOC EXAMPLE BUCKET` with the name of the bucket to use.

   ```
   {
   ```
b. Run the following command to create the policy from the policy document in component-artifact-policy.json.

```
aws iam create-policy
--policy-name MyGreengrassV2ComponentArtifactPolicy
--policy-document file://component-artifact-policy.json
```

Copy the policy Amazon Resource Name (ARN) from the policy metadata in the output. You use this ARN to attach this policy to the core device role in the next step.

c. Run the following command to attach the policy to the core device role. Replace `GreengrassV2TokenExchangeRole` with the name of the role that you specified when you ran the AWS IoT Greengrass Core software. Replace the policy ARN with the ARN from the previous step.

```
aws iam attach-role-policy
--role-name GreengrassV2TokenExchangeRole
--policy-arn arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
```

If the command has no output, it succeeded, and your core device can access artifacts that you upload to this S3 bucket.

3. Upload the Hello World Python script artifact to the S3 bucket. Run the following command to upload the script to the same path in the bucket where the script exists on your AWS IoT Greengrass core. Replace `DOC-EXAMPLE-BUCKET` with the name of the S3 bucket.

```
aws s3 cp
artifacts/com.example.HelloWorld/1.0.0/hello_world.py
s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/hello_world.py
```

The command outputs a line that starts with `upload:` if the request succeeds.

4. Add the artifact's Amazon S3 URI to the component recipe. The Amazon S3 URI is composed of the bucket name and the path to the artifact object in the bucket. Your script artifact's Amazon S3 URI is the URI that you upload the artifact to in the previous step. This URI should look similar to the following example. Replace `DOC-EXAMPLE-BUCKET` with the name of the S3 bucket.

```
s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/hello_world.py
```

To add the artifact to the recipe, add a list of `Artifacts` that contains a structure with the Amazon S3 URI.

```json
"Artifacts": [
{
"Version": "2012-10-17",
"Statement": [
{
"Effect": "Allow",
"Action": ["s3:GetObject"],
"Resource": "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*"
}
]
}
```
Upload your component

"URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/hello_world.py"
]

Run the following command to open the recipe file.

nano recipes/com.example.HelloWorld-1.0.0.json

Add the artifact to the recipe. Your recipe file should look similar to the following example.

```json
{
   "RecipeFormatVersion": "2020-01-25",
   "ComponentName": "com.example.HelloWorld",
   "ComponentVersion": "1.0.0",
   "ComponentDescription": "My first Greengrass component.",
   "ComponentPublisher": "Amazon",
   "ComponentConfiguration": {
      "DefaultConfiguration": {
         "Message": "world"
      }
   },
   "Manifests": [
      {
         "Platform": {
            "os": "linux"
         },
         "Lifecycle": {
            "Run": "python3 -u {artifacts:path}/hello_world.py '{configuration:/Message}'"
         },
         "Artifacts": [
            {
               "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/hello_world.py"
            }
         ]
      }
   ]
}
```

YAML

Artifacts:
- URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/hello_world.py

Run the following command to open the recipe file.

nano recipes/com.example.HelloWorld-1.0.0.yaml

Add the artifact to the recipe. Your recipe file should look similar to the following example.

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: My first AWS IoT Greengrass component.
ComponentPublisher: Amazon
```
5. Create a component resource in AWS IoT Greengrass from the recipe. Run the following command to create the component from the recipe, which you provide as a binary file.

**JSON**

```bash
aws greengrassv2 create-component-version \
--inline-recipe fileb://recipes/com.example.HelloWorld-1.0.0.json
```

**YAML**

```bash
aws greengrassv2 create-component-version \
--inline-recipe fileb://recipes/com.example.HelloWorld-1.0.0.yaml
```

The response looks similar to the following example if the request succeeds.

```json
{
    "componentName": "com.example.HelloWorld",
    "componentVersion": "1.0.0",
    "creationTimestamp": "Mon Nov 30 09:04:05 UTC 2020",
    "status": {
        "componentState": "REQUESTED",
        "message": "NONE",
        "errors": {}
    }
}
```

Copy the **arn** from the output to check the state of the component in the next step.

**Note**

You can also see your Hello World component in the **AWS IoT Greengrass console** on the **Components** page.

6. Verify that the component creates and is ready to be deployed. When you create a component, its state is **REQUESTED**. Then, AWS IoT Greengrass validates that the component is deployable. You can run the following command to query the component status and verify that your component is deployable. Replace the **arn** with the ARN from the previous step.

```bash
aws greengrassv2 describe-component \
--arn
```

If the component validates, the response indicates that the component state is **DEPLOYABLE**.

```json
{
```

---

ComponentConfiguration:

  DefaultConfiguration:
    Message: world

Manifests:
- Platform:
  os: linux

Lifecycle:
  Run: |
  python3 -u {artifacts:path}/hello_world.py '{configuration:/Message}'

Artifacts:
- URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/
  hello_world.py
Your Hello World component is now available in AWS IoT Greengrass. You can deploy it back to this Greengrass core device or to other core devices.

Deploy your component

With AWS IoT Greengrass, you can deploy components to individual devices or groups of devices. When you deploy a component, AWS IoT Greengrass installs and runs that component's software on each target device. You specify which components to deploy and the configuration update to deploy for each component. You can also control how the deployment rolls out to the devices that the deployment targets. For more information, see Deploy AWS IoT Greengrass components to devices (p. 383).

In this section, you deploy your Hello World component back to your AWS IoT Greengrass core device.

To deploy your Hello World component

1. On your development computer, create a file called hello-world-deployment.json and copy the following JSON into the file. This file defines the components and configurations to deploy.

```json
{
  "components": {
    "com.example.HelloWorld": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "merge": "{"Message":"universe"}
      }
    }
  }
}
```

This configuration file specifies to deploy version 1.0.0 of the Hello World component that you developed and published in the previous procedure. The configurationUpdate specifies to merge the component configuration in a JSON-encoded string. This configuration update sets the Hello World Message parameter to universe for the device in this deployment.

2. Run the following command to deploy the component to your Greengrass core device. You can deploy to things, which are individual devices, or thing groups, which are groups of devices. Replace MyGreengrassCore with the name of the AWS IoT thing for your core device.

```
aws greengrass deploy-component
  --deployment-file hello-world-deployment.json
  --thing-group MyGreengrassCore
```

This command deploys the hello-world-deployment.json file to the MyGreengrassCore thing group. The command installs and runs the component on each device in the group, setting the Hello World Message parameter to universe.
Deploy your component

```bash
aws greengrassv2 create-deployment \
  --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" \
  --cli-input-json file://hello-world-deployment.json
```

The command outputs a response similar to the following example.

```json
{
    "deploymentId": "deb69c37-314a-4369-a6a1-3dff9f6ce73a9",
    "iotJobId": "b5d92151-6348-4941-8603-bdbfb3e02b75",
    "iotJobArn": "arn:aws:iot:region:account-id:job/b5d92151-6348-4941-8603-bdbfb3e02b75"
}
```

3. Verify that the deployment completes successfully. The deployment can take several minutes to complete. Check the Hello World log to verify the change. Run the following command on your Greengrass core device.

```
tail -f /tmp/Greengrass_HelloWorld.log
```

You should see messages similar to the following example.

```
```

**Note**

If the log messages don't change, the deployment failed or didn't reach the core device. This can occur if your core device isn't connected to the internet or doesn't have permissions to retrieve artifacts from your S3 bucket. Run the following command on your core device to view the AWS IoT Greengrass Core software log file. This file includes logs from the Greengrass core device's deployment service.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

For more information, see Troubleshooting AWS IoT Greengrass V2 (p. 751).

You've completed this tutorial. The AWS IoT Greengrass Core software and your Hello World component run on your device. Also, your Hello World component is available in AWS IoT Greengrass to deploy to other devices. For more information about the topics that this tutorial explores, see the following:

- Create local AWS IoT Greengrass components (p. 324)
- Upload components to deploy to your core devices (p. 327)
- Deploy AWS IoT Greengrass components to devices (p. 383)
Setting up AWS IoT Greengrass core devices

Complete the tasks in this section to install, configure, and run the AWS IoT Greengrass Core software V2.

Note
This section describes advanced configuration of the AWS IoT Greengrass Core software. If you're a first-time user of AWS IoT Greengrass V2, we recommend that you complete the getting started tutorial (p. 24) to set up a core device and explore the features of AWS IoT Greengrass.

Topics
- Supported platforms (p. 41)
- Requirements (p. 42)
- Set up an AWS account (p. 44)
- Install the AWS IoT Greengrass Core software (p. 45)
- Run the AWS IoT Greengrass Core software (p. 91)
- Run AWS IoT Greengrass Core software in a Docker container (p. 93)
- Configure the AWS IoT Greengrass Core software (p. 119)
- Update the AWS IoT Greengrass Core software (OTA) (p. 129)
- Uninstall the AWS IoT Greengrass Core software (p. 131)

Supported platforms

AWS IoT Greengrass supports devices running the following platforms:

- Architecture: Armv7l
  - OS: Linux
- Architecture: Armv8 (AArch64)
  - OS: Linux
- Architecture: x86_64
  - OS: Linux

Linux platforms can also run AWS IoT Greengrass V2 in a Docker container. For more information, see Run AWS IoT Greengrass Core software in a Docker container (p. 93).

To build a custom Linux-based operating system, you can use the BitBake recipe for AWS IoT Greengrass V2 in the meta-aws project. The Yocto Project is an open source collaboration project that helps you build custom Linux-based systems for embedded applications regardless of hardware architecture. The meta-aws project provides recipes that you can use to build AWS edge software capabilities in embedded Linux systems that are built with OpenEmbedded and Yocto Project build frameworks. The
BitBake recipe for AWS IoT Greengrass V2 installs, configures, and automatically runs the AWS IoT Greengrass Core software on your device.

## Requirements

Devices must meet the following requirements to install and run the AWS IoT Greengrass Core software v2.x.

**Note**
You can use AWS IoT Device Tester for AWS IoT Greengrass to verify that your device can run the AWS IoT Greengrass Core software and communicate with the AWS Cloud. For more information, see Using AWS IoT Device Tester for AWS IoT Greengrass V2 (p. 660).

- Minimum 256 MB disk space available for the AWS IoT Greengrass Core software. This requirement doesn't include components deployed to the core device.
- Minimum 96 MB RAM allocated to the AWS IoT Greengrass Core software. This requirement doesn't include components that run on the core device. For more information, see Control memory allocation with JVM options (p. 121).
- Java version 8 or greater. We recommend Amazon Corretto 11 or OpenJDK 11.
- GNU C Library (glibc) version 2.25 or greater.
- The user that runs the AWS IoT Greengrass Core software (typically `root`), must have permission to run `sudo` with any user and any group. The `/etc/subrers` file must give this user permission to run `sudo` as other groups. The permission for the user in `/etc/subrers` should look like the following example.

```
root ALL=(ALL:ALL) ALL
```

- The `/tmp` directory must be mounted with `exec` permissions.
- All of the following shell commands:
  - `ps -ax -o pid,puid`
  - `sudo`
  - `sh`
  - `kill`
  - `cp`
  - `chmod`
  - `rm`
  - `ln`
  - `echo`
  - `exit`
  - `id`
  - `uname`
  - `grep`

Your device may also require the following optional shell commands:
- (Optional) `systemctl` (to set up the AWS IoT Greengrass Core software as a system service)
- (Optional) `useradd`, `groupadd`, and `usermod` (to set up the `ggc_user` system user and `ggc_group` system group)
- (Optional) `mkfifo` (to run Lambda functions as components)
- To configure system resource limits for component processes, your device must run Linux kernel version 2.6.24 or later.
To run Lambda functions, your device must meet additional requirements. For more information, see Requirements to run Lambda functions (p. 43).

Requirements to run Lambda functions

Your device must meet the following requirements to run Lambda functions:

- You must run the AWS IoT Greengrass Core software as a root user. Use `sudo`, for example.
- Your device must have the `mkfifo` shell command.
- Your device must run the programming language libraries that a Lambda function requires. You must install the required libraries on the device and add them to the `PATH` environment variable.
  - Python version 3.8 for functions that use the Python 3.8 runtime.
  - Python version 3.7 for functions that use the Python 3.7 runtime.
  - Python version 2.7 for functions that use the Python 2.7 runtime.
  - Node.js version 12.x for functions that use the Node.js 12.x runtime.
  - Node.js version 10.x for functions that use the Node.js 10.x runtime.
  - Java version 8 or later for functions that use the Java 8 runtime.

For more information about AWS IoT Greengrass support for Lambda runtimes, see Run AWS Lambda functions (p. 361).

- To run containerized Lambda functions, your device must meet the following requirements:
  - Linux kernel version 4.4 or later.
  - The kernel must support cgroups, and you must enable and mount the following cgroups:
    - The `memory` cgroup for AWS IoT Greengrass to set the memory limit for containerized Lambda functions.
    - The `devices` cgroup for containerized Lambda functions to access system devices or volumes.
  - You must enable the following Linux kernel configurations on the device:
    - Namespace:
      - `CONFIG_IPC_NS`
      - `CONFIG_UTS_NS`
      - `CONFIG_USER_NS`
      - `CONFIG_PID_NS`
    - Cgroups:
      - `CONFIG_CGROUP_DEVICE`
      - `CONFIG_CGROUPS`
      - `CONFIG_MEMCG`
    - Others:
      - `CONFIG_POSIX_MQUEUE`
      - `CONFIG_OVERLAY_FS`
      - `CONFIG_HAVE_ARCH_SECCOMP_FILTER`
      - `CONFIG_SECCOMP_FILTER`
      - `CONFIG_KEYS`
      - `CONFIG_SECCOMP`
      - `CONFIG_SHMEM`
Set up an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account
2. Follow the online instructions.
   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

To create an administrator user for yourself and add the user to an administrators group (console)
1. Sign in to the IAM console as the account owner by choosing Root user and entering your AWS account email address. On the next page, enter your password.
   Note
   We strongly recommend that you adhere to the best practice of using the Administrator IAM user that follows and securely lock away the root user credentials. Sign in as the root user only to perform a few account and service management tasks.
2. In the navigation pane, choose Users and then choose Add user.
3. For User name, enter Administrator.
4. Select the check box next to AWS Management Console access. Then select Custom password, and then enter your new password in the text box.
5. (Optional) By default, AWS requires the new user to create a new password when first signing in. You can clear the check box next to User must create a new password at next sign-in to allow the new user to reset their password after they sign in.
6. Choose Next: Permissions.
7. Under Set permissions, choose Add user to group.
8. Choose Create group.
9. In the Create group dialog box, for Group name enter Administrators.
10. Choose Filter policies, and then select AWS managed - job function to filter the table contents.
11. In the policy list, select the check box for AdministratorAccess. Then choose Create group.
   Note
   You must activate IAM user and role access to Billing before you can use the AdministratorAccess permissions to access the AWS Billing and Cost Management console. To do this, follow the instructions in step 1 of the tutorial about delegating access to the billing console.
12. Back in the list of groups, select the check box for your new group. Choose Refresh if necessary to see the group in the list.
13. Choose Next: Tags.
14. (Optional) Add metadata to the user by attaching tags as key-value pairs. For more information about using tags in IAM, see Tagging IAM entities in the IAM User Guide.
15. Choose Next: Review to see the list of group memberships to be added to the new user. When you are ready to proceed, choose Create user.

You can use this same process to create more groups and users and to give your users access to your AWS account resources. To learn about using policies that restrict user permissions to specific AWS resources, see Access management and Example policies.
Install the AWS IoT Greengrass Core software

AWS IoT Greengrass extends AWS to edge devices so that they can act on the data they generate, while they use the AWS Cloud for management, analytics, and durable storage. Install the AWS IoT Greengrass Core software on edge devices to integrate with AWS IoT Greengrass and the AWS Cloud.

**Important**
Before you download the AWS IoT Greengrass Core software, check that your core device meets the requirements (p. 42) to install and run the AWS IoT Greengrass Core software v2.0.

The AWS IoT Greengrass Core software includes an installer that sets up your device as a Greengrass core device. When you run the installer, you can configure options, such as the root folder and the AWS Region to use. You can choose to have the installer create required AWS IoT and IAM resources for you. You can also choose to deploy local development tools to configure a device that you use for custom component development.

The AWS IoT Greengrass Core software requires the following AWS IoT and IAM resources to connect to the AWS Cloud and operate:

- An AWS IoT thing. When you register a device as an AWS IoT thing, that device can use a digital certificate to authenticate with AWS. This certificate allows the device to communicate with AWS IoT and AWS IoT Greengrass. For more information, see Device authentication and authorization for AWS IoT Greengrass (p. 610).
- (Optional) An AWS IoT thing group. You use thing groups to manage fleets of Greengrass core devices. When you deploy software components to your devices, you can choose to deploy to individual devices or to groups of devices. You can add a device to a thing group to deploy that thing group's software components to the device. For more information, see Deploy AWS IoT Greengrass components to devices (p. 383).
- An IAM role. Greengrass core devices use the AWS IoT Core credentials provider to authorize calls to AWS services with an IAM role. This role allows your device to interact with AWS IoT, send logs to Amazon CloudWatch Logs, and download custom component artifacts from Amazon Simple Storage Service (Amazon S3). For more information, see Authorize core devices to interact with AWS services (p. 627).
- An AWS IoT role alias. Greengrass core devices use the role alias to identify the IAM role to use. The role alias enables you to change the IAM role but keep the device configuration the same. For more information, see Authorizing direct calls to AWS services in the AWS IoT Core Developer Guide.

Choose one of the following options to install the AWS IoT Greengrass Core software on your core device.

- **Quick installation**

  Choose this option to set up a Greengrass core device in as few steps as possible. The installer creates the required AWS IoT and IAM resources for you. This option requires you to provide AWS credentials to the installer to create resources in your AWS account.

  You can't use this option to install behind a firewall or network proxy. If your devices are behind a firewall or network proxy, consider manual installation (p. 50).

  For more information, see Install AWS IoT Greengrass Core software with automatic resource provisioning (p. 46).

- **Manual installation**

  Choose this option to create the required AWS resources manually or to install behind a firewall or network proxy. By using a manual installation, you don't need to give the installer permission to create resources in your AWS account, because you create the required AWS IoT and IAM resources. You can also configure your device to connect on port 443 or through a network proxy.
Install with automatic provisioning

For more information, see Install AWS IoT Greengrass Core software with manual resource provisioning (p. 50).

• **Installation with AWS IoT fleet provisioning**

Choose this option to create the required AWS resources from an AWS IoT fleet provisioning template. You might choose this option to create similar devices in a fleet, or if you manufacture devices that your customers later activate, such as vehicles or smart home devices. Devices use claim certificates to authenticate and provision AWS resources, including an X.509 client certificate that the device uses to connect to the AWS Cloud for normal operation. You can embed or flash the claim certificates into the device's hardware during manufacturing, and you can use the same claim certificate and key to provision multiple devices. You can also configure devices to connect on port 443 or through a network proxy.

For more information, see Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning (p. 62).

• **Installation with custom provisioning**

Choose this option to develop a custom Java application that provisions the required AWS resources. You might choose this option if you create your own X.509 client certificates or if you want more control over the provisioning process. AWS IoT Greengrass provides an interface that you can implement to exchange information between your custom provisioning application and the AWS IoT Greengrass Core software installer.

For more information, see Install AWS IoT Greengrass Core software with custom resource provisioning (p. 83).

AWS IoT Greengrass also provides containerized environments that run the AWS IoT Greengrass Core software. You can use a Dockerfile to run AWS IoT Greengrass in a Docker container (p. 93).

**Topics**

- Install AWS IoT Greengrass Core software with automatic resource provisioning (p. 46)
- Install AWS IoT Greengrass Core software with manual resource provisioning (p. 50)
- Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning (p. 62)
- Install AWS IoT Greengrass Core software with custom resource provisioning (p. 83)
- Installer arguments (p. 89)

**Install AWS IoT Greengrass Core software with automatic resource provisioning**

The AWS IoT Greengrass Core software includes an installer that sets up your device as a Greengrass core device. To set up a device quickly, the installer can provision the AWS IoT thing, AWS IoT thing group, IAM role, and AWS IoT role alias that the core device requires to operate. The installer can also deploy the local development tools to the core device, so you can use the device to develop and test custom software components. The installer requires AWS credentials to provision these resources and create the deployment.

If you can't provide AWS credentials to the device, you can provision the AWS resources that the core device requires to operate. You can also deploy the development tools to a core device to use as a development device. This enables you to provide fewer permissions to the device when you run the installer. For more information, see Install AWS IoT Greengrass Core software with manual resource provisioning (p. 50).
Important
Before you download the AWS IoT Greengrass Core software, check that your core device meets the requirements (p. 42) to install and run the AWS IoT Greengrass Core software v2.0.

Topics
- Provide AWS credentials to the device (p. 47)
- Download the AWS IoT Greengrass Core software (p. 47)
- Install the AWS IoT Greengrass Core software (p. 48)

Provide AWS credentials to the device

Provide your AWS credentials to your device so that the installer can provision the required AWS resources. For more information about the required permissions, see Minimal IAM policy for installer to provision resources (p. 629).

Note
The installer doesn't save or store your credentials.

To provide AWS credentials to the device

- On your device, provide AWS credentials by doing one of the following:
  - Use long-term credentials from an IAM user:
    a. Provide the access key ID and secret access key for your IAM user. For more information about how to retrieve long-term credentials, see Managing access keys for IAM users in the IAM User Guide.
    b. Run the following commands to provide the credentials to the AWS IoT Greengrass Core software.

      ```
      export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
      export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
      ```
  - (Recommended) Use temporary security credentials from an IAM role:
    a. Provide the access key ID, secret access key, and session token from an IAM role that you assume. For more information about how to retrieve these credentials, see Using temporary security credentials with the AWS CLI in the IAM User Guide.
    b. Run the following commands to provide the credentials to the AWS IoT Greengrass Core software.

      ```
      export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
      export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfjCYEXAMPLEKEY
      export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o50ytwEXAMPLE=
      ```

Download the AWS IoT Greengrass Core software

You can download the latest version of the AWS IoT Greengrass Core software from the following location:

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

Note
You can download a specific version of the AWS IoT Greengrass Core software from the following location. Replace version with the version to download.
To download the AWS IoT Greengrass Core software (Linux)

1. On your device, download the AWS IoT Greengrass Core software to a file named
   `greengrass-nucleus-latest.zip`.

   ```bash
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip > greengrass-nucleus-latest.zip
   ```

   By downloading this software, you agree to the Greengrass Core Software License Agreement.

2. Unzip the AWS IoT Greengrass Core software to a folder on your device. Replace
   `GreengrassInstaller` with the folder that you want to use.

   ```bash
   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-nucleus-latest.zip
   ```

   **Important**
   If you install a version of the Greengrass nucleus earlier than v2.4.0, don't remove this
   folder after you install the AWS IoT Greengrass Core software. The AWS IoT Greengrass
   Core software uses the files in this folder to run.
   If you downloaded the latest version of the software, you install v2.4.0 or later, and you can
   remove this folder after you install the AWS IoT Greengrass Core software.

3. (Optional) Run the following command to see the version of the AWS IoT Greengrass Core software.

   ```bash
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

Install the AWS IoT Greengrass Core software

Run the installer with arguments that specify to do the following:

- Create the AWS resources that the core device requires to operate.
- Use the `ggc_user` system user and `ggc_group` system group to run software components on the
  core device. The installer creates this default user and group if they don't exist.
- Install the software as a system service that runs on boot, if your device has the `systemd` init system.

To set up a development device with local development tools, specify the `--deploy-dev-tools` `true` argument. The local development tools can take up to a minute to deploy after the installation completes.

For more information about the arguments that you can specify, see Installer arguments (p. 89).

**Note**
If you are running AWS IoT Greengrass on a device with limited memory, you can control the
amount of memory that AWS IoT Greengrass Core software uses. To control memory allocation,
you can set JVM heap size options in the `jvmOptions` configuration parameter in your nucleus
component. For more information, see Control memory allocation with JVM options (p. 121).

To install the AWS IoT Greengrass Core software (Linux)

1. Run the AWS IoT Greengrass Core installer. Replace argument values in your command as follows.

   a. `/greengrass/v2`: The path to the root folder to use to install the AWS IoT Greengrass Core
      software.
b. **GreengrassInstaller**. The path to the folder where you unpacked the AWS IoT Greengrass Core software installer.

c. **region**. The AWS Region in which to find or create resources.

d. **MyGreengrassCore**. The name of the AWS IoT thing for your Greengrass core device. If the thing doesn't exist, the installer creates it. The installer downloads the certificates to authenticate as the AWS IoT thing. For more information, see Device authentication and authorization for AWS IoT Greengrass (p. 610).

   **Note**
   The thing name can't contain colon (:) characters.

e. **MyGreengrassCoreGroup**. The name of AWS IoT thing group for your Greengrass core device. If the thing group doesn't exist, the installer creates it and adds the thing to it. If the thing group exists and has an active deployment, the core device downloads and runs the software that the deployment specifies.

   **Note**
   The thing group name can't contain colon (:) characters.

f. **GreengrassV2IoTThingPolicy**. The name of the AWS IoT policy that allows the Greengrass core devices to communicate with AWS IoT and AWS IoT Greengrass. If the AWS IoT policy doesn't exist, the installer creates a permissive AWS IoT policy with this name. You can restrict this policy's permissions for you use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices (p. 614).

g. **GreengrassV2TokenExchangeRole**. The name of the IAM role that allows the Greengrass core device to get temporary AWS credentials. If the role doesn't exist, the installer creates it and creates and attaches a policy named **GreengrassV2TokenExchangeRoleAccess**. For more information, see Authorize core devices to interact with AWS services (p. 627).

h. **GreengrassCoreTokenExchangeRoleAlias**. The alias to the IAM role that allows the Greengrass core device to get temporary credentials later. If the role alias doesn't exist, the installer creates it and points it to the IAM role that you specify. For more information, see Authorize core devices to interact with AWS services (p. 627).

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
  -jar ./GreengrassInstaller/lib/Greengrass.jar \
  --aws-region region \
  --thing-name MyGreengrassCore \
  --thing-group-name MyGreengrassCoreGroup \
  --thing-policy-name GreengrassV2IoTThingPolicy \
  --tes-role-name GreengrassV2TokenExchangeRole \
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias \
  --component-default-user ggc_user:ggc_group \
  --provision true \
  --setup-system-service true
```

The installer prints the following messages if it succeeds:

- If you specify `--provision`, the installer prints **Successfully configured Nucleus with provisioned resource details** if it configured the resources successfully.
- If you specify `--deploy-dev-tools`, the installer prints **Configured Nucleus to deploy aws.greengrass.Cli component** if it created the deployment successfully.
- If you specify `--setup-system-service true`, the installer prints **Successfully set up Nucleus as a system service** if it set up and ran the software as a service.
- If you don't specify `--setup-system-service true`, the installer prints **Launched Nucleus successfully** if it succeeded and ran the software.

2. Skip this step if you installed Greengrass nucleus (p. 136) v2.0.4 or later. If you downloaded the latest version of the software, you installed v2.0.4 or later.
Run the following command to set the required file permissions for your AWS IoT Greengrass Core software root folder. Replace `/greengrass/v2` with the root folder that you specified in your installation command, and replace `/greengrass` with the parent folder for your root folder.

```
sudo chmod 755 /greengrass/v2 && sudo chmod 755 /greengrass
```

If you installed the AWS IoT Greengrass Core software as a system service, the installer runs the software for you. Otherwise, you must run the software manually. For more information, see Run the AWS IoT Greengrass Core software (p. 91).

**Note**
By default, the IAM role that the installer creates doesn’t allow access to component artifacts in S3 buckets. To deploy custom components that define artifacts in Amazon S3, you must add permissions to the role to allow your core device to retrieve component artifacts. For more information, see Allow access to S3 buckets for component artifacts (p. 628).

If you don’t yet have an S3 bucket for component artifacts, you can add these permissions later after you create a bucket.

For more information about how to configure and use the software and AWS IoT Greengrass, see the following:

- Configure the AWS IoT Greengrass Core software (p. 119)
- Develop AWS IoT Greengrass components (p. 322)
- Deploy AWS IoT Greengrass components to devices (p. 383)
- Greengrass Command Line Interface (p. 595)

## Install AWS IoT Greengrass Core software with manual resource provisioning

The AWS IoT Greengrass Core software includes an installer that sets up your device as a Greengrass core device. To set up a device manually, you can create the required AWS IoT and IAM resources for the device to use. If you create these resources manually, you don’t need to provide AWS credentials to the installer.

When you manually install the AWS IoT Greengrass Core software, you can also configure the device to use a network proxy or connect to AWS on port 443. You might need to specify these configuration options if your device runs behind a firewall or a network proxy, for example. For more information, see Connect on port 443 or through a network proxy (p. 125).

**Important**
Before you download the AWS IoT Greengrass Core software, check that your core device meets the requirements (p. 42) to install and run the AWS IoT Greengrass Core software v2.0.

**Topics**

- Create an AWS IoT thing (p. 51)
- Retrieve AWS IoT endpoints (p. 54)
- Create a token exchange role (p. 55)
- Download certificates to the device (p. 58)
- Download the AWS IoT Greengrass Core software (p. 47)
- Install the AWS IoT Greengrass Core software (p. 60)
Create an AWS IoT thing

AWS IoT things represent devices and logical entities that connect to AWS IoT. Greengrass core devices are AWS IoT things. When you register a device as an AWS IoT thing, that device can use a digital certificate to authenticate with AWS. This certificate allows the device to communicate with AWS IoT and AWS IoT Greengrass.

In this section, you create an AWS IoT thing and download certificates that your device can use to connect to AWS.

To create an AWS IoT thing

1. Create an AWS IoT thing for your device. On your development computer, run the following command.

   - Replace `MyGreengrassCore` with the thing name to use. This name is also the name of your Greengrass core device.

   ```
   aws iot create-thing --thing-name MyGreengrassCore
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
   "thingName": "MyGreengrassCore",
   "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
   "thingId": "8cb4b6cd-268e-4959-b5b9-1713d73bf42"
   }
   ```

2. Create a folder where you download the certificates for the AWS IoT thing.

   ```
   mkdir greengrass-v2-certs
   ```

3. Create and download the certificates for the AWS IoT thing.

   ```
   aws iot create-keys-and-certificate --set-as-active --certificate-pem-outfile greengrass-v2-certs/device.pem.crt --public-key-outfile greengrass-v2-certs/public.pem.key --private-key-outfile greengrass-v2-certs/private.pem.key
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
   "certificateArn": "arn:aws:iot:us-west-2:123456789012:cert/aa0bb795877878bae251d8a7d9d547f4889c524c9b574ab9fb65f32248b1d4",
   "certificateId": "aa0bb795877878bae251d8a7d9d547f4889c524c9b574ab9fb65f32248b1d4",
   "certificatePem": "-----BEGIN CERTIFICATE-----
   BAQfQfDcBdELMgkAgAUEhBhMCVVMxM5zAJBGVBA9GgkP251d8a7d9d547f4889c524c9b574ab9fb65f32248b1d4"
   }
   ```
Install with manual provisioning

4. Attach the certificate to the AWS IoT thing.
   - Replace `MyGreengrassCore` with the name of your AWS IoT thing.
   - Replace the certificate Amazon Resource Name (ARN) with the ARN of the certificate that you created in the previous step.

   ```
   aws iot attach-thing-principal --thing-name MyGreengrassCore --principal arn:aws:iot:us-west-2:123456789012:cert/aa0b7958770878eabe251d8a7dd547f4889c524c9b574ab5fbf65f32248b1d4
   ``

   The command doesn't have any output if the request succeeds.

5. Create and attach an AWS IoT policy that defines the AWS IoT permissions for your Greengrass core device. The following policy allows access to all MQTT topics and Greengrass operations, so your device works with custom applications and future changes that require new Greengrass operations. You can restrict this policy down based on your use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices (p. 614).

   If you have set up a Greengrass core device before, you can attach its AWS IoT policy instead of creating a new one.

   Do the following:
   a. Create a file that contains the AWS IoT policy document that Greengrass core devices require.

      ```
      nano greengrass-v2-lot-policy.json
      ```

      Copy the following JSON into the file.

      ```
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Action": [
              "iot:Publish",
              "iot:Subscribe",
              "iot:Receive",
              "iot:Connect",
              "greengrass:*"
            ],
          }
        ]
      }
      ```
b. Create an AWS IoT policy from the policy document.

- Replace `GreengrassV2IoTThingPolicy` with the name of the policy to create.

```
aws iot create-policy --policy-name GreengrassV2IoTThingPolicy --policy-document file://greengrass-v2-iot-policy.json
```

The response looks similar to the following example, if the request succeeds.

```
{
  "policyName": "GreengrassV2IoTThingPolicy",
  "policyDocument": "{
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Resource": ["*"]
      }
    ]
  },
  "ResourceArn": ["*"]
},
"policyVersionId": "1"
}
```

c. Attach the AWS IoT policy to the AWS IoT thing's certificate.

- Replace `GreengrassV2IoTThingPolicy` with the name of the policy to attach.
- Replace the target ARN with the ARN of the certificate for your AWS IoT thing.

```
aws iot attach-policy --policy-name GreengrassV2IoTThingPolicy --target arn:aws:iot:us-west-2:123456789012:cert/aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9f6b65f32248b1d4
```

The command doesn't have any output if the request succeeds.

6. (Optional) Add the AWS IoT thing to a new or existing thing group. You use thing groups to manage fleets of Greengrass core devices. When you deploy software components to your devices, you can choose to target individual devices or groups of devices. You can add a device to a thing group with an active Greengrass deployment to deploy that thing group's software components to the device. Do the following:

a. (Optional) Create an AWS IoT thing group.
• Replace `MyGreengrassCoreGroup` with the name of the thing group to create.

  **Note**  
  The thing group name can’t contain colon (:) characters.

```
aws iot create-thing-group --thing-group-name MyGreengrassCoreGroup
```

The response looks similar to the following example, if the request succeeds.

```
{
  "thingGroupName": "MyGreengrassCoreGroup",
  "thingGroupId": "4df721e1-ff9f-4f97-92dd-02db4e3f03aa"
}
```

b. Add the AWS IoT thing to a thing group.

• Replace `MyGreengrassCore` with the name of your AWS IoT thing.

• Replace `MyGreengrassCoreGroup` with the name of the thing group.

```
aws iot add-thing-to-thing-group --thing-name MyGreengrassCore --thing-group-name MyGreengrassCoreGroup
```

The command doesn't have any output if the request succeeds.

## Retrieve AWS IoT endpoints

Get the AWS IoT endpoints for your AWS account, and save them to use later. Your device uses these endpoints to connect to AWS IoT. Do the following:

1. Get the AWS IoT data endpoint for your AWS account.

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

The response looks similar to the following example, if the request succeeds.

```
{
  "endpointAddress": "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
}
```

2. Get the AWS IoT credentials endpoint for your AWS account.

```
aws iot describe-endpoint --endpoint-type iot:CredentialProvider
```

The response looks similar to the following example, if the request succeeds.

```
{
  "endpointAddress": "device-credentials-prefix.credentials.iot.us-west-2.amazonaws.com"
}
```
Create a token exchange role

Greengrass core devices use an IAM service role, called the token exchange role, to authorize calls to AWS services. The device uses the AWS IoT credentials provider to get temporary AWS credentials for this role, which allows the device to interact with AWS IoT, send logs to Amazon CloudWatch Logs, and download custom component artifacts from Amazon S3. For more information, see Authorize core devices to interact with AWS services (p. 627).

You use an AWS IoT role alias to configure the token exchange role for Greengrass core devices. Role aliases enable you to change the token exchange role for a device but keep the device configuration the same. For more information, see Authorizing direct calls to AWS services in the AWS IoT Core Developer Guide.

In this section, you create a token exchange IAM role and an AWS IoT role alias that points to the role. If you have already set up a Greengrass core device, you can use its token exchange role and role alias instead of creating new ones. Then, you configure your device's AWS IoT thing to use that role and alias.

**To create a token exchange IAM role**

1. Create an IAM role that your device can use as a token exchange role. Do the following:
   a. Create a file that contains the trust policy document that the token exchange role requires.
      
      ```
      nano device-role-trust-policy.json
      ```

      Copy the following JSON into the file.

      ```json
      {
      "Version": "2012-10-17",
      "Statement": [
      {
      "Effect": "Allow",
      "Principal": {
      "Service": "credentials.iot.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
      }
      ]
      }
      ```
   b. Create the token exchange role with the trust policy document.
      - Replace `GreengrassV2TokenExchangeRole` with the name of the IAM role to create.

      ```
      aws iam create-role --role-name GreengrassV2TokenExchangeRole --assume-role-policy-document file://device-role-trust-policy.json
      ```

      The response looks similar to the following example, if the request succeeds.

      ```json
      {
      "Role": {
      "Path": "/",
      "RoleName": "GreengrassV2TokenExchangeRole",
      "RoleId": "AROAZ2YMUHYHK5OKM77FB",
      "Arn": "arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole",
      "CreateDate": "2021-02-06T00:13:29+00:00",
      "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      ```
c. Create a file that contains the access policy document that the token exchange role requires.

nano device-role-access-policy.json

Copy the following JSON into the file.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:DescribeCertificate",
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
}
```

**Note**
This access policy doesn't allow access to component artifacts in S3 buckets. To deploy custom components that define artifacts in Amazon S3, you must add permissions to the role to allow your core device to retrieve component artifacts. For more information, see Allow access to S3 buckets for component artifacts (p. 628). If you don't yet have an S3 bucket for component artifacts, you can add these permissions later after you create a bucket.

d. Create the IAM policy from the policy document.

- Replace `GreengrassV2TokenExchangeRoleAccess` with the name of the IAM policy to create.

```bash
aws iam create-policy --policy-name GreengrassV2TokenExchangeRoleAccess --policy-document file://device-role-access-policy.json
```

The response looks similar to the following example, if the request succeeds.

```json
{
  "Policy": {
    "PolicyName": "GreengrassV2TokenExchangeRoleAccess",
    "PolicyId": "ANPAZ2YMUHYHACI7C5Z66",
```
e. Attach the IAM policy to the token exchange role.

- Replace `GreengrassV2TokenExchangeRole` with the name of the IAM role.
- Replace the policy ARN with the ARN of the IAM policy that you created in the previous step.

```bash
```

The command doesn't have any output if the request succeeds.

2. Create an AWS IoT role alias that points to the token exchange role.

- Replace `GreengrassCoreTokenExchangeRoleAlias` with the name of the role alias to create.
- Replace the role ARN with the ARN of the IAM role that you created in the previous step.

```bash
aws iot create-role-alias --role-alias GreengrassCoreTokenExchangeRoleAlias --role-arn arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole
```

The response looks similar to the following example, if the request succeeds.

```json
{
  "roleAlias": "GreengrassCoreTokenExchangeRoleAlias",
  "roleAliasArn": "arn:aws:iot:us-west-2:123456789012:rolealias/GreengrassCoreTokenExchangeRoleAlias"
}
```

**Note**

To create a role alias, you must have permission to pass the token exchange IAM role to AWS IoT. If you receive an error message when you try to create a role alias, check that your AWS user has this permission. For more information, see [Granting a user permissions to pass a role to an AWS service](https://docs.aws.amazon.com/IAM/latest/UserGuide/idRoleId.html) in the `AWS Identity and Access Management User Guide`.

3. Create and attach an AWS IoT policy that allows your Greengrass core device to use the role alias to assume the token exchange role. If you have set up a Greengrass core device before, you can attach its role alias AWS IoT policy instead of creating a new one. Do the following:

a. (Optional) Create a file that contains the AWS IoT policy document that the role alias requires.

   ```bash
   nano greengrass-v2-iot-role-alias-policy.json
   ```

   Copy the following JSON into the file.

   - Replace the resource ARN with the ARN of your role alias.

   ```json
   {
   ```
b. Create an AWS IoT policy from the policy document.

- Replace `GreengrassCoreTokenExchangeRoleAliasPolicy` with the name of the AWS IoT policy to create.

```bash
aws iot create-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy --policy-document file://greengrass-v2-iot-role-alias-policy.json
```

The response looks similar to the following example, if the request succeeds.

```json
{
    "policyName": "GreengrassCoreTokenExchangeRoleAliasPolicy",
    "policyDocument": "{
        "Version": "2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Action": "iot:AssumeRoleWithCertificate",
            }
        ]
    }
}
```

| 1 |

---

c. Attach the AWS IoT policy to the AWS IoT thing's certificate.

- Replace `GreengrassCoreTokenExchangeRoleAliasPolicy` with the name of the role alias AWS IoT policy.
- Replace the target ARN with the ARN of the certificate for your AWS IoT thing.

```bash
aws iot attach-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy --target arn:aws:iot:us-west-2:123456789012:cert/aa0b7958770878eabe251d8a7dd547f4889c524c9b574ab9f6f32248b1d4
```

The command doesn't have any output if the request succeeds.

---

**Download certificates to the device**

Earlier, you downloaded your device's certificates to your development computer. In this section, you copy these certificates to your device set up the device with the certificates that it uses to connect to AWS IoT.
To download certificates to the device

1. Copy the AWS IoT thing certificates from your development computer to the device. You might be able to use the `scp` command, for example.
   - Replace `device-ip-address` with the IP of your device.
   
   ```bash
   scp -r greengrass-v2-certs/ device-ip-address:~
   ```

2. Create the Greengrass root folder on the device. You'll later install the AWS IoT Greengrass Core software to this folder.
   - Replace `/greengrass/v2` with the folder to use.
   
   ```bash
   sudo mkdir -p /greengrass/v2
   ```

3. Set the permissions of the parent of the Greengrass root folder.
   - Replace `/greengrass` with the parent of the root folder.
   
   ```bash
   sudo chmod 755 /greengrass
   ```

4. Copy the AWS IoT thing certificates to the Greengrass root folder.
   - Replace `/greengrass/v2` with the Greengrass root folder.
   
   ```bash
   sudo cp -R ~/greengrass-v2-certs/* /greengrass/v2
   ```

5. Download the Amazon root certificate authority (CA) certificate. AWS IoT certificates are associated with Amazon's root CA certificate by default.
   
   ```bash
   sudo curl -o /greengrass/v2/AmazonRootCA1.pem https://www.amazontrust.com/repository/AmazonRootCA1.pem
   ```

Download the AWS IoT Greengrass Core software

You can download the latest version of the AWS IoT Greengrass Core software from the following location:

- [https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip](https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip)

**Note**

You can download a specific version of the AWS IoT Greengrass Core software from the following location. Replace `version` with the version to download.

- [https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip](https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip)

To download the AWS IoT Greengrass Core software (Linux)

1. On your device, download the AWS IoT Greengrass Core software to a file named `greengrass-nucleus-latest.zip`.  


1. Check the version of the AWS IoT Greengrass Core software.
   - Replace `GreengrassInstaller` with the path to the folder that contains the software.

   ```bash
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

2. Use a text editor to create a configuration file named `config.yaml` to provide to the installer.
   - For example, on a Linux-based system, you can run the following command to use GNU nano to create `config.yaml` in the `GreengrassInstaller` folder.

   ```bash
   nano config.yaml
   ```
nano `GreengrassInstaller/config.yaml`

Copy the following YAML content into the file. This partial configuration file specifies system parameters and Greengrass nucleus parameters.

```yaml
---
system:
certificateFilePath: "/greengrass/v2/device.pem.crt"
privateKeyPath: "/greengrass/v2/private.pem.key"
rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
rootpath: "/greengrass/v2"
thingName: "MyGreengrassCore"
services:
  aws.greengrass.Nucleus:
    componentType: "NUCLEUS"
    version: "2.4.0"
    configuration:
      awsRegion: "us-west-2"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      iotCredEndpoint: "device-credentials-prefix.credentials.iot.us-west-2.amazonaws.com"
---
```

Then, do the following:

- Replace each instance of `/greengrass/v2` with the Greengrass root folder.
- Replace `MyGreengrassCore` with the name of the AWS IoT thing.
- Replace `2.4.0` with the version of the AWS IoT Greengrass Core software.
- Replace `us-west-2` with the AWS Region where you created the resources.
- Replace `GreengrassCoreTokenExchangeRoleAlias` with the name of the token exchange role alias.
- Replace the `iotDataEndpoint` with your AWS IoT data endpoint.
- Replace the `iotCredEndpoint` with your AWS IoT credentials endpoint.

**Note**

In this configuration file, you can customize other nucleus configuration options such as the ports and network proxy to use, as shown in the following example. For more information, see `Greengrass nucleus configuration` (p. 137).
3. Run the installer, and specify --init-config to provide the configuration file.

   • Replace `/greengrass/v2` with the Greengrass root folder.
   • Replace each instance of `GreengrassInstaller` with the folder where you unpacked the installer.

   ```bash
   sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
   -jar ./GreengrassInstaller/lib/Greengrass.jar \
   --init-config ./GreengrassInstaller/config.yaml \
   --component-default-user ggc_user:ggc_group \
   --setup-system-service true
   ```

   If you specify --setup-system-service true, the installer prints Successfully set up Nucleus as a system service if it set up and ran the software as a system service. Otherwise, the installer doesn’t output any message if it installs the software successfully.

   **Note**
   You can’t use the deploy-dev-tools argument to deploy local development tools when you run the installer without the --provision true argument. For information about deploying the Greengrass CLI directly on your device, see Greengrass Command Line Interface (p. 595).

4. Verify the installation by viewing the files in the root folder.

   ```bash
   ls /greengrass/v2
   ```

   If the installation succeeded, the root folder contains several folders, such as config, packages, and logs.

   If you installed the AWS IoT Greengrass Core software as a system service, the installer runs the software for you. Otherwise, you must run the software manually. For more information, see Run the AWS IoT Greengrass Core software (p. 91).

   For more information about how to configure and use the software and AWS IoT Greengrass, see the following:

   • Configure the AWS IoT Greengrass Core software (p. 119)
   • Develop AWS IoT Greengrass components (p. 322)
   • Deploy AWS IoT Greengrass components to devices (p. 383)
   • Greengrass Command Line Interface (p. 595)

**Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning**

This feature is available for v2.4.0 and later of the Greengrass nucleus component (p. 136).
With AWS IoT fleet provisioning, you can configure AWS IoT to generate and securely deliver X.509 device certificates and private keys to your devices when they connect to AWS IoT for the first time. AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA). You can also configure AWS IoT to specify thing groups, thing types, and permissions for Greengrass core devices that you provision with fleet provisioning. You define a provisioning template to define how AWS IoT provisions each device. The provisioning template specifies the thing, policy, and certificate resources to create for a device when provisioning. For more information, see Provisioning templates in the AWS IoT Core Developer Guide.

AWS IoT Greengrass provides an AWS IoT fleet provisioning plugin that you can use to install the AWS IoT Greengrass Core software using AWS resources created by AWS IoT fleet provisioning. The fleet provisioning plugin uses provisioning by claim. Devices use a provisioning claim certificate and private key to obtain a unique X.509 device certificate and private key that they can use for regular operations. You can embed the claim certificate and private key in each device during manufacturing, so your customers can activate devices later when each device comes online. You can use the same claim certificate and private key for multiple devices. For more information, see Provisioning by claim in the AWS IoT Core Developer Guide.

To install the AWS IoT Greengrass Core software with AWS IoT fleet provisioning, you must set up resources in your AWS account that AWS IoT uses to provision Greengrass core devices. These resources include a provisioning template, claim certificates, and a token exchange IAM role (p. 627). After you create these resources, you can reuse them to provision multiple core devices in a fleet. For more information, see Set up AWS IoT fleet provisioning for Greengrass core devices (p. 69).

**Important**
Before you download the AWS IoT Greengrass Core software, check that your core device meets the requirements (p. 42) to install and run the AWS IoT Greengrass Core software v2.0.

**Topics**
- Prerequisites (p. 63)
- Retrieve AWS IoT endpoints (p. 54)
- Download certificates to the device (p. 64)
- Download the AWS IoT Greengrass Core software (p. 47)
- Download the AWS IoT fleet provisioning plugin (p. 66)
- Install the AWS IoT Greengrass Core software (p. 66)
- Set up AWS IoT fleet provisioning for Greengrass core devices (p. 69)
- Configure the AWS IoT fleet provisioning plugin (p. 81)

**Prerequisites**

To install the AWS IoT Greengrass Core software with AWS IoT fleet provisioning, you must first set up AWS IoT fleet provisioning for Greengrass core devices (p. 69). After you complete these steps once, you can use fleet provisioning to install the AWS IoT Greengrass Core software on any number of devices.

**Retrieve AWS IoT endpoints**

Get the AWS IoT endpoints for your AWS account, and save them to use later. Your device uses these endpoints to connect to AWS IoT. Do the following:

1. Get the AWS IoT data endpoint for your AWS account.

   ```
   aws iot describe-endpoint --endpoint-type iot:Data-ATS
   ```

   The response looks similar to the following example, if the request succeeds.
Install with fleet provisioning

```json
{
  "endpointAddress": "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
}
```

2. Get the AWS IoT credentials endpoint for your AWS account.

```bash
aws iot describe-endpoint --endpoint-type iot:CredentialProvider
```

The response looks similar to the following example, if the request succeeds.

```json
{
  "endpointAddress": "device-credentials-prefix.credentials.iot.us-west-2.amazonaws.com"
}
```

Download certificates to the device

The device uses a claim certificate and private key to authenticate its request to provision AWS resources and acquire an X.509 device certificate. You can embed the claim certificate and private key into the device during manufacturing, or copy the certificate and key to the device during installation. In this section, you copy the claim certificate and private key to the device. You also download the Amazon Root certificate authority (CA) certificate to the device.

**Important**

Provisioning claim private keys should be secured at all times, including on Greengrass core devices. We recommend that you use Amazon CloudWatch metrics and logs to monitor for indications of misuse, such as unauthorized use of the claim certificate to provision devices. If you detect misuse, disable the provisioning claim certificate so that it can't be used for device provisioning. For more information, see Monitoring AWS IoT in the AWS IoT Core Developer Guide.

To help you better manage the number of devices, and which devices, that register themselves in your AWS account, you can specify a pre-provisioning hook when you create a fleet provisioning template. A pre-provisioning hook is an AWS Lambda function that validates template parameters that devices provide during registration. For example, you might create a pre-provisioning hook that checks a device ID against a database to verify that the device has permission to provision. For more information, see Pre-provisioning hooks in the AWS IoT Core Developer Guide.

**To download claim certificates to the device**

1. Copy the claim certificate and private key to the device.

   For example, you might run the `scp` command on your development computer to copy the claim certificate and private key from a folder named `claim-certs` on your development computer to the device.

   - Replace `device-ip-address` with the IP address of your device.

   ```bash
   scp -r claim-certs/ device-ip-address:~
   ```

2. Create the Greengrass root folder on the device. You'll later install the AWS IoT Greengrass Core software to this folder.

   - Replace `/greengrass/v2` with the folder to use.
3. Set the permissions of the parent of the Greengrass root folder.
   - Replace `/greengrass` with the parent of the root folder.

   ```bash
   sudo chmod 755 /greengrass
   ```

4. Move the claim certificates to the Greengrass root folder.
   - Replace `/greengrass/v2` with the Greengrass root folder.

   ```bash
   sudo mv ~/claim-certs /greengrass/v2
   ```

5. Download the Amazon root certificate authority (CA) certificate. AWS IoT certificates are associated with Amazon's root CA certificate by default.

   ```bash
   sudo curl -o /greengrass/v2/AmazonRootCA1.pem https://www.amazontrust.com/repository/AmazonRootCA1.pem
   ```

### Download the AWS IoT Greengrass Core software

You can download the latest version of the AWS IoT Greengrass Core software from the following location:

- [https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip](https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip)

**Note**
You can download a specific version of the AWS IoT Greengrass Core software from the following location. Replace `version` with the version to download.

[https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip](https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip)

### To download the AWS IoT Greengrass Core software (Linux)

1. On your device, download the AWS IoT Greengrass Core software to a file named `greengrass-nucleus-latest.zip`.

   ```bash
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip > greengrass-nucleus-latest.zip
   ```

   By downloading this software, you agree to the Greengrass Core Software License Agreement.

2. Unzip the AWS IoT Greengrass Core software to a folder on your device. Replace `GreengrassInstaller` with the folder that you want to use.

   ```bash
   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-nucleus-latest.zip
   ```
Install with fleet provisioning

Important
If you install a version of the Greengrass nucleus earlier than v2.4.0, don’t remove this folder after you install the AWS IoT Greengrass Core software. The AWS IoT Greengrass Core software uses the files in this folder to run.
If you downloaded the latest version of the software, you install v2.4.0 or later, and you can remove this folder after you install the AWS IoT Greengrass Core software.

3. (Optional) Run the following command to see the version of the AWS IoT Greengrass Core software.

```
java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
```

Download the AWS IoT fleet provisioning plugin

You can download the latest version of the AWS IoT fleet provisioning plugin from the following location:

- https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar

To download the AWS IoT fleet provisioning plugin (Linux)

- On your device, download the AWS IoT fleet provisioning plugin to a file named aws.greengrass.FleetProvisioningByClaim.jar. Replace GreengrassInstaller with the folder that you want to use.

```
curl --s https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar > GreengrassInstaller/aws.greengrass.FleetProvisioningByClaim.jar
```

By downloading this software, you agree to the Greengrass Core Software License Agreement.

Install the AWS IoT Greengrass Core software

Run the installer with arguments that specify the following actions:

- Install from a partial configuration file that specifies to use the fleet provisioning plugin to provision AWS resources. The AWS IoT Greengrass Core software uses a configuration file that specifies the configuration of every Greengrass component on the device. The installer creates a complete configuration file from the partial configuration file that you provide and the AWS resources that the fleet provisioning plugin creates.
- Use the ggc_user system user and ggc_group system group to run software components on the core device. The installer creates this default user and group if they don’t exist.
- Install the software as a system service that runs on boot, if your device has the systemd init system.

For more information about the arguments that you can specify, see Installer arguments (p. 89).

Note
If you are running AWS IoT Greengrass on a device with limited memory, you can control the amount of memory that AWS IoT Greengrass Core software uses. To control memory allocation, you can set JVM heap size options in the jvmOptions configuration parameter in your nucleus component. For more information, see Control memory allocation with JVM options (p. 121).
To install the AWS IoT Greengrass Core software (Linux)

1. Check the version of the AWS IoT Greengrass Core software.
   - Replace `GreengrassInstaller` with the path to the folder that contains the software.

   ```java
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

2. Use a text editor to create a configuration file named `config.yaml` to provide to the installer.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create `config.yaml` in the `GreengrassInstaller` folder.

   ```sh
   nano GreengrassInstaller/config.yaml
   ```

   Copy the following YAML content into the file. This partial configuration file specifies parameters for the fleet provisioning plugin. For more information about the options that you can specify, see Configure the AWS IoT fleet provisioning plugin (p. 81).

   ```yaml
   ---
   services:
     aws.greengrass.Nucleus:
       version: "2.4.0"
     aws.greengrass.FleetProvisioningByClaim:
       configuration:
         rootPath: /greengrass/v2
         awsRegion: "us-west-2"
         iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
         iotCredentialEndpoint: "device-credentials-prefix.credentials.iot.us-west-2.amazonaws.com"
         iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
         provisioningTemplate: "GreengrassFleetProvisioningTemplate"
         claimCertificatePath: "/greengrass/v2/claim-certs/claim.pem.crt"
         claimCertificatePrivateKeyPath: "/greengrass/v2/claim-certs/claim.private.pem.key"
         rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
         templateParameters:
           ThingName: "MyGreengrassCore"
           ThingGroupName: "MyGreengrassCoreGroup"
   ```

   Then, do the following:
   - Replace `2.4.0` with the version of the AWS IoT Greengrass Core software.
   - Replace each instance of `/greengrass/v2` with the Greengrass root folder.
   - Replace `us-west-2` with the AWS Region where you created the provisioning template and other resources.
   - Replace the `iotDataEndpoint` with your AWS IoT data endpoint.
   - Replace the `iotCredentialEndpoint` with your AWS IoT credentials endpoint.
   - Replace `GreengrassCoreTokenExchangeRoleAlias` with the name of the token exchange role alias.
   - Replace `GreengrassFleetProvisioningTemplate` with the name of the fleet provisioning template.
   - Replace the `claimCertificatePath` with the path to the claim certificate on the device.
   - Replace the `claimCertificatePrivateKeyPath` with the path to the claim certificate private key on the device.
**Install with fleet provisioning**

- Replace the template parameters (templateParameters) with the values to use to provision the device. This example refers to the example template (p. 77) that defines ThingName and ThingGroupName parameters.

**Note**

In this configuration file, you can customize other configuration options such as the ports and network proxy to use, as shown in the following example. For more information, see Greengrass nucleus configuration (p. 137).

```yaml
---

services:
  aws.greengrass.Nucleus:
    version: "2.4.0"
    configuration:
      mqtt:
        port: 443
        greengrassDataPlanePort: 443
      networkProxy:
        proxy:
          url: "http://my-proxy-server:1100"
          username: "Mary_Major"
          password: "pass@word1357"
  aws.greengrass.FleetProvisioningByClaim:
    configuration:
      rootPath: /greengrass/v2
      awsRegion: "us-west-2"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      iotCredentialEndpoint: "device-credentials-prefix.credentials.iot.us-west-2.amazonaws.com"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      provisioningTemplate: "GreengrassFleetProvisioningTemplate"
      claimCertificatePath: "/greengrass/v2/claim-certs/claim.pem.crt"
      claimCertificatePrivateKeyPath: "/greengrass/v2/claim-certs/claim.private.pem.key"
      rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
      templateParameters:
        ThingName: "MyGreengrassCore"
        ThingGroupName: "MyGreengrassCoreGroup"
      mqttPort: 443
      proxyUrl: "http://my-proxy-server:1100"
      proxyUserName: "Mary_Major"
      proxyPassword: "pass@word1357"

3. Run the installer. Specify --trusted-plugin to provide the fleet provisioning plugin, and specify --init-config to provide the configuration file.

- Replace /greengrass/v2 with the Greengrass root folder.
- Replace each instance of GreengrassInstaller with the folder where you unpacked the installer.

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \ -jar ./GreengrassInstaller/lib/Greengrass.jar \ --trusted-plugin ./GreengrassInstaller/aws.greengrass.FleetProvisioningByClaim.jar \ --init-config ./GreengrassInstaller/config.yaml \ --component-default-user ggc_user:ggc_group \ --setup-system-service true
```
If you specify `--setup-system-service true`, the installer prints **Successfully set up Nucleus as a system service** if it set up and ran the software as a system service. Otherwise, the installer doesn't output any message if it installs the software successfully.

**Note**
You can't use the `deploy-dev-tools` argument to deploy local development tools when you run the installer without the `--provision true` argument. For information about deploying the Greengrass CLI directly on your device, see [Greengrass Command Line Interface](p. 595).

4. Verify the installation by viewing the files in the root folder.

```bash
ls /greengrass/v2
```

If the installation succeeded, the root folder contains several folders, such as `config`, `packages`, and `logs`.

If you installed the AWS IoT Greengrass Core software as a system service, the installer runs the software for you. Otherwise, you must run the software manually. For more information, see [Run the AWS IoT Greengrass Core software](p. 91).

For more information about how to configure and use the software and AWS IoT Greengrass, see the following:

- Configure the AWS IoT Greengrass Core software (p. 119)
- Develop AWS IoT Greengrass components (p. 322)
- Deploy AWS IoT Greengrass components to devices (p. 383)
- Greengrass Command Line Interface (p. 595)

**Set up AWS IoT fleet provisioning for Greengrass core devices**

To install the AWS IoT Greengrass Core software with fleet provisioning (p. 62), you must first set up the following resources in your AWS account. These resources enable devices to register themselves with AWS IoT and operate as Greengrass core devices. Follow steps in this section once to create and configure these resources in your AWS account.

- A token exchange IAM role, which core devices use to authorize calls to AWS services.
- An AWS IoT role alias that points to the token exchange role.
- (Optional) An AWS IoT policy, which core devices use to authorize calls to the AWS IoT and AWS IoT Greengrass services. This AWS IoT policy must allow the `iot:AssumeRoleWithCertificate` permission for the AWS IoT role alias that points to the token exchange role.

You can use a single AWS IoT policy for all core devices in your fleet, or you can configure your fleet provisioning template to create an AWS IoT policy for each core device.

- An AWS IoT fleet provisioning template. This template must specify the following:
  - An AWS IoT thing resource. You can specify a list of existing thing groups to deploy components to each device when it comes online.
  - An AWS IoT policy resource. This resource can define one of the following properties:
    - The name of an existing AWS IoT policy. If you choose this option, the core devices that you create from this template use the same AWS IoT policy, and you can manage their permissions as a fleet.
    - An AWS IoT policy document. If you choose this option, each core device that you create from this template uses a unique AWS IoT policy, and you can manage permissions for each individual core device.
Install with fleet provisioning

- An AWS IoT certificate resource. This certificate resource must use the AWS::IoT::Certificate::Id parameter to attach the certificate to the core device. For more information, see Just-in-time provisioning in the AWS IoT Developer Guide.

- An AWS IoT provisioning claim certificate and private key for the fleet provisioning template. You can embed this certificate and private key in devices during manufacturing, so the devices can register and provision themselves when they come online.

  **Important**
  
  Provisioning claim private keys should be secured at all times, including on Greengrass core devices. We recommend that you use Amazon CloudWatch metrics and logs to monitor for indications of misuse, such as unauthorized use of the claim certificate to provision devices. If you detect misuse, disable the provisioning claim certificate so that it can't be used for device provisioning. For more information, see Monitoring AWS IoT in the AWS IoT Core Developer Guide.
  
  To help you better manage the number of devices, and which devices, that register themselves in your AWS account, you can specify a pre-provisioning hook when you create a fleet provisioning template. A pre-provisioning hook is an AWS Lambda function that validates template parameters that devices provide during registration. For example, you might create a pre-provisioning hook that checks a device ID against a database to verify that the device has permission to provision. For more information, see Pre-provisioning hooks in the AWS IoT Core Developer Guide.

- An AWS IoT policy that you attach to the provisioning claim certificate to allow devices to register and use the fleet provisioning template.

**Topics**

- Create a token exchange role (p. 55)
- Create an AWS IoT policy (p. 73)
- Create a fleet provisioning template (p. 75)
- Create a provisioning claim certificate and private key (p. 78)

**Create a token exchange role**

Greengrass core devices use an IAM service role, called the token exchange role, to authorize calls to AWS services. The device uses the AWS IoT credentials provider to get temporary AWS credentials for this role, which allows the device to interact with AWS IoT, send logs to Amazon CloudWatch Logs, and download custom component artifacts from Amazon S3. For more information, see Authorize core devices to interact with AWS services (p. 627).

You use an AWS IoT role alias to configure the token exchange role for Greengrass core devices. Role aliases enable you to change the token exchange role for a device but keep the device configuration the same. For more information, see Authorizing direct calls to AWS services in the AWS IoT Core Developer Guide.

In this section, you create a token exchange IAM role and an AWS IoT role alias that points to the role. If you have already set up a Greengrass core device, you can use its token exchange role and role alias instead of creating new ones.

**To create a token exchange IAM role**

1. Create an IAM role that your device can use as a token exchange role. Do the following:
   a. Create a file that contains the trust policy document that the token exchange role requires.

   ```sh
nano device-role-trust-policy.json
   ```
Copy the following JSON into the file.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.iot.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

b. Create the token exchange role with the trust policy document.

- Replace `GreengrassV2TokenExchangeRole` with the name of the IAM role to create.

```bash
aws iam create-role --role-name GreengrassV2TokenExchangeRole --assume-role-policy-document file://device-role-trust-policy.json
```

The response looks similar to the following example, if the request succeeds.

```json
{
  "Role": {
    "Path": "/",
    "RoleName": "GreengrassV2TokenExchangeRole",
    "RoleId": "AROAZ2YMUHYK50X77FB",
    "Arn": "arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole",
    "CreateDate": "2021-02-06T00:13:29+00:00",
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "credentials.iot.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
  }
}
```

c. Create a file that contains the access policy document that the token exchange role requires.

```bash
nano device-role-access-policy.json
```

Copy the following JSON into the file.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:DescribeCertificate",
        "logs:CreateLogGroup",
      ]
    }
  ]
}
```
Install with fleet provisioning

Note
This access policy doesn't allow access to component artifacts in S3 buckets. To deploy custom components that define artifacts in Amazon S3, you must add permissions to the role to allow your core device to retrieve component artifacts. For more information, see Allow access to S3 buckets for component artifacts (p. 628). If you don't yet have an S3 bucket for component artifacts, you can add these permissions later after you create a bucket.

d. Create the IAM policy from the policy document.

- Replace `GreengrassV2TokenExchangeRoleAccess` with the name of the IAM policy to create.

```bash
aws iam create-policy --policy-name GreengrassV2TokenExchangeRoleAccess --policy-document file://device-role-access-policy.json
```

The response looks similar to the following example, if the request succeeds.

```json
{
  "Policy": {
    "PolicyName": "GreengrassV2TokenExchangeRoleAccess",
    "PolicyId": "ANPAZ2YMUHYHACI7C5Z66",
    "Path": "/",
    "DefaultVersionId": "v1",
    "AttachmentCount": 0,
    "PermissionsBoundaryUsageCount": 0,
    "IsAttachable": true,
    "CreateDate": "2021-02-06T00:37:17+00:00",
    "UpdateDate": "2021-02-06T00:37:17+00:00"
  }
}
```

e. Attach the IAM policy to the token exchange role.

- Replace `GreengrassV2TokenExchangeRole` with the name of the IAM role.
- Replace the policy ARN with the ARN of the IAM policy that you created in the previous step.

```bash
```

The command doesn't have any output if the request succeeds.

2. Create an AWS IoT role alias that points to the token exchange role.

- Replace `GreengrassCoreTokenExchangeRoleAlias` with the name of the role alias to create.
- Replace the role ARN with the ARN of the IAM role that you created in the previous step.
Install with fleet provisioning

aws iot create-role-alias --role-alias GreengrassCoreTokenExchangeRoleAlias --role-arn arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole

The response looks similar to the following example, if the request succeeds.

```json
{
    "roleAlias": "GreengrassCoreTokenExchangeRoleAlias",
    "roleAliasArn": "arn:aws:iot:us-west-2:123456789012:rolealias/GreengrassCoreTokenExchangeRoleAlias"
}
```

**Note**

To create a role alias, you must have permission to pass the token exchange IAM role to AWS IoT. If you receive an error message when you try to create a role alias, check that your AWS user has this permission. For more information, see Granting a user permissions to pass a role to an AWS service in the AWS Identity and Access Management User Guide.

Create an AWS IoT policy

After you register a device as an AWS IoT thing, that device can use a digital certificate to authenticate with AWS. This certificate includes one or more AWS IoT policies that define the permissions that a device can use with the certificate. These policies allow the device to communicate with AWS IoT and AWS IoT Greengrass.

With AWS IoT fleet provisioning, devices connect to AWS IoT to create and download a device certificate. In the fleet provisioning template that you create in the next section, you can specify whether AWS IoT attaches the same AWS IoT policy to all devices' certificates, or creates a new policy for each device.

In this section, you create an AWS IoT policy that AWS IoT attaches to all devices' certificates. With this approach, you can manage permissions for all devices as a fleet. If you would rather create a new AWS IoT policy for each device, you can skip this section, and refer to the policy in it when you define your fleet template.

**To create an AWS IoT policy**

- Create an AWS IoT policy that defines the AWS IoT permissions for your fleet of Greengrass core devices. The following policy allows access to all MQTT topics and Greengrass operations, so your device works with custom applications and future changes that require new Greengrass operations. This policy also allows the `iot:AssumeRoleWithCertificate` permission, which allows your devices to use the token exchange role that you created in the previous section. You can restrict this policy down based on your use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices (p. 614).

Do the following:

a. Create a file that contains the AWS IoT policy document that Greengrass core devices require.

```bash
nano greengrass-v2-iot-policy.json
```

Copy the following JSON into the file.

- Replace the `iot:AssumeRoleWithCertificate` resource with the ARN of the AWS IoT role alias that you created in the previous section.
Install with fleet provisioning

b. Create an AWS IoT policy from the policy document.

- Replace `GreengrassV2IoTThingPolicy` with the name of the policy to create.

```bash
aws iot create-policy --policy-name GreengrassV2IoTThingPolicy --policy-document file://greengrass-v2-iot-policy.json
```

The response looks similar to the following example, if the request succeeds.

```json
{
    "policyName": "GreengrassV2IoTThingPolicy",
    "policyDocument": "{ "Version": "2012-10-17", "Statement": [ {
        "Effect": "Allow",
        "Action": [ "iot:Publish", "iot:Subscribe", "iot:Receive", "iot:Connect", "greengrass:*" ],
        "Resource": [ "*" ]
    }, {
        "Effect": "Allow",
        "Action": "iot:AssumeRoleWithCertificate",
    } ]
}
```
Create a fleet provisioning template

AWS IoT fleet provisioning templates define how to provision AWS IoT things, policies, and certificates. To provision Greengrass core devices with the fleet provisioning plugin, you must create a template that specifies that following:

- An AWS IoT thing resource. You can specify a list of existing thing groups to deploy components to each device when it comes online.
- An AWS IoT policy resource. This resource can define one of the following properties:
  - The name of an existing AWS IoT policy. If you choose this option, the core devices that you create from this template use the same AWS IoT policy, and you can manage their permissions as a fleet.
  - An AWS IoT policy document. If you choose this option, each core device that you create from this template uses a unique AWS IoT policy, and you can manage permissions for each individual core device.
- An AWS IoT certificate resource. This certificate resource must use the AWS::IoT::Certificate::Id parameter to attach the certificate to the core device. For more information, see Just-in-time provisioning in the AWS IoT Developer Guide.

In the template, you can specify to add the AWS IoT thing to a list of existing thing groups. When the core device connects to AWS IoT Greengrass for the first time, it receives Greengrass deployments for each thing group where it’s a member. You can use thing groups to deploy the latest software to each device as soon as it comes online. For more information, see Deploy AWS IoT Greengrass components to devices (p. 383).

The AWS IoT service requires permissions to create and update AWS IoT resources in your AWS account when provisioning devices. To give the AWS IoT service access, you create an IAM role and provide it when you create the template. AWS IoT provides an managed policy, AWSIoTThingsRegistration, that allows access to all permissions that AWS IoT might use when provisioning devices. You can use this managed policy, or create a custom policy that scopes down the permissions in the managed policy for your use case.

In this section, you create an IAM role that allows AWS IoT to provision resources for devices, and you create a fleet provisioning template that uses that IAM role.

To create a fleet provisioning template

1. Create an IAM role that AWS IoT can assume to provision resources in your AWS account. Do the following:
   a. Create a file that contains the trust policy document that allows AWS IoT to assume the role.

```
nano aws-iot-trust-policy.json
```

Copy the following JSON into the file.

```json
{
 "Version": "2012-10-17",
 "Statement": [
 { 
 "Effect": "Allow",
 "Principal": {
```
b. Create an IAM role with the trust policy document.
   • Replace `GreengrassFleetProvisioningRole` with the name of the IAM role to create.

   ```bash
   aws iam create-role --role-name GreengrassFleetProvisioningRole
   --assume-role-policy-document file://aws-iot-trust-policy.json
   ```

   The response looks similar to the following example, if the request succeeds.

   ```json
   {  
     "Role": {  
       "Path": "/",  
       "RoleName": "GreengrassFleetProvisioningRole",  
       "RoleId": "AROAZ2YMUHYHK5OM77FB",  
       "Arn": "arn:aws:iam::123456789012:role/GreengrassFleetProvisioningRole",  
       "CreateDate": "2021-07-26T00:15:12+00:00",  
       "AssumeRolePolicyDocument": {  
         "Version": "2012-10-17",  
         "Statement": [  
           {  
             "Effect": "Allow",  
             "Principal": {  
               "Service": "iot.amazonaws.com"  
             },  
             "Action": "sts:AssumeRole"  
           ]  
         ]  
       }  
     }  
   }
   ```

c. Review the `AWSIoTThingsRegistration` policy, which allows access to all permissions that AWS IoT might use when provisioning devices. You can use this managed policy, or create a custom policy that defines scoped-down permissions for your use case. If you choose to create a custom policy, do so now.

d. Attach the IAM policy to the fleet provisioning role.
   • Replace `GreengrassFleetProvisioningRole` with the name of the IAM role.
   • If you created a custom policy in the previous step, replace the policy ARN with the ARN of the IAM policy to use.

   ```bash
   aws iam attach-role-policy --role-name GreengrassFleetProvisioningRole
   --policy-arn arn:aws:iam::aws:policy/service-role/AWSIoTThingsRegistration
   ```

   The command doesn't have any output if the request succeeds.

2. (Optional) Create a pre-provisioning hook, which is an AWS Lambda function that validates template parameters that devices provide during registration. You can use a pre-provisioning hook to gain more control over which and how many devices onboard in your AWS account. For more information, see Pre-provisioning hooks in the AWS IoT Core Developer Guide.

3. Create a fleet provisioning template. Do the following:
a. Create a file to contain the provisioning template document.

```
nano greengrass-fleet-provisioning-template.json
```

Write the provisioning template document. You can start from the following example provisioning template, which specifies to create an AWS IoT thing with the following properties:

- The thing's name is the value that you specify in the `ThingName` template parameter.
- The thing is a member of the thing group that you specify in the `ThingGroupName` template parameter. The thing group must exist in your AWS account.
- The thing's certificate has the AWS IoT policy named `GreengrassV2IoTThingPolicy` attached to it.

For more information, see Provisioning templates in the AWS IoT Core Developer Guide.

```json
{
    "Parameters": {
        "ThingName": {
            "Type": "String"
        },
        "ThingGroupName": {
            "Type": "String"
        },
        "AWS::IoT::Certificate::Id": {
            "Type": "String"
        }
    },
    "Resources": {
        "MyThing": {
            "OverrideSettings": {
                "AttributePayload": "REPLACE",
                "ThingGroups": "REPLACE",
                "ThingTypeName": "REPLACE"
            },
            "Properties": {
                "AttributePayload": {},
                "ThingGroups": [
                    {
                        "Ref": "ThingGroupName"
                    }
                ],
                "ThingName": {
                    "Ref": "ThingName"
                }
            },
            "Type": "AWS::IoT::Thing"
        },
        "MyPolicy": {
            "Properties": {
                "PolicyName": "GreengrassV2IoTThingPolicy"
            },
            "Type": "AWS::IoT::Policy"
        },
        "MyCertificate": {
            "Properties": {
                "CertificateId": {
                    "Ref": "AWS::IoT::Certificate::Id"
                },
                "Status": "Active"
            }
        }
    }
}
b. Create the fleet provisioning template from the provisioning template document.

- Replace `GreengrassFleetProvisioningTemplate` with the name of the template to create.
- Replace the template description with a description for your template.
- Replace the provisioning role ARN with the ARN of the role that you created earlier.

```bash
aws iot create-provisioning-template
  --template-name GreengrassFleetProvisioningTemplate
  --description "A provisioning template for Greengrass core devices." 
  --provisioning-role-arn "arn:aws:iam::123456789012:role/GreengrassFleetProvisioningRole"
  --template-body file://greengrass-fleet-provisioning-template.json
  --enabled
```

**Note**

If you created a pre-provisioning hook, specify the ARN of the pre-provisioning hook's Lambda function with the `--pre-provisioning-hook` argument.

```bash
```

The response looks similar to the following example, if the request succeeds.

```json
{
  "templateName": "GreengrassFleetProvisioningTemplate",
  "defaultVersionId": 1
}
```

Create a provisioning claim certificate and private key

Claim certificates are X.509 certificates that allow devices to register as AWS IoT things and retrieve a unique X.509 device certificate to use for regular operations. After you create a claim certificate, you attach an AWS IoT policy that allows devices to use it to create unique device certificates and provision with a fleet provisioning template. Devices with the claim certificate can provision using only the provisioning template that you allow in the AWS IoT policy.

In this section, you create the claim certificate and configure it for devices to use with the fleet provisioning template that you created in the previous section.

**Important**

Provisioning claim private keys should be secured at all times, including on Greengrass core devices. We recommend that you use Amazon CloudWatch metrics and logs to monitor for indications of misuse, such as unauthorized use of the claim certificate to provision devices. If you detect misuse, disable the provisioning claim certificate so that it can't be used for device provisioning. For more information, see Monitoring AWS IoT in the AWS IoT Core Developer Guide.

To help you better manage the number of devices, and which devices, that register themselves in your AWS account, you can specify a pre-provisioning hook when you create a fleet
provisioning template. A pre-provisioning hook is an AWS Lambda function that validates template parameters that devices provide during registration. For example, you might create a pre-provisioning hook that checks a device ID against a database to verify that the device has permission to provision. For more information, see Pre-provisioning hooks in the AWS IoT Core Developer Guide.

To create a provisioning claim certificate and private key

1. Create a folder where you download the claim certificate and private key.

   ```bash
mkdir claim-certs
   ```

2. Create and save a certificate and private key to use for provisioning. AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA).

   ```bash
aws iot create-keys-and-certificate \
  --certificate-pem-outfile "claim-certs/claim.pem.crt" \
  --public-key-outfile "claim-certs/claim.public.pem.key" \
  --private-key-outfile "claim-certs/claim.private.pem.key" \
  --set-as-active
   ```

   The response contains information about the certificate, if the request succeeds. Save the certificate's ARN to use later.

3. Create and attach an AWS IoT policy that allows devices to use the certificate to create unique device certificates and provision with the fleet provisioning template. The following policy allows access to the device provisioning MQTT API. For more information, see Device provisioning MQTT API in the AWS IoT Core Developer Guide.

   Do the following:

   a. Create a file that contains the AWS IoT policy document that Greengrass core devices require.

      ```bash
nano greengrass-provisioning-claim-iot-policy.json
      ```

      Copy the following JSON into the file.

      - Replace each instance of `region` with the AWS Region where you set up fleet provisioning.
      - Replace each instance of `account-id` with your AWS account ID.
      - Replace each instance of `GreengrassFleetProvisioningTemplate` with the name of the fleet provisioning template that you created in the previous section.

      ```json
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Action": ["iot:Connect"],
            "Resource": "*"
          },
          {
            "Effect": "Allow",
            "Action": ["iot:Publish", "iot:Receive"],
            "Resource": [
            ]
          }
        ]
      }
      ```
Install with fleet provisioning

b. Create an AWS IoT policy from the policy document.

- Replace `GreengrassProvisioningClaimPolicy` with the name of the policy to create.

```bash
aws iot create-policy --policy-name GreengrassProvisioningClaimPolicy --policy-document file://greengrass-provisioning-claim-iot-policy.json
```

The response looks similar to the following example, if the request succeeds.

```json
{
  "policyName": "GreengrassProvisioningClaimPolicy",
  "policyDocument": "{
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": "iot:Connect",
        "Resource": "*
      },
      {
        "Effect": "Allow",
        "Action": ["iot:Publish", "iot:Receive"],
      },
      {
        "Effect": "Allow",
        "Action": ["iot:Publish", "iot:Receive"],
      }
    ]
  }
}
```

"policyVersionId": "1"
4. Attach the AWS IoT policy to the provisioning claim certificate.

   • Replace `GreengrassProvisioningClaimPolicy` with the name of the policy to attach.
   • Replace the target ARN with the ARN of the provisioning claim certificate.

   ```bash
   aws iot attach-policy --policy-name GreengrassProvisioningClaimPolicy
   --target arn:aws:iot:us-west-2:123456789012:cert/aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
   ```

   The command doesn't have any output if the request succeeds.

You now have a provisioning claim certificate and private key that devices can use to register with AWS IoT and provision themselves as Greengrass core devices. You can embed the claim certificate and private key in devices during manufacturing, or copy the certificate and key to devices before you install the AWS IoT Greengrass Core software. For more information, see Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning (p. 62).

**Configure the AWS IoT fleet provisioning plugin**

The AWS IoT fleet provisioning plugin provides the following configuration parameters that you can customize when you install the AWS IoT Greengrass Core software with fleet provisioning (p. 62).

- **rootPath**
  The path to the folder to use as the root for the AWS IoT Greengrass Core software.

- **awsRegion**
  The AWS Region that the fleet provisioning plugin uses to provision AWS resources.

- **iotDataEndpoint**
  The AWS IoT data endpoint for your AWS account.

- **iotCredentialEndpoint**
  The AWS IoT credentials endpoint for your AWS account.

- **iotRoleAlias**
  The AWS IoT role alias that points to a token exchange IAM role. The AWS IoT credentials provider assumes this role to allow the Greengrass core device to interact with AWS services. For more information, see Authorize core devices to interact with AWS services (p. 627).

- **provisioningTemplate**
  The AWS IoT fleet provisioning template to use to provision AWS resources. This template must specify the following:
  • An AWS IoT thing resource. You can specify a list of existing thing groups to deploy components to each device when it comes online.
  • An AWS IoT policy resource. This resource can define one of the following properties:
    • The name of an existing AWS IoT policy. If you choose this option, the core devices that you create from this template use the same AWS IoT policy, and you can manage their permissions as a fleet.
    • An AWS IoT policy document. If you choose this option, each core device that you create from this template uses a unique AWS IoT policy, and you can manage permissions for each individual core device.
• An AWS IoT certificate resource. This certificate resource must use the
AWS::IoT::Certificate::Id parameter to attach the certificate to the core device. For more
information, see Just-in-time provisioning in the AWS IoT Developer Guide.

For more information, see Provisioning templates in the AWS IoT Core Developer Guide.

claimCertificatePath

The path to the provisioning claim certificate for the provisioning template that you specify in
provisioningTemplate. For more information, see CreateProvisioningClaim in the AWS IoT Core
API Reference.

claimCertificatePrivateKeyPath

The path to the provisioning claim certificate private key for the provisioning template that you
specify in provisioningTemplate. For more information, see CreateProvisioningClaim in the AWS
IoT Core API Reference.

Important

Provisioning claim private keys should be secured at all times, including on Greengrass core
devices. We recommend that you use Amazon CloudWatch metrics and logs to monitor for
indications of misuse, such as unauthorized use of the claim certificate to provision devices.
If you detect misuse, disable the provisioning claim certificate so that it can't be used for
device provisioning. For more information, see Monitoring AWS IoT in the AWS IoT Core
Developer Guide.

To help you better manage the number of devices, and which devices, that register
themselves in your AWS account, you can specify a pre-provisioning hook when you create
a fleet provisioning template. A pre-provisioning hook is an AWS Lambda function that
validates template parameters that devices provide during registration. For example, you
might create a pre-provisioning hook that checks a device ID against a database to verify
that the device has permission to provision. For more information, see Pre-provisioning
hooks in the AWS IoT Core Developer Guide.

rootCaPath

The path to the Amazon root certificate authority (CA) certificate.

templateParameters

(Optional) The map of parameters to provide to the fleet provisioning template. For more
information, see Provisioning templates' parameters section in the AWS IoT Core Developer Guide.

deviceId

(Optional) The device identifier to use as the client ID when the fleet provisioning plugin creates an
MQTT connection to AWS IoT.

Default: A random UUID.

mqttPort

(Optional) The port to use for MQTT connections.

Default: 8883

proxyUrl

(Optional) The URL of the proxy server in the format scheme:// userinfo@host:port.

• scheme – The scheme, which must be http.

• userinfo – (Optional) The user name and password information. If you specify this information
  in the url, the Greengrass core device ignores the username and password fields.

• host – The host name or IP address of the proxy server.

• port – (Optional) The port number. If you don't specify the port, then the Greengrass core device
  uses the following default value:

  • http – 80
proxyUserName
(Optional) The user name that authenticates the proxy server.
proxyPassword
(Optional) The user name that authenticates the proxy server.

Install AWS IoT Greengrass Core software with custom resource provisioning

This feature is available for v2.4.0 and later of the Greengrass nucleus component (p. 136).

The AWS IoT Greengrass Core software installer provides a Java interface that you can implement in a custom plugin that provisions required AWS resources. You can develop a provisioning plugin to use custom X.509 client certificates or to run complex provisioning steps that other installation processes don’t support. For more information, see Create your own client certificates in the AWS IoT Core Developer Guide.

To run a custom provisioning plugin when you install the AWS IoT Greengrass Core software, you create a JAR file that you provide to the installer. The installer runs the plugin, and the plugin returns a provisioning configuration that defines the AWS resources for the Greengrass core device. The installer uses this information to configure the AWS IoT Greengrass Core software on the device. For more information, see Develop custom provisioning plugins (p. 87).

Important
Before you download the AWS IoT Greengrass Core software, check that your core device meets the requirements (p. 42) to install and run the AWS IoT Greengrass Core software v2.0.

Topics
- Prerequisites (p. 83)
- Download the AWS IoT Greengrass Core software (p. 47)
- Install the AWS IoT Greengrass Core software (p. 84)
- Develop custom provisioning plugins (p. 87)

Prerequisites

To install the AWS IoT Greengrass Core software with custom provisioning, you must have the following:

- A JAR file for a custom provisioning plugin that implements the DeviceIdentityInterface. The custom provisioning plugin must return values for each system and nucleus configuration parameter. Otherwise, you must provide those values in the configuration file during installation. For more information, see Develop custom provisioning plugins (p. 87).

Download the AWS IoT Greengrass Core software

You can download the latest version of the AWS IoT Greengrass Core software from the following location:

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

Note
You can download a specific version of the AWS IoT Greengrass Core software from the following location. Replace version with the version to download.
To download the AWS IoT Greengrass Core software (Linux)

1. On your device, download the AWS IoT Greengrass Core software to a file named `greengrass-nucleus-latest.zip`.

   ```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip > greengrass-nucleus-latest.zip
   ```

   By downloading this software, you agree to the Greengrass Core Software License Agreement.

2. Unzip the AWS IoT Greengrass Core software to a folder on your device. Replace `GreengrassInstaller` with the folder that you want to use.

   ```
   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-nucleus-latest.zip
   ```

   **Important**
   - If you install a version of the Greengrass nucleus earlier than v2.4.0, don't remove this folder after you install the AWS IoT Greengrass Core software. The AWS IoT Greengrass Core software uses the files in this folder to run.
   - If you downloaded the latest version of the software, you install v2.4.0 or later, and you can remove this folder after you install the AWS IoT Greengrass Core software.

3. (Optional) Run the following command to see the version of the AWS IoT Greengrass Core software.

   ```
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

Install the AWS IoT Greengrass Core software

Run the installer with arguments that specify the following actions:

- Install from a partial configuration file that specifies to use your custom provisioning plugin to provision AWS resources. The AWS IoT Greengrass Core software uses a configuration file that specifies the configuration of every Greengrass component on the device. The installer creates a complete configuration file from the partial configuration file that you provide and the AWS resources that the custom provisioning plugin creates.
- Use the `ggc_user` system user and `ggc_group` system group to run software components on the core device. The installer creates this default user and group if they don't exist.
- Install the software as a system service that runs on boot, if your device has the `systemd` init system.

For more information about the arguments that you can specify, see Installer arguments (p. 89).

**Note**

If you are running AWS IoT Greengrass on a device with limited memory, you can control the amount of memory that AWS IoT Greengrass Core software uses. To control memory allocation, you can set JVM heap size options in the `jvmOptions` configuration parameter in your nucleus component. For more information, see Control memory allocation with JVM options (p. 121).

To install the AWS IoT Greengrass Core software (Linux)

1. Check the version of the AWS IoT Greengrass Core software.

   - Replace `GreengrassInstaller` with the path to the folder that contains the software.
Install with custom provisioning

2. Use a text editor to create a configuration file named `config.yaml` to provide to the installer.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create `config.yaml` in the `GreengrassInstaller` folder.

```
nano GreengrassInstaller/config.yaml
```

   Copy the following YAML content into the file.

   ```yaml
---
system:
  rootpath: "/greengrass/v2"
  # The following values are optional. Return them from the provisioning plugin or set them here.
  # certificateFilePath: ""
  # privateKeyPath: ""
  # rootCaPath: ""
  # thingName: ""
services:
  aws.greengrass.Nucleus:
    version: "2.4.0"
    configuration:
      # The following values are optional. Return them from the provisioning plugin or set them here.
      # awsRegion: ""
      # iotRoleAlias: ""
      # iotDataEndpoint: ""
      # iotCredEndpoint: ""
  com.example.CustomProvisioning:
    configuration:
      # You can specify configuration parameters to provide to your plugin.
      # pluginParameter: ""
---
```

   Then, do the following:

   - Replace `2.4.0` with the version of the AWS IoT Greengrass Core software.
   - Replace each instance of `/greengrass/v2` with the Greengrass root folder.
   - (Optional) Specify system and nucleus configuration values. You must set these values if your provisioning plugin doesn't provide them.
   - (Optional) Specify configuration parameters to provide to your provisioning plugin.

**Note**

In this configuration file, you can customize other configuration options, such as the ports and network proxy to use, as shown in the following example. For more information, see `Greengrass nucleus configuration` (p. 137).
# thingName: ""

services:
  aws.greengrass.Nucleus:
    version: "2.4.0"
    configuration:
      mqtt:
        port: 443
greengrassDataPlanePort: 443
      networkProxy:
      proxy:
        url: "http://my-proxy-server:1100"
        username: "Mary_Major"
        password: "pass@word1357"

# The following values are optional. Return them from the provisioning plugin or set them here.
# awsRegion: ""
# iotRoleAlias: ""
# iotDataEndpoint: ""
# iotCredEndpoint: ""

com.example.CustomProvisioning:
  configuration:
    # You can specify configuration parameters to provide to your plugin.
    # pluginParameter: ""

3. Run the installer. Specify --trusted-plugin to provide your custom provisioning plugin, and specify --init-config to provide the configuration file.

- Replace /greengrass/v2 with the Greengrass root folder.
- Replace each instance of GreengrassInstaller with the folder where you unpacked the installer.
- Replace the path to the custom provisioning plugin JAR file with the path to your plugin's JAR file.

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \\
  -jar ./GreengrassInstaller/lib/Greengrass.jar \\
  --trusted-plugin /path/to/com.example.CustomProvisioning.jar \\
  --init-config ./GreengrassInstaller/config.yaml \\
  --component-default-user ggc_user:ggc_group \\
  --setup-system-service true
```

If you specify --setup-system-service true, the installer prints Successfully set up Nucleus as a system service if it set up and ran the software as a system service. Otherwise, the installer doesn't output any message if it installs the software successfully.

**Note**
You can't use the deploy-dev-tools argument to deploy local development tools when you run the installer without the -- provision true argument. For information about deploying the Greengrass CLI directly on your device, see Greengrass Command Line Interface (p. 595).

4. Verify the installation by viewing the files in the root folder.

```
ls /greengrass/v2
```

If the installation succeeded, the root folder contains several folders, such as config, packages, and logs.
If you installed the AWS IoT Greengrass Core software as a system service, the installer runs the software for you. Otherwise, you must run the software manually. For more information, see Run the AWS IoT Greengrass Core software (p. 91).

For more information about how to configure and use the software and AWS IoT Greengrass, see the following:

- Configure the AWS IoT Greengrass Core software (p. 119)
- Develop AWS IoT Greengrass components (p. 322)
- Deploy AWS IoT Greengrass components to devices (p. 383)
- Greengrass Command Line Interface (p. 595)

**Develop custom provisioning plugins**

To develop a custom provisioning plugin, create a Java class that implements the `com.aws.greengrass.provisioning.DeviceIdentityInterface` interface. You can include the Greengrass nucleus JAR file in your project to access this interface and its classes. This interface defines a method that inputs a plugin configuration and outputs a provisioning configuration. The provisioning configuration defines configurations for the system and the Greengrass nucleus component (p. 136). The AWS IoT Greengrass Core software installer uses this provisioning configuration to configure the AWS IoT Greengrass Core software on a device.

After you develop a custom provisioning plugin, build it as a JAR file that you can provide to the AWS IoT Greengrass Core software installer to run your plugin during installation. The installer runs your custom provisioning plugin in the same JVM that the installer uses, so you can create a JAR that contains only your plugin code.

**Requirements**

To develop a custom provisioning plugin, you must create a Java class that meets the following requirements:

- Uses the `com.aws.greengrass` package, or a package within the `com.aws.greengrass` package.
- Has a constructor without any arguments.
- Implements the `DeviceIdentityInterface` interface. For more information, see Implement the `DeviceIdentityInterface` interface (p. 87).

**Implement the DeviceIdentityInterface interface**

To use the `com.aws.greengrass.provisioning.DeviceIdentityInterface` interface in your custom plugin, add the Greengrass nucleus as a dependency to your project.

**To use the DeviceIdentityInterface in a custom provisioning plugin project**

- You can add the Greengrass nucleus JAR file as a library, or add the Greengrass nucleus as a Maven dependency. Do one of the following:
  - To add the Greengrass nucleus JAR file as a library, download the AWS IoT Greengrass Core software, which contains the Greengrass nucleus JAR. You can download the latest version of the AWS IoT Greengrass Core software from the following location:
Install with custom provisioning

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

You can find the Greengrass nucleus JAR file (Greengrass.jar) in the lib folder in the ZIP file. Add this JAR file to your project.

- To consume the Greengrass nucleus in a Maven project, add a dependency the on the nucleus artifact in the com.aws.greengrass group. You must also add the greengrass-common repository, because the Greengrass nucleus isn't available in the Maven Central Repository.

```xml
<project ...>
  ...
  <repositories>
    <repository>
      <id>greengrass-common</id>
      <name>greengrass common</name>
      <url>https://d2jrmugq4soldf.cloudfront.net/snapshots</url>
    </repository>
  </repositories>
  ...
  <dependencies>
    <dependency>
      <groupId>com.aws.greengrass</groupId>
      <artifactId>nucleus</artifactId>
      <version>2.4.0-SNAPSHOT</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

The `DeviceIdentityInterface` interface

The com.aws.greengrass.provisioning.DeviceIdentityInterface interface has the following shape.

**Note**
You can also explore these classes in the com.aws.greengrass.provisioning package of the Greengrass nucleus source code on GitHub.

```java
public interface com.aws.greengrass.provisioning.DeviceIdentityInterface {
  ProvisionConfiguration updateIdentityConfiguration(ProvisionContext context)
    throws RetryableProvisioningException, InterruptedException;

  // Return the name of the plugin.
  String name();
}
```

```java
com.aws.greengrass.provisioning.ProvisionConfiguration {
  SystemConfiguration systemConfiguration;
  NucleusConfiguration nucleusConfiguration
}
```

```java
com.aws.greengrass.provisioning.ProvisionConfiguration.SystemConfiguration {
  String certificateFilePath;
  String privateKeyPath;
  String rootCAPath;
  String thingName;
}
```

```java
  String awsRegion;
  String iotCredentialsEndpoint;
```
Each configuration value in the `SystemConfiguration` and `NucleusConfiguration` is required to install the AWS IoT Greengrass Core software, but you can return `null`. If your custom provisioning plugin returns `null` for any configuration value, you must provide that value in the system or nucleus configuration when you create the `config.yaml` file to provide to the AWS IoT Greengrass Core software installer. If your custom provisioning plugin returns a non-null value for an option that you also define in `config.yaml`, then the installer replaces the value in `config.yaml` with the value returned by the plugin.

### Installer arguments

The AWS IoT Greengrass Core software includes an installer that sets up the software and provisions the required AWS resources for the Greengrass core device to run. The installer includes the following arguments that you can specify to configure the installation:

- `-h`, `--help`
  (Optional) Show the installer's help information.
- `--version`
  (Optional) Show the version of the AWS IoT Greengrass Core software.
- `-Droot`, `--aws-region`
  The AWS Region that the AWS IoT Greengrass Core software uses to retrieve or create its required AWS resources.
- `-p`, `--provision`
  (Optional) You can register this device as an AWS IoT thing and provision the AWS resources that the core device requires. If you specify `true`, the AWS IoT Greengrass Core software provisions an AWS IoT thing, (optional) an AWS IoT thing group, an IAM role, and an AWS IoT role alias.
  Default: `false`
- `-tn`, `--thing-name`
  (Optional) The name of the AWS IoT thing that you register as this core device. If the thing with the name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it.
  **Note**
  The thing name can't contain colon (`:`) characters.
You must specify `--provision true` to apply this argument.

Default: `GreengrassV2IoTThing_` plus a random UUID.

- **tgn**, `--thing-group-name`

  (Optional) The name of the AWS IoT thing group where you add this core device's AWS IoT thing. If a deployment targets this thing group, this core device receives that deployment when it connects to AWS IoT Greengrass. If the thing group with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it.

  **Note**
  The thing group name can't contain colon (`:`) characters.

  You must specify `--provision true` to apply this argument.

- **tpn**, `--thing-policy-name`

  This feature is available for v2.4.0 and later of the Greengrass nucleus component (p. 136).

  (Optional) The name of the AWS IoT policy to attach to this core device's AWS IoT thing certificate. If the AWS IoT policy with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it.

  The AWS IoT Greengrass Core software creates a permissive AWS IoT policy by default. You can scope down this policy, or create a custom policy where you restrict permissions for your use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices (p. 614).

  You must specify `--provision true` to apply this argument.

  Default: `GreengrassV2IoTThingPolicy`

- **trn**, `--tes-role-name`

  (Optional) The name of the IAM role to use to acquire AWS credentials that let the core device interact with AWS services. If the role with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it with the `GreengrassV2TokenExchangeRoleAccess` policy. This role doesn't have access to your S3 buckets where you host component artifacts. So, you must add permissions to your artifacts' S3 buckets and objects when you create a component. For more information, see Authorize core devices to interact with AWS services (p. 627).

  You must specify `--provision true` to apply this argument.

  Default: `GreengrassV2TokenExchangeRole`

- **tra**, `--tes-role-alias-name`

  (Optional) The name of the AWS IoT role alias that points to the IAM role that provides AWS credentials for this core device. If the role alias with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it and points it to the IAM role that you specify.

  You must specify `--provision true` to apply this argument.

  Default: `GreengrassV2TokenExchangeRoleAlias`

- **ss**, `--setup-system-service`

  (Optional) You can set up the AWS IoT Greengrass Core software as a system service that runs when this device boots. The system service name is `greengrass`. For more information, see Configure AWS IoT Greengrass as a system service (p. 119).

  This argument requires that the systemd init system is available on the device.

  Default: `false`
Run the AWS IoT Greengrass Core software

After you install the AWS IoT Greengrass Core software (p. 45), run it to connect your device to AWS IoT Greengrass.

When you install the AWS IoT Greengrass Core software, you can specify whether to install it as a system service with `systemd`. If you choose this option, the installer runs the software for you and configures it to run when your device boots.

**Topics**
- Check if the AWS IoT Greengrass Core software runs as a system service (p. 92)
- Run the AWS IoT Greengrass Core software as a system service (p. 92)
Check if the AWS IoT Greengrass Core software runs as a system service

When you install the AWS IoT Greengrass Core software, you can specify the `--setup-system-service true` argument to install the AWS IoT Greengrass Core software as a systemd service. If you use this option, the installer runs the software for you and configures it to run when your device boots. The installer outputs the following message if it successfully installs the AWS IoT Greengrass Core software as a system service.

```
Successfully set up Nucleus as a system service
```

If you previously installed the AWS IoT Greengrass Core software and don’t have the installer output, you can check if the software installed as a system service.

**To check if the AWS IoT Greengrass Core software is installed as a system service**

- Run the following command to check the status of the Greengrass system service.

```
sudo systemctl status greengrass.service
```

The response looks similar to the following example if the AWS IoT Greengrass Core software is installed as a system service and active.

```
# greengrass.service - Greengrass Core
Loaded: loaded (/etc/systemd/system/greengrass.service; enabled; vendor preset: disabled)
Active: active (running) since Thu 2021-02-11 01:33:44 UTC; 4 days ago
Main PID: 16107 (sh)
CGroup: /system.slice/greengrass.service
 #16107 /bin/sh /greengrass/v2/alts/current/distro/bin/loader
 #16111 java -Dlog.store=FILE -Droot=/greengrass/v2 -jar /greengrass/v2/
 alts/current/distro/lib/Greengrass...
```

If `systemctl` or `greengrass.service` isn't found, the AWS IoT Greengrass Core software isn't installed as a system service. To run the software, see [Run the AWS IoT Greengrass Core software without a system service](#).

**Run the AWS IoT Greengrass Core software as a system service**

If the AWS IoT Greengrass Core software is installed as a system service, you can use the system service manager to start, stop, and manage the software. For more information, see [Configure AWS IoT Greengrass as a system service](#).

**To run the AWS IoT Greengrass Core software (systemd)**

- Run the following command to start the AWS IoT Greengrass Core software.

```
sudo systemctl start greengrass.service
```
Run the AWS IoT Greengrass Core software without a system service

If the AWS IoT Greengrass Core software isn't installed as a system service, you can run the software's loader script to run the software.

**To run the AWS IoT Greengrass Core software without a system service**

- Run the following command to start the AWS IoT Greengrass Core software.
  - Replace `/greengrass/v2` with the Greengrass root folder that you use.

```bash
sudo /greengrass/v2/alts/current/distro/bin/loader
```

The software prints the following message if it launches successfully.

```
Launched Nucleus successfully.
```

**Note**

If you run this command in a terminal, you must keep the terminal session open to keep the AWS IoT Greengrass Core software running.

Run AWS IoT Greengrass Core software in a Docker container

AWS IoT Greengrass can be configured to run in a Docker container. Docker is a platform that provides the tools for you to build, run, test, and deploy applications that are based on Linux containers. When you run an AWS IoT Greengrass Docker image, you can choose whether to provide your AWS credentials to the Docker container and allow the AWS IoT Greengrass Core software installer to automatically provision the AWS resources that a Greengrass core device requires to operate. If you don't want to provide AWS credentials, then you can manually provision AWS resources and run AWS IoT Greengrass Core software in the Docker container.

**Topics**

- Supported platforms and requirements (p. 93)
- AWS IoT Greengrass Docker software downloads (p. 94)
- Choose how to provision AWS resources (p. 95)
- Build the AWS IoT Greengrass container image from a Dockerfile (p. 95)
- Run AWS IoT Greengrass in a Docker container with automatic resource provisioning (p. 99)
- Run AWS IoT Greengrass in a Docker container with manual resource provisioning (p. 104)
- Troubleshooting AWS IoT Greengrass in a Docker container (p. 117)

**Supported platforms and requirements**

Host computers must meet the following minimum requirements to install and run the AWS IoT Greengrass Core software in a Docker container:

- A Linux-based operating system with an internet connection.
• **Docker Engine** version 18.09 or later.
• (Optional) **Docker Compose** version 1.22 or later. Docker Compose is required only if you want to use the Docker Compose CLI to run your Docker images.

To run Lambda function components inside of the Docker container, you must configure the container to meet additional requirements. For more information, see **Requirements to run Lambda functions (p. 43)**.

**Run components in process mode**

AWS IoT Greengrass doesn't support running Lambda functions or AWS-provided components in an isolated runtime environment inside the AWS IoT Greengrass Docker container. You must run these components in process mode without any isolation.

When you configure a Lambda function component, set the isolation mode to **No container**. For more information, see **Run AWS Lambda functions (p. 361)**.

When you deploy any of the following AWS-provided components, update the configuration for each component to set the `containerMode` parameter to `NoContainer`. For more information about configuration updates, see **Update component configurations (p. 390)**.

- CloudWatch metrics (p. 153)
- Device Defender (p. 170)
- Kinesis Data Firehose (p. 180)
- Modbus-RTU protocol adapter (p. 264)
- Amazon SNS (p. 298)

**AWS IoT Greengrass Docker software downloads**

AWS IoT Greengrass provides the following Dockerfile and Docker images that make it easier for you to run AWS IoT Greengrass Core software in a Docker container.

**Dockerfile**

AWS IoT Greengrass provides a Dockerfile to build a container image that has AWS IoT Greengrass Core software and dependencies installed on an Amazon Linux 2 (x86_64) base image. You can modify the base image in the Dockerfile to run AWS IoT Greengrass on a different platform architecture.

Download the Dockerfile package from GitHub.

For information about building the AWS IoT Greengrass container image from the Dockerfile, see **Build the AWS IoT Greengrass container image from a Dockerfile (p. 95)**.

**Docker images**

AWS IoT Greengrass provides a Docker image that has AWS IoT Greengrass Core software and dependencies installed on an Amazon Linux 2 (x86_64) base image. To run AWS IoT Greengrass in Docker on a different platform architecture, use the Dockerfile to build a container image for that platform.

Download a prebuilt image from Docker Hub or Amazon Elastic Container Registry (Amazon ECR).

To pull the AWS IoT Greengrass Docker image from Docker Hub or Amazon ECR, run the following command.
Choose how to provision AWS resources

When you install the AWS IoT Greengrass Core software in a Docker container, you can choose whether to automatically provision the AWS resources that a Greengrass core device requires to operate, or to use resources that you manually provision.

• **Automatic resource provisioning**—The installer provisions the AWS IoT thing, AWS IoT thing group, IAM role, and AWS IoT role alias when you run the AWS IoT Greengrass container image for the first time. The installer can also deploy the local development tools to the core device, so you can use the device to develop and test custom software components. To automatically provision these resources, you must provide AWS credentials as environment variables to the Docker image.

  To use automatic provisioning, you must set the Docker environment variable `PROVISION=true` and mount a credential file to provide your AWS credentials to the container.

• **Manual resource provisioning**—If you don’t want to provide AWS credentials to the container, then you can manually provision the AWS resources before you run the AWS IoT Greengrass container image. You must create a configuration file to provide information about these resources to the AWS IoT Greengrass Core software installer within the Docker container.

  To use manual provisioning, you must set the Docker environment variable `PROVISION=false`. Manual provisioning is the default option.

For more information, see [Build the AWS IoT Greengrass container image from a Dockerfile](p. 95).

Build the AWS IoT Greengrass container image from a Dockerfile

AWS provides a Dockerfile that you can download and use to run AWS IoT Greengrass Core software in a Docker container. Dockerfiles contain source code for building AWS IoT Greengrass container images.

Before you build an AWS IoT Greengrass container image, you must configure your Dockerfile to select the version of AWS IoT Greengrass Core software that you want to install. You can also configure environment variables to choose how to provision resources during installation, and customize other installation options. This section describes how to configure and build an AWS IoT Greengrass Docker image from a Dockerfile.

Download the Dockerfile package

You can download the AWS IoT Greengrass Dockerfile package from GitHub:
aws-greengrass-docker/releases

After you download the package, extract the contents to the download-directory/aws-greengrass-docker-nucleus-version folder on your computer.

Specify the AWS IoT Greengrass Core software version

Use the following build argument in the Dockerfile to specify the version of the AWS IoT Greengrass Core software that you want to use in the AWS IoT Greengrass Docker image. By default, the Dockerfile uses the latest version of the AWS IoT Greengrass Core software.

GREENGRASS_RELEASE_VERSION

The version of the AWS IoT Greengrass Core software. By default, the Dockerfile downloads the latest available version of the Greengrass nucleus. Set the value to the version of the nucleus that you want to download.

Set environment variables

Environment variables enable you to customize how AWS IoT Greengrass Core software is installed in the Docker container. You can set environment variables for your AWS IoT Greengrass Docker image in various ways.

- To use the same environment variables to create multiple images, set environment variables directly in the Dockerfile.
- If you use docker run to start your container, pass environment variables as arguments in the command, or set environment variables in an environment variables file and then pass the file as an argument. For more information about setting environment variables in Docker, see the Docker documentation.
- If you use docker-compose up to start your container, set environment variables in an environment variables file and then pass the file as an argument. For more information about setting environment variables in Compose, see the Docker documentation.

You can configure the following environment variables for the AWS IoT Greengrass Docker image.

Note

Don't modify the TINI_KILL_PROCESS_GROUP variable in the Dockerfile. This variable allows forwarding SIGTERM to all PIDs in the PID group so that AWS IoT Greengrass Core software can shut down correctly when the Docker container is stopped.

GGC_ROOT_PATH

(Optional) The path to the folder within the container to use as the root for AWS IoT Greengrass Core software.

Default: /greengrass/v2

PROVISION

(Optional) Determines whether the AWS IoT Greengrass Core provisions AWS resources.

- If you specify true, AWS IoT Greengrass Core software registers the container image as an AWS IoT thing and provisions the AWS resources that the Greengrass core device requires. The AWS IoT Greengrass Core software provisions an AWS IoT thing, (optional) an AWS IoT thing group, an IAM role, and an AWS IoT role alias. For more information, see Run AWS IoT Greengrass in a Docker container with automatic resource provisioning (p. 99).
- If you specify false, then you must create a configuration file to provide to the AWS IoT Greengrass Core installer that specifies to use the AWS resources and certificates that you
manually created. For more information, see Run AWS IoT Greengrass in a Docker container with manual resource provisioning (p. 104).

Default: false

AWS_REGION

(Optional) The AWS Region that the AWS IoT Greengrass Core software uses to retrieve or create required AWS resources.

Default: us-east-1.

THING_NAME

(Optional) The name of the AWS IoT thing that you register as this core device. If the thing with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it.

You must specify PROVISION=true to apply this argument.

Default: GreengrassV2IotThing_ plus a random UUID.

THING_GROUP_NAME

(Optional) The name of the AWS IoT thing group where you add this core device's AWS IoT If a deployment targets this thing group, this and other core devices in that group receive that deployment when it connects to AWS IoT Greengrass. If the thing group with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it.

You must specify PROVISION=true to apply this argument.

TES_ROLE_NAME

(Optional) The name of the IAM role to use to acquire AWS credentials that let the Greengrass core device interact with AWS services. If the role with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it with the GreengrassV2TokenExchangeRoleAccess policy. This role doesn't have access to your S3 buckets where you host component artifacts. So, you must add permissions to your artifacts' S3 buckets and objects when you create a component. For more information, see Authorize core devices to interact with AWS services (p. 627).

Default: GreengrassV2TokenExchangeRole

TES_ROLE_ALIAS_NAME

(Optional) The name of the AWS IoT role alias that points to the IAM role that provides AWS credentials for the Greengrass core device. If the role alias with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it and points it to the IAM role that you specify.

Default: GreengrassV2TokenExchangeRoleAlias

COMPONENT_DEFAULT_USER

(Optional) The name or ID of the system user and group that the AWS IoT Greengrass Core software uses to run components. Specify the user and group, separated by a colon. The group is optional. For example, you can specify ggc_user:ggc_group or ggc_user.

• If you run as root, this defaults to the user and group that the configuration file defines. If the configuration file doesn't define a user and group, this defaults to ggc_user:ggc_group. If ggc_user or ggc_group don't exist, the software creates them.

• If you run as a non-root user, the AWS IoT Greengrass Core software uses that user to run components.

• If you don't specify a group, the AWS IoT Greengrass Core software uses the primary group of the system user.

For more information, see Configure the user and group that run components (p. 122).
DEPLOY_DEVTOOLS

(Optional) Defines whether to download and deploy the Greengrass CLI component in the container image. You can use the Greengrass CLI to develop and debug components locally.

Default: false

Note
Use local development tools in development environments only. Don't specify this option on production devices.

INIT_CONFIG

(Optional) The path to the configuration file to use to install the AWS IoT Greengrass Core software. You can use this option to set up new Greengrass core devices with a specific nucleus configuration, or to specify manually provisioned resources, for example. You must mount your configuration file to the path that you specify in this argument.

TRUSTED_PLUGIN

This feature is available for v2.4.0 and later of the Greengrass nucleus component (p. 136).

(Optional) The path to a JAR file to load as a trusted plugin. Use this option to provide provisioning plugin JAR files, such as to install with fleet provisioning (p. 62) or custom provisioning (p. 83).

THING_POLICY_NAME

This feature is available for v2.4.0 and later of the Greengrass nucleus component (p. 136).

(Optional) The name of the AWS IoT policy to attach to this core device's AWS IoT thing certificate. If the AWS IoT policy with this name doesn't exist in your AWS account the AWS IoT Greengrass Core software creates it.

You must specify PROVISION=true to apply this argument.

Note
The AWS IoT Greengrass Core software creates a permissive AWS IoT policy by default. You can scope down this policy, or create a custom policy where you restrict permissions for your use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices (p. 614).

Specify the dependencies to install

The RUN instruction in the AWS IoT Greengrass Dockerfile prepares up the container environment to run the AWS IoT Greengrass Core software installer. You can customize the dependencies that are installed before the AWS IoT Greengrass Core software installer runs in the Docker container.

Build the AWS IoT Greengrass image

Use the AWS IoT Greengrass Dockerfile to build an AWS IoT Greengrass container image. You can use the Docker CLI or the Docker Compose CLI to build the image and start the container. You can also use the Docker CLI to build the image and then use Docker Compose to start your container from that image.

Docker

1. On the host machine, run the following command to switch to the directory that contains the configured Dockerfile.

   cd download-directory/aws-greengrass-docker-nucleus-version

2. Run the following command to build the AWS IoT Greengrass container image from the Dockerfile.
Run AWS IoT Greengrass in Docker with automatic provisioning

sudo docker build -t "platform/aws-iot-greengrass:nucleus-version" .

Docker Compose

1. On the host machine, run the following command to switch to the directory that contains the Dockerfile and the Compose file.

   cd download-directory/aws-greengrass-docker-nucleus-version

2. Run the following command to use the Compose file to build the AWS IoT Greengrass container image.

   docker-compose -f docker-compose.yml build

You have successfully created the AWS IoT Greengrass container image. The Docker image has the AWS IoT Greengrass Core software installed. You can now run the AWS IoT Greengrass Core software in a Docker container.

**Run AWS IoT Greengrass in a Docker container with automatic resource provisioning**

This tutorial shows you how to install and run AWS IoT Greengrass Core software in a Docker container with automatically provisioned AWS resources and local development tools. You can use this development environment to explore AWS IoT Greengrass features in a Docker container.

**Prerequisites**

To complete this tutorial, you need the following.

- An AWS account. If you don't have one, see Set up an AWS account (p. 44).
- An AWS IAM user with permissions to provision the AWS IoT and IAM resources for a Greengrass core device. The AWS IoT Greengrass Core software installer uses your AWS credentials to automatically provision these resources. For information about the minimal IAM policy to automatically provision resources, see Minimal IAM policy for installer to provision resources (p. 629).
- An AWS IoT Greengrass Docker image. This tutorial shows you how to pull the AWS IoT Greengrass Docker image from Docker Hub. You can also pull the AWS IoT Greengrass Docker image (p. 94) from Amazon Elastic Container Registry (Amazon ECR), or you can build an image from the AWS IoT Greengrass Dockerfile (p. 95).
- A Linux-based operating system with an internet connection.
- Docker Engine version 18.09 or later.
- (Optional) Docker Compose version 1.22 or later. Docker Compose is required only if you want to use the Docker Compose CLI to run your Docker images.

**Configure your AWS credentials**

In this step, you create a credential file on the host computer that contains your AWS security credentials. When you run the AWS IoT Greengrass Docker image, you must mount the folder that contains this credential file to /root/.aws/ in the Docker container. The AWS IoT Greengrass installer uses these credentials to provision resources in your AWS account. For information about the minimal...
I AM policy that the installer requires to automatically provision resources, see Minimal IAM policy for installer to provision resources (p. 629).

1. Retrieve one of the following.

   - Long-term credentials for an IAM user. For information about how to retrieve long-term credentials, see Managing access keys for IAM users in the IAM User Guide.
   - (Recommended) Temporary credentials for an IAM role. For information about how to retrieve temporary credentials, see Using temporary security credentials with the AWS CLI in the IAM User Guide.

2. Create a folder where you place your credential file.

   ```bash
   mkdir ./greengrass-v2-credentials
   ```

3. Use a text editor to create a configuration file named `credentials` in the `./greengrass-v2-credentials` folder.

   For example, you can run the following command to use GNU nano to create the `credentials` file.

   ```bash
   nano ./greengrass-v2-credentials/credentials
   ```

4. Add your AWS credentials to the `credentials` file in the following format.

   ```ini
   [default]
   aws_access_key_id = AKIAIOSFODNN7EXAMPLE
   aws_secret_access_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
   aws_session_token = AQoEXAMPLEH4aoAH0gNCAFY...truncated...zrkuWJ0gQs81ZZa1v2B1a2R4Olgk
   ```

   Include `aws_session_token` for temporary credentials only.

   **Note**
   Remove the credential file from the host computer after you start the AWS IoT Greengrass container. If you don't remove the credential file, then your AWS credentials will remain mounted inside the container. For more information, see Run the AWS IoT Greengrass Core software in a container (p. 101).

Create an environment file

This tutorial uses an environment file to set the environment variables that will be passed to the AWS IoT Greengrass Core software installer inside the Docker container. You can also use the `-e` or `--env` argument in your `docker run` command to set environment variables in the Docker container or you can set the variables in an environment block in the `docker-compose.yml` file.

1. Use a text editor to create an environment file named `.env`.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the `.env` in the current directory.

   ```bash
   nano .env
   ```

2. Copy the following content into the file.

   ```
   GGC_ROOT_PATH=/greengrass/v2
   AWS_REGION=region
   PROVISION=true
   THING_NAME=MyGreengrassCore
   ```
Run AWS IoT Greengrass in Docker with automatic provisioning

```
THING_GROUP_NAME=MyGreengrassCoreGroup
TES_ROLE_NAME=GreengrassV2TokenExchangeRole
TES_ROLE_ALIAS_NAME=GreengrassCoreTokenExchangeRoleAlias
COMPONENT_DEFAULT_USER=ggc_user:ggc_group
DEPLOY_DEV_TOOLS=true
```

Then, replace the following values.

- `/greengrass/v2`. The Greengrass root folder that you want to use for installation. You use the `GGC_ROOT` environment variable to set this value.
- `region`. The AWS Region where you created the resources.
- `MyGreengrassCore`. The name of the AWS IoT thing. If the thing doesn't exist, the installer creates it. The installer downloads the certificates to authenticate as the AWS IoT thing.
- `MyGreengrassCore`. The name of the AWS IoT thing group. If the thing group doesn't exist, the installer creates it and adds the thing to it. If the thing group exists and has an active deployment, the core device downloads and runs the software that the deployment specifies.
- `GreengrassV2TokenExchangeRole`. Replace with the name of the IAM token exchange role that allows the Greengrass core device to get temporary AWS credentials. If the role doesn't exist, the installer creates it and creates and attaches a policy named `GreengrassV2TokenExchangeRoleAccess`. For more information, see Authorize core devices to interact with AWS services (p. 627).
- `GreengrassCoreTokenExchangeRoleAlias`. The token exchange role alias. If the role alias doesn't exist, the installer creates it and points it to the IAM token exchange role that you specify. For more information, see

**Note**

Setting the `DEPLOY_DEV_TOOLS` environment variable to `true` deploys local development tools that enable custom component development inside of the Docker container. Don't use this environment variable for production devices.

Run the AWS IoT Greengrass Core software in a container

This tutorial shows you how to pull the latest AWS IoT Greengrass Docker image from Docker Hub and start the Docker container. You can use the Docker CLI or the Docker Compose CLI to run the AWS IoT Greengrass Core software image in a Docker container.

Docker

1. Run the following command to pull the latest AWS IoT Greengrass Docker image from Docker Hub.

   ```
docker pull amazon/aws-iot-greengrass:latest
   
   This example command uses the following arguments for `docker run`:
   ```

2. Run the following command to start the Docker container. This command runs the Greengrass Docker image that you downloaded from Docker Hub. If you use a Docker image from a different source, replace `amazon/aws-iot-greengrass:latest` with the name of your Docker image.

   ```
docker run --rm --init -it --name aws-iot-greengrass \
   --mount type=bind,src=/path/to/greengrass-v2-credentials:/root/.aws:,ro \
   --mount type=bind,src=/path/to/greengrass-v2-credentials:/root/.aws/,ro \
   --env-file .env \
   -p 8883 \
   amazon/aws-iot-greengrass:latest
   ```

   This example command uses the following arguments for `docker run`:
Run AWS IoT Greengrass in Docker with automatic provisioning

- **--rm.** Cleans up the container when it exits.
- **--init.** Uses an init process in the container.

**Note**
The **--init** argument is required to shut down AWS IoT Greengrass Core software when you stop the Docker container.

- **-it.** (Optional) Runs the Docker container in the foreground as an interactive process. You can replace this with the **--d** argument to run the Docker container in detached mode instead. For more information, see [Detached vs foreground](https://docs.docker.com/engine/tutorials/run/) in the Docker documentation.

- **--name.** Runs a container named **aws-iot-greengrass**

- **-v.** Mounts a volume into the Docker container to make the configuration file and the certificate files available to AWS IoT Greengrass running inside the container.

- **--env-file.** (Optional) Specifies the environment file to set the environment variables that will be passed to the AWS IoT Greengrass Core software installer inside the Docker container. This argument is required only if you created an environment file (p. 113) to set environment variables. If you didn't create an environment file, you can use **--env** arguments to set environment variables directly in your Docker run command.

- **-p.** (Optional) Publishes the 8883 container port to the host machine. This argument is required if you want to connect and communicate over MQTT because AWS IoT Greengrass uses port 8883 for MQTT traffic. To open other ports, use additional **--p** arguments.

**Note**
To run your Docker container with increased security, you can use the **--cap-drop** and **--cap-add** arguments to selectively enable Linux capabilities for your container. For more information, see [Runtime privilege and Linux capabilities](https://docs.docker.com/engine/reference/run/#linux-capabilities) in the Docker documentation.

3. Remove the credential file from **path/to/greengrass-v2-credentials**.

**Docker Compose**

1. Use a text editor to create a Docker Compose file named **docker-compose.yml**.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the **docker-compose.yml** in the current directory.

   ```bash
   nano docker-compose.yml
   ```

   **Note**
   You can also download and use the latest version of the AWS-provided Compose file from [GitHub](https://github.com/aws-iot-greengrass/dockerscripts).

2. Add the following content to the Compose file. Your file should look similar to the following example. This example specifies the Greengrass Docker image that you downloaded from Docker Hub. If you use a Docker image from a different source, replace **amazon/aws-iot-greengrass:latest** with the name of your Docker image.

   ```yaml
   version: '3.7'
   services:
     greengrass:
       init: true
       build:
         context: .
         container_name: aws-iot-greengrass
       image: amazon/aws-iot-greengrass:latest
   ```
Run AWS IoT Greengrass in Docker with automatic provisioning

```yaml
volumes:
  - ./greengrass-v2-credentials:/root/.aws/:ro
env_file: .env
ports:
  - "8883:8883"
```

The following parameters in this example Compose file are optional:

- **ports**—Publishes the 8883 container ports to the host machine. This parameter is required if you want to connect and communicate over MQTT because AWS IoT Greengrass uses port 8883 for MQTT traffic.
- **env_file**—Specifies the environment file to set the environment variables that will be passed to the AWS IoT Greengrass Core software installer inside the Docker container. This parameter is required only if you created an environment file (p. 113) to set environment variables. If you didn't create an environment file, you can use the `environment` parameter to set the variables directly in your Compose file.

**Note**
To run your Docker container with increased security, you can use `cap_drop` and `cap_add` in your Compose file to selectively enable Linux capabilities for your container. For more information, see Runtime privilege and Linux capabilities in the Docker documentation.

3. Run the following command start the Docker container.

```bash
docker-compose -f docker-compose.yml up
```

4. Remove the credential file from `./greengrass-v2-credentials`.

**Next steps**

AWS IoT Greengrass Core software is now running in a Docker container. Run the following command to retrieve the container ID for the currently running container.

```bash
docker ps
```

You can then run the following command to access the container and explore AWS IoT Greengrass Core software running inside the container.

```bash
docker exec -it container-id /bin/bash
```

For information about creating a simple component, see Create your first component (p. 30) in Getting started with AWS IoT Greengrass V2 (p. 24)

**Note**
When you use `docker exec` to run commands inside the Docker container, those commands are not logged in the Docker logs. To log your commands in the Docker logs, attach an interactive shell to the Docker container. For more information, see Attach an interactive shell to the Docker container (p. 119).

The AWS IoT Greengrass Core log file is called `greengrass.log` and is located in `/greengrass/v2/logs`. Component log files are also located in the same directory. To copy Greengrass logs to a temporary directory on the host, run the following command:

```bash
docker cp container-id:/greengrass/v2/logs /tmp/logs
```
If you want to persist logs after a container exits or has been removed, we recommend that you bind-mount only the `/greengrass/v2/logs` directory to the temporary logs directory on the host instead of mounting the entire Greengrass directory. For more information, see Persist Greengrass logs outside of the Docker container (p. 118).

To stop a running AWS IoT Greengrass Docker container, run `docker stop` or `docker-compose -f docker-compose.yml stop`. This action sends `SIGTERM` to the Greengrass process and shuts down all associated processes that were started in the container. The Docker container is initialized with the `docker-init` executable as process PID 1, which helps in removing any leftover zombie processes. For more information, see the Docker documentation.

For information about troubleshooting issues with running AWS IoT Greengrass in a Docker container, see Troubleshooting AWS IoT Greengrass in a Docker container (p. 117).

## Run AWS IoT Greengrass in a Docker container with manual resource provisioning

This tutorial shows you how to install and run AWS IoT Greengrass Core software in Docker container with manually provisioned AWS resources.

### Topics
- Prerequisites (p. 104)
- Create an AWS IoT thing (p. 51)
- Download the Amazon root certification authority (p. 108)
- Retrieve AWS IoT endpoints (p. 54)
- Create a token exchange role (p. 55)
- Create a configuration file (p. 113)
- Create an environment file (p. 113)
- Run the AWS IoT Greengrass Core software in a container (p. 114)
- Next steps (p. 103)

### Prerequisites

To complete this tutorial, you need the following:

- An AWS account. If you don't have one, see Set up an AWS account (p. 44).
- An AWS IoT Greengrass Docker image. This tutorial shows you how to pull the AWS IoT Greengrass Docker image from Docker Hub. You can also pull the AWS IoT Greengrass Docker image (p. 94) from Amazon Elastic Container Registry (Amazon ECR), or you can build an image from the AWS IoT Greengrass Dockerfile (p. 95).
- A Linux-based operating system with an internet connection.
- Docker Engine version 18.09 or later.
- (Optional) Docker Compose version 1.22 or later. Docker Compose is required only if you want to use the Docker Compose CLI to run your Docker images.

### Create an AWS IoT thing

AWS IoT things represent devices and logical entities that connect to AWS IoT. Greengrass core devices are AWS IoT things. When you register a device as an AWS IoT thing, that device can use a digital...
In this section, you create an AWS IoT thing and download certificates that your device can use to connect to AWS.

To create an AWS IoT thing

1. Create an AWS IoT thing for your device. On your development computer, run the following command.

   - Replace `MyGreengrassCore` with the thing name to use. This name is also the name of your Greengrass core device.

   ```
   Note
   The thing name can't contain colon (:) characters.
   ```

   ```bash
   aws iot create-thing --thing-name MyGreengrassCore
   ```

   The response looks similar to the following example, if the request succeeds.

   ```json
   {
   "thingName": "MyGreengrassCore",
   "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
   "thingId": "8cb4b6cd-268e-495d-b5b9-1713d71dbf42"
   }
   ```

2. Create a folder where you download the certificates for the AWS IoT thing.

   ```bash
   mkdir greengrass-v2-certs
   ```

3. Create and download the certificates for the AWS IoT thing.

   ```bash
   aws iot create-keys-and-certificate --set-as-active --certificate-pem-outfile greengrass-v2-certs/device.pem.crt --public-key-outfile greengrass-v2-certs/public.pem.key --private-key-outfile greengrass-v2-certs/private.pem.key
   ```

   The response looks similar to the following example, if the request succeeds.

   ```json
   {
   "certificateArn": "arn:aws:iot:us-west-2:123456789012:cert/aa0b7958770878eabe251d8a7dd547f4889c5249b574ab9fbb65f32248b1d4",
   "certificateId": "aa0b7958770878eabe251d8a7dd547f4889c5249b574ab9fbb65f32248b1d4",
   "certificatePem": "-----BEGIN CERTIFICATE-----MIICCiCCAfICCQD6m7oRw0uXOJANBngkhhkG9w...
   ```

   ```
   EXAMPLE=
   ```

   105
4. Attach the certificate to the AWS IoT thing.

- Replace `MyGreengrassCore` with the name of your AWS IoT thing.
- Replace the certificate Amazon Resource Name (ARN) with the ARN of the certificate that you created in the previous step.

```bash
aws iot attach-thing-principal --thing-name MyGreengrassCore --principal arn:aws:iot:us-west-2:123456789012:cert/aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab5f65f32248b1d4
```

The command doesn't have any output if the request succeeds.

5. Create and attach an AWS IoT policy that defines the AWS IoT permissions for your Greengrass core device. The following policy allows access to all MQTT topics and Greengrass operations, so your device works with custom applications and future changes that require new Greengrass operations. You can restrict this policy down based on your use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices (p. 614).

If you have set up a Greengrass core device before, you can attach its AWS IoT policy instead of creating a new one.

Do the following:

a. Create a file that contains the AWS IoT policy document that Greengrass core devices require.

```bash
nano greengrass-v2-iot-policy.json
```

Copy the following JSON into the file.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Subscribe",
        "iot:Receive",
        "iot:Connect",
        "greengrass:*"
      ],
      "Resource": [
        "*
      ]
    }
  ]
}
```
b. Create an AWS IoT policy from the policy document.

- Replace `GreengrassV2IoTThingPolicy` with the name of the policy to create.

```bash
aws iot create-policy --policy-name GreengrassV2IoTThingPolicy --policy-document file://greengrass-v2-iot-policy.json
```

The response looks similar to the following example, if the request succeeds.

```
{
  "policyName": "GreengrassV2IoTThingPolicy",
```

c. Attach the AWS IoT policy to the AWS IoT thing's certificate.

- Replace `GreengrassV2IoTThingPolicy` with the name of the policy to attach.
- Replace the target ARN with the ARN of the certificate for your AWS IoT thing.

```bash
aws iot attach-policy --policy-name GreengrassV2IoTThingPolicy --target arn:aws:iot:us-west-2:123456789012:cert/aa0b7958770878eabe251d8a7dd547f4889c524c9b574ab9f65f32248b1d4
```

The command doesn't have any output if the request succeeds.

6. (Optional) Add the AWS IoT thing to a new or existing thing group. You use thing groups to manage fleets of Greengrass core devices. When you deploy software components to your devices, you can choose to target individual devices or groups of devices. You can add a device to a thing group with an active Greengrass deployment to deploy that thing group's software components to the device. Do the following:

a. (Optional) Create an AWS IoT thing group.

- Replace `MyGreengrassCoreGroup` with the name of the thing group to create.

  **Note**
  
  The thing group name can't contain colon (:) characters.
Run AWS IoT Greengrass in Docker with manual provisioning

aws iot create-thing-group --thing-group-name MyGreengrassCoreGroup

The response looks similar to the following example, if the request succeeds.

```
{
  "thingGroupName": "MyGreengrassCoreGroup",
  "thingGroupId": "4df721e1-ff9f-4f97-92dd-02db4e3f03aa"
}
```

b. Add the AWS IoT thing to a thing group.

- Replace MyGreengrassCore with the name of your AWS IoT thing.
- Replace MyGreengrassCoreGroup with the name of the thing group.

aws iot add-thing-to-thing-group --thing-name MyGreengrassCore --thing-group-name MyGreengrassCoreGroup

The command doesn't have any output if the request succeeds.

Download the Amazon root certification authority

In the previous step, you downloaded the certificates for your AWS IoT thing. In this step, you download the Amazon root certificate authority (CA) certificate. AWS IoT certificates are associated with Amazon's root CA certificate by default.

Run the following command to download the root CA certificate.

curl ./greengrass-v2-certs/AmazonRootCA1.pem https://www.amazontrust.com/repository/AmazonRootCA1.pem

Retrieve AWS IoT endpoints

Get the AWS IoT endpoints for your AWS account, and save them to use later. Your device uses these endpoints to connect to AWS IoT. Do the following:

1. Get the AWS IoT data endpoint for your AWS account.

aws iot describe-endpoint --endpoint-type iot:Data-ATS

The response looks similar to the following example, if the request succeeds.

```
{
  "endpointAddress": "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
}
```

2. Get the AWS IoT credentials endpoint for your AWS account.

aws iot describe-endpoint --endpoint-type iot:CredentialProvider

The response looks similar to the following example, if the request succeeds.
Create a token exchange role

Greengrass core devices use an IAM service role, called the token exchange role, to authorize calls to AWS services. The device uses the AWS IoT credentials provider to get temporary AWS credentials for this role, which allows the device to interact with AWS IoT, send logs to Amazon CloudWatch Logs, and download custom component artifacts from Amazon S3. For more information, see Authorize core devices to interact with AWS services (p. 627).

You use an AWS IoT role alias to configure the token exchange role for Greengrass core devices. Role aliases enable you to change the token exchange role for a device but keep the device configuration the same. For more information, see Authorizing direct calls to AWS services in the AWS IoT Core Developer Guide.

In this section, you create a token exchange IAM role and an AWS IoT role alias that points to the role. If you have already set up a Greengrass core device, you can use its token exchange role and role alias instead of creating new ones. Then, you configure your device's AWS IoT thing to use that role and alias.

To create a token exchange IAM role

1. Create an IAM role that your device can use as a token exchange role. Do the following:
   a. Create a file that contains the trust policy document that the token exchange role requires.  
      nano device-role-trust-policy.json
      Copy the following JSON into the file.

      ```json
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Principal": {
              "Service": "credentials.iot.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
          }
        ]
      }
      ```

   b. Create the token exchange role with the trust policy document.
      • Replace `GreengrassV2TokenExchangeRole` with the name of the IAM role to create.

      ```bash
      aws iam create-role --role-name GreengrassV2TokenExchangeRole --assume-role-policy-document file://device-role-trust-policy.json
      ```

      The response looks similar to the following example, if the request succeeds.

      ```json
      ```
Run AWS IoT Greengrass in Docker with manual provisioning

```
"Role": {
  "Path": "/",
  "RoleName": "GreengrassV2TokenExchangeRole",
  "RoleId": "AROAZ2YMUHYHK5OKM77FB",
  "Arn": "arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole",
  "CreateDate": "2021-02-06T00:13:29+00:00",
  "AssumeRolePolicyDocument": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Principal": {
          "Service": "credentials.iot.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
      }
    ]
  }
}
```

c. Create a file that contains the access policy document that the token exchange role requires.

```
nano device-role-access-policy.json
```

Copy the following JSON into the file.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:DescribeCertificate",
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
}
```

**Note**

This access policy doesn't allow access to component artifacts in S3 buckets. To deploy custom components that define artifacts in Amazon S3, you must add permissions to the role to allow your core device to retrieve component artifacts. For more information, see Allow access to S3 buckets for component artifacts (p. 628). If you don't yet have an S3 bucket for component artifacts, you can add these permissions later after you create a bucket.

d. Create the IAM policy from the policy document.

- Replace `GreengrassV2TokenExchangeRoleAccess` with the name of the IAM policy to create.

```
aws iam create-policy --policy-name GreengrassV2TokenExchangeRoleAccess --policy-document file://device-role-access-policy.json
```
The response looks similar to the following example, if the request succeeds.

```
{
  "Policy": {
    "PolicyName": "GreengrassV2TokenExchangeRoleAccess",
    "PolicyId": "ANPAZ2YMUHHAC17C5266",
    "Path": "/",
    "DefaultVersionId": "v1",
    "AttachmentCount": 0,
    "PermissionsBoundaryUsageCount": 0,
    "IsAttachable": true,
    "CreateDate": "2021-02-06T00:37:17+00:00",
    "UpdateDate": "2021-02-06T00:37:17+00:00"
  }
}
```

e. Attach the IAM policy to the token exchange role.

- Replace `GreengrassV2TokenExchangeRole` with the name of the IAM role.
- Replace the policy ARN with the ARN of the IAM policy that you created in the previous step.

```
```

The command doesn't have any output if the request succeeds.

2. Create an AWS IoT role alias that points to the token exchange role.

- Replace `GreengrassCoreTokenExchangeRoleAlias` with the name of the role alias to create.
- Replace the role ARN with the ARN of the IAM role that you created in the previous step.

```
aws iot create-role-alias --role-alias GreengrassCoreTokenExchangeRoleAlias --role-arn arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole
```

The response looks similar to the following example, if the request succeeds.

```
{
  "roleAlias": "GreengrassCoreTokenExchangeRoleAlias",
  "roleAliasArn": "arn:aws:iot:us-west-2:123456789012:rolealias/GreengrassCoreTokenExchangeRoleAlias"
}
```

**Note**

To create a role alias, you must have permission to pass the token exchange IAM role to AWS IoT. If you receive an error message when you try to create a role alias, check that your AWS user has this permission. For more information, see Granting a user permissions to pass a role to an AWS service in the AWS Identity and Access Management User Guide.

3. Create and attach an AWS IoT policy that allows your Greengrass core device to use the role alias to assume the token exchange role. If you have set up a Greengrass core device before, you can attach its role alias AWS IoT policy instead of creating a new one. Do the following:

a. (Optional) Create a file that contains the AWS IoT policy document that the role alias requires.

```
nano greengrass-v2-iot-role-alias-policy.json
```
Copy the following JSON into the file.

- Replace the resource ARN with the ARN of your role alias.

```json
{
  "Version":"2012-10-17",
  "Statement": [ 
    { 
      "Effect": "Allow",
      "Action": "iot:AssumeRoleWithCertificate",
    }
  ]
}
```

b. Create an AWS IoT policy from the policy document.

- Replace `GreengrassCoreTokenExchangeRoleAliasPolicy` with the name of the AWS IoT policy to create.

```bash
aws iot create-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy --policy-document file://greengrass-v2-iot-role-alias-policy.json
```

The response looks similar to the following example, if the request succeeds.

```json
{
  "policyName": "GreengrassCoreTokenExchangeRoleAliasPolicy",
  "policyDocument": "{
    "Version":"2012-10-17",
    "Statement": [ 
      { 
        "Effect": "Allow",
        "Action": "iot:AssumeRoleWithCertificate",
      }
    ]
  },
  "policyVersionId": "1"
}
```

c. Attach the AWS IoT policy to the AWS IoT thing's certificate.

- Replace `GreengrassCoreTokenExchangeRoleAliasPolicy` with the name of the role alias AWS IoT policy.
- Replace the target ARN with the ARN of the certificate for your AWS IoT thing.

```bash
aws iot attach-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy --target arn:aws:iot:us-west-2:123456789012:cert/aa0b79s870878eabe251d8a7ddd547f4889c524c9b574ab9f665f32248b1d4
```

The command doesn't have any output if the request succeeds.
Create a configuration file

1. Create a folder where you place your configuration file.

```bash
mkdir ./greengrass-v2-config
```

2. Use a text editor to create a configuration file named `config.yaml` in the `./greengrass-v2-config` folder.

   For example, you can run the following command to use GNU nano to create the `config.yaml`.

   ```bash
   nano ./greengrass-v2-config/config.yaml
   ```

3. Copy the following YAML content into the file. This partial configuration file specifies system parameters and Greengrass nucleus parameters.

   ```yaml
   ---
   system:
     certificateFilePath: "./tmp/certs/device.pem.crt"
     privateKeyPath: "./tmp/certs/private.pem.key"
     rootCaPath: "./tmp/certs/AmazonRootCA1.pem"
     rootpath: "./greengrass/v2"
     thingName: "MyGreengrassCore"
   services:
     aws.greengrass.Nucleus:
       componentType: "NUCLEUS"
       version: "nucleus-version"
       configuration:
         awsRegion: "region"
         iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
         iotDataEndpoint: "device-data-prefix-ats.iot.region.amazonaws.com"
         iotCredentialEndpoint: "device-credentials-prefix.credentials.region.amazonaws.com"
   ```

   Then, replace the following values:
   - `/tmp/certs`. The directory in the Docker container to which you mount the downloaded certificates when you start the container.
   - `/greengrass/v2`. The Greengrass root folder that you want to use for installation. You use the `GCC_ROOT` environment variable to set this value.
   - `MyGreengrassCore`. The name of the AWS IoT thing.
   - `nucleus-version`. The version of the AWS IoT Greengrass Core software to install. This value must match the version of the Docker image or Dockerfile that you downloaded. If you downloaded the Greengrass Docker image with the latest tag, use `docker inspect image-id` to see the image version.
   - `region`. The AWS Region where you created your AWS IoT resources. You must also specify the same value for the `AWS_REGION` environment variable in your environment file (p. 113).
   - `GreengrassCoreTokenExchangeRoleAlias`. The token exchange role alias.
   - `device-data-prefix`. The prefix for your AWS IoT data endpoint.
   - `device-credentials-prefix`. The prefix for your AWS IoT credentials endpoint.

Create an environment file

This tutorial uses an environment file to set the environment variables that will be passed to the AWS IoT Greengrass Core software installer inside the Docker container. You can also use the `-e` or `--env`
Run AWS IoT Greengrass with manual provisioning

argument in your docker run command to set environment variables in the Docker container or you can set the variables in an environment block in the docker-compose.yml file.

1. Use a text editor to create an environment file named .env.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the .env in the current directory.

   ```
   nano .env
   ```

2. Copy the following content into the file.

   ```
   GGC_ROOT_PATH=/greengrass/v2
   AWS_REGION=region
   PROVISION=false
   COMPONENT_DEFAULT_USER=ggc_user:ggc_group
   DEPLOY_DEV_TOOLS=true
   INIT_CONFIG=/tmp/config/config.yaml
   ```

   Then, replace the following values.

   - /greengrass/v2. The path to the root folder to use to install the AWS IoT Greengrass Core software.
   - region. The AWS Region where you created your AWS IoT resources. You must specify the same value for the awsRegion configuration parameter in your configuration file (p. 113).
   - /tmp/config/. The directory to which you mount the configuration file when you start the Docker container.

Run the AWS IoT Greengrass Core software in a container

This tutorial shows you how to pull the latest AWS IoT Greengrass Docker image from Docker Hub and start the Docker container. You can use the Docker CLI or the Docker Compose CLI to run the AWS IoT Greengrass Core software image in a Docker container.

Docker

1. Run the following command to pull the latest AWS IoT Greengrass Docker image from Docker Hub.

   ```
   docker pull amazon/aws-iot-greengrass:latest
   ```

2. Run the following command to start the Docker container. This command runs the Greengrass Docker image that you downloaded from Docker Hub. If you use a Docker image from a different source, replace amazon/aws-iot-greengrass:latest with the name of your Docker image.

   ```
   docker run --rm --init -it --name aws-iot-greengrass \
   --env-file .env \
   -v path/to/greengrass-v2-config:/tmp/config:ro \
   -v path/to/greengrass-v2-certs:/tmp/certs:ro \
   -p 8883 \
   amazon/aws-iot-greengrass:latest
   ```

   This example command uses the following arguments for docker run:

   - --rm. Cleans up the container when it exits.
Run AWS IoT Greengrass in Docker with manual provisioning

- **--init**. Uses an init process in the container.
  
  **Note**
  The `--init` argument is required to shut down AWS IoT Greengrass Core software when you stop the Docker container.

- **--it**. (Optional) Runs the Docker container in the foreground as an interactive process. You can replace this with the `-d` argument to run the Docker container in detached mode instead. For more information, see Detached vs foreground in the Docker documentation.

- **--name**. Runs a container named `aws-iot-greengrass`

- **-v**. Mounts a volume into the Docker container to make the configuration file and the certificate files available to AWS IoT Greengrass running inside the container.

- **--env-file**. (Optional) Specifies the environment file to set the environment variables that will be passed to the AWS IoT Greengrass Core software installer inside the Docker container. This argument is required only if you created an environment file (p. 113) to set environment variables. If you didn’t create an environment file, you can use `--env` arguments to set environment variables directly in your Docker run command.

- **-p**. (Optional) Publishes the 8883 container port to the host machine. This argument is required if you want to connect and communicate over MQTT because AWS IoT Greengrass uses port 8883 for MQTT traffic. To open other ports, use additional `-p` arguments.

  **Note**
  To run your Docker container with increased security, you can use the `--cap-drop` and `--cap-add` arguments to selectively enable Linux capabilities for your container. For more information, see Runtime privilege and Linux capabilities in the Docker documentation.

Docker Compose

1. Use a text editor to create a Docker Compose file named `docker-compose.yml`.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the `docker-compose.yml` in the current directory.

   ```bash
nano docker-compose.yml
   ```

   **Note**
   You can also download and use the latest version of the AWS-provided Compose file from [GitHub](https://github.com).

2. Add the following content to the Compose file. Your file should look similar to the following example. This example specifies the Greengrass Docker image that you downloaded from Docker Hub. If you use a Docker image from a different source, replace `amazon/aws-iot-greengrass:latest` with the name of your Docker image.

   ```yaml
   version: '3.7'
   services:
     greengrass:
       init: true
       build:
         context: .
         container_name: aws-iot-greengrass
         image: amazon/aws-iot-greengrass:latest
         volumes:
           - path/to/greengrass-v2-config:/tmp/config:ro
           - path/to/greengrass-v2-certs:/tmp/certs:ro
   ```
The following parameters in this example Compose file are optional:

- **ports**—Publishes the 8883 container ports to the host machine. This parameter is required if you want to connect and communicate over MQTT because AWS IoT Greengrass uses port 8883 for MQTT traffic.

- **env_file**—Specifies the environment file to set the environment variables that will be passed to the AWS IoT Greengrass Core software installer inside the Docker container. This parameter is required only if you created an environment file (p. 113) to set environment variables. If you didn't create an environment file, you can use the `environment` parameter to set the variables directly in your Compose file.

**Note**

To run your Docker container with increased security, you can use `cap_drop` and `cap_add` in your Compose file to selectively enable Linux capabilities for your container. For more information, see Runtime privilege and Linux capabilities in the Docker documentation.

3. Run the following command to start the container.

```bash
docker-compose -f docker-compose.yml up
```

**Next steps**

AWS IoT Greengrass Core software is now running in a Docker container. Run the following command to retrieve the container ID for the currently running container.

```bash
docker ps
```

You can then run the following command to access the container and explore AWS IoT Greengrass Core software running inside the container.

```bash
docker exec -it container-id /bin/bash
```

For information about creating a simple component, see Create your first component (p. 30) in Getting started with AWS IoT Greengrass V2 (p. 24)

**Note**

When you use `docker exec` to run commands inside the Docker container, those commands are not logged in the Docker logs. To log your commands in the Docker logs, attach an interactive shell to the Docker container. For more information, see Attach an interactive shell to the Docker container (p. 119).

The AWS IoT Greengrass Core log file is called `greengrass.log` and is located in `/greengrass/v2/logs`. Component log files are also located in the same directory. To copy Greengrass logs to a temporary directory on the host, run the following command:

```bash
docker cp container-id:/greengrass/v2/logs /tmp/logs
```

If you want to persist logs after a container exits or has been removed, we recommend that you bind-mount only the `/greengrass/v2/logs` directory to the temporary logs directory on the host instead
of mounting the entire Greengrass directory. For more information, see Persist Greengrass logs outside of the Docker container (p. 118).

To stop a running AWS IoT Greengrass Docker container, run `docker stop` or `docker-compose -f docker-compose.yml stop`. This action sends `SIGTERM` to the Greengrass process and shuts down all associated processes that were started in the container. The Docker container is initialized with the `docker-init` executable as process PID 1, which helps in removing any leftover zombie processes. For more information, see the Docker documentation.

For information about troubleshooting issues with running AWS IoT Greengrass in a Docker container, see Troubleshooting AWS IoT Greengrass in a Docker container (p. 117).

Troubleshooting AWS IoT Greengrass in a Docker container

Use the following information to help you troubleshoot issues with running AWS IoT Greengrass in a Docker container and to debug issues with AWS IoT Greengrass in the Docker container.

Topics
- Troubleshooting issues with running the Docker container (p. 117)
- Debugging AWS IoT Greengrass in a Docker container (p. 118)

Troubleshooting issues with running the Docker container

Use the following information to help troubleshoot issues with running AWS IoT Greengrass in a Docker container.

Topics
- Error: Cannot perform an interactive login from a non TTY device (p. 117)
- Error: Unknown options: -no-include-email (p. 117)
- Error: A firewall is blocking file Sharing between windows and the containers. (p. 117)
- Error: You have reached your pull rate limit (p. 118)

Error: Cannot perform an interactive login from a non TTY device

This error can occur when you run the `aws ecr get-login-password` command. Make sure that you installed the latest AWS CLI version 2 or version 1. We recommend that you use the AWS CLI version 2. For more information, see Installing the AWS CLI in the AWS Command Line Interface User Guide.

Error: Unknown options: -no-include-email

This error can occur when you run the `aws ecr get-login` command. Make sure that you have the latest AWS CLI version installed (for example, run: `pip install awscli --upgrade --user`). For more information, see Installing the AWS Command Line Interface on Microsoft Windows in the AWS Command Line Interface User Guide.

Error: A firewall is blocking file Sharing between windows and the containers.

You might receive this error or a Firewall Detected message when running Docker on a Windows computer. This can also occur if you are signed in on a virtual private network (VPN) and your network
settings are preventing the shared drive from being mounted. In that situation, turn off VPN and re-run the Docker container.


You might receive this error when running the `aws ecr get-login-password` command if you don’t have sufficient permissions to access an Amazon ECR repository. For more information, see Amazon ECR Repository Policy Examples and Accessing One Amazon ECR Repository in the Amazon ECR User Guide.

**Error: You have reached your pull rate limit**

Docker Hub limits the number of pull requests that anonymous and Free Docker Hub users can make. If you exceed the rate limits for anonymous or free user pull requests, then you receive one of the following errors:

```
ERROR: toomanyrequests: Too Many Requests.
```

```
You have reached your pull rate limit.
```

To resolve these errors, you can wait for a few hours before you try another pull request. If you plan on consistently submitting a large number of pull requests, see the Docker Hub website for information about rate limits, and options for authenticating and upgrading your Docker account.

**Debugging AWS IoT Greengrass in a Docker container**

To debug issues with a Docker container, you can persist the Greengrass runtime logs or attach an interactive shell to the Docker container.

**Persist Greengrass logs outside of the Docker container**

After you stop a AWS IoT Greengrass container, you can use the following `docker cp` command to copy the Greengrass logs from the Docker container to a temporary logs directory.

```
docker cp container-id:/greengrass/v2/logs /tmp/logs
```

To persist logs even after a container exits or is removed, you must run the AWS IoT Greengrass Docker container after bind-mounting the `/greengrass/v2/logs` directory.

To bind-mount the `/greengrass/v2/logs` directory, do one of the following when you run a new AWS IoT Greengrass Docker container.

- Include `–v /tmp/logs:/greengrass/v2/logs:ro` in your `docker run` command.

  Modify the `volumes` block in the Compose file to include the following line before you run your `docker-compose up` command.

  ```yaml
  volumes:
  - /tmp/logs:/greengrass/v2/logs:ro
  ```

  You can then check your logs at `/tmp/logs` on your host to see Greengrass logs while AWS IoT Greengrass is running inside the Docker container.
Attach an interactive shell to the Docker container

When you use `docker exec` to run commands inside the Docker container, those commands are not captured in the Docker logs. Logging your commands in the Docker logs can help you investigate the state of the Greengrass Docker container. Do one of the following:

- Run the following command in a separate terminal to attach your terminal's standard input, output, and error to the running container. This enables you to view and control the Docker container from your current terminal.

  ```bash
docker attach container-id
  ```

- Run the following command in a separate terminal. This enables you to run your commands in interactive mode, even if the container is not attached.

  ```bash
docker exec -it container-id sh -c "command > /proc/1/fd/1"
  ```

For general AWS IoT Greengrass troubleshooting, see Troubleshooting (p. 751).

## Configure the AWS IoT Greengrass Core software

The AWS IoT Greengrass Core software provides options that you can use to configure the software. You can create deployments to configure the AWS IoT Greengrass Core software on each core device.

**Topics**

- Deploy the Greengrass core nucleus component (p. 119)
- Configure AWS IoT Greengrass as a system service (p. 119)
- Control memory allocation with JVM options (p. 121)
- Configure the user and group that run components (p. 122)
- Configure system resource limits for components (p. 124)
- Connect on port 443 or through a network proxy (p. 125)
- Configure MQTT timeouts and cache settings (p. 128)

### Deploy the Greengrass core nucleus component

AWS IoT Greengrass provides the AWS IoT Greengrass Core software as a component that you can deploy to your Greengrass core devices. You can create a deployment to apply the same configuration to multiple Greengrass core devices. For more information, see Greengrass nucleus (p. 136) and Update the AWS IoT Greengrass Core software (OTA) (p. 129).

### Configure AWS IoT Greengrass as a system service

You must configure the AWS IoT Greengrass Core software as a system service in your device's init system to do the following:

- Start the AWS IoT Greengrass Core software when the device boots. This is a good practice if you manage large fleets of devices.
Configure AWS IoT Greengrass as a system service

- Install and run plugin components. Several AWS-provided components are plugin components, which enables them to interface directly with the Greengrass nucleus. For more information about component types, see Component types (p. 323).
- Apply over-the-air (OTA) updates to the core device's AWS IoT Greengrass Core software. For more information, see Update the AWS IoT Greengrass Core software (OTA) (p. 129).
- Enable components to restart the AWS IoT Greengrass Core software when a deployment updates the component to a new version or updates certain configuration parameters. For more information, see the bootstrap lifecycle step (p. 346).

There are different init systems, such as initd, systemd, and SystemV. AWS IoT Greengrass provides a built-in option to configure the AWS IoT Greengrass Core software as a system service on a device with systemd. Use the --setup-system-service true argument when you install the AWS IoT Greengrass Core software to start the nucleus as a system service and configure it to launch when the device boots.

You can also manually configure the nucleus to run as a system service. The following example is a service file for systemd.

```
[Unit]
Description=Greengrass Core

[Service]
Type=simple
PIDFile=/greengrass/v2/alts/loader.pid
RemainAfterExit=no
Restart=on-failure
RestartSec=10
ExecStart=/bin/sh /greengrass/v2/alts/current/distro/bin/loader

[Install]
WantedBy=multi-user.target
```

For information about how to create and enable a service file for systemd on a Raspberry Pi, see SYSTEMD in the Raspberry Pi documentation.

After you configure the system service, you can use the following commands to configure starting the device on boot and to start or stop the AWS IoT Greengrass Core software.

**To check the status of the service (systemd)**

- Run the following command to check the status of the system service.
  ```bash
  sudo systemctl status greengrass.service
  ```

**To enable service start on device boot (systemd)**

- Run the following command to enable the nucleus to start when the device boots.
  ```bash
  sudo systemctl enable greengrass.service
  ```

**To disable service start on device boot (systemd)**

- Run the following command to stop the nucleus from starting when the device boots.
  ```bash
  sudo systemctl disable greengrass.service
  ```
To start the nucleus as a system service (systemd)

- Run the following command to start the AWS IoT Greengrass Core software.

  ```bash
  sudo systemctl start greengrass.service
  ```

To stop the nucleus as a system service (systemd)

- Run the following command to stop the AWS IoT Greengrass Core software.

  ```bash
  sudo systemctl stop greengrass.service
  ```

Control memory allocation with JVM options

If you’re running AWS IoT Greengrass on a device with limited memory, you can use Java virtual machine (JVM) options to control the maximum heap size, garbage collection modes, and compiler options, which control the amount of memory that AWS IoT Greengrass Core software uses. The heap size in the JVM determines how much memory an application can use before garbage collection occurs, or before the application runs out of memory. The maximum heap size specifies the maximum amount of memory the JVM can allocate when expanding the heap during heavy activity.

To control memory allocation, create a new deployment or revise an existing deployment that includes the nucleus component, and specify your JVM options in the `jvmOptions` configuration parameter in the nucleus component configuration (p. 139).

Depending on your requirements, you can run AWS IoT Greengrass Core software with reduced memory allocation or with minimum memory allocation.

Reduced memory allocation

To run AWS IoT Greengrass Core software with reduced memory allocation, we recommend that you use the following example configuration merge update to set JVM options in your nucleus configuration:

```json
{
  "jvmOptions": "-Xmx64m -XX:+UseSerialGC -XX:TieredStopAtLevel=1"
}
```

Minimum memory allocation

To run AWS IoT Greengrass Core software with minimum memory allocation, we recommend that you use the following example configuration software merge update to set JVM options in your nucleus configuration:

```json
{
  "jvmOptions": "-Xmx32m -XX:+UseSerialGC -Xint"
}
```

These example configuration merge updates use the following JVM options:

- `XmxNNm`

  Sets the maximum JVM heap size.

  For reduced memory allocation, use `-Xmx64m` as a starting value to limit the heap size to 64 MB. For minimum memory allocation, use `-Xmx32m` as a starting value to limit the heap size to 32 MB.
You can increase or decrease the `-Xmx` value depending on your actual requirements; however, we strongly recommend that you don't set the maximum heap size below 16 MB. If the maximum heap size is too low for your environment, then the AWS IoT Greengrass Core software might encounter unexpected errors because of insufficient memory.

`-XX:+UseSerialGC`

Specifies to use serial garbage collection for JVM heap space. The serial garbage collector is slower, but uses less memory than other JVM garbage collection implementations.

`-XX:TieredStopAtLevel=1`

Instructs the JVM to use the Java just-in-time (JIT) compiler once. Because JIT compiled code uses space in the device memory, using the JIT compiler more than once consumes more memory than a single compilation.

`-Xint`

Instructs the JVM not to use the just-in-time (JIT) compiler. Instead, the JVM runs in interpreted-only mode. This mode is slower than running JIT compiled code; however, the compiled code doesn't use any space in memory.

For information about creating configuration merge updates, see Update component configurations (p. 390).

## Configure the user and group that run components

The AWS IoT Greengrass Core software can run component processes as a system user and group different from the one that runs the software. This increases security, because you can run the AWS IoT Greengrass Core software as root without giving root permissions to components that run on the core device.

The following table indicates which types of components the AWS IoT Greengrass Core software can run as a system user and group that you specify. For more information, see Component types (p. 323).

<table>
<thead>
<tr>
<th>Component type</th>
<th>Configure system user/group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nucleus</td>
<td>☝ No</td>
</tr>
<tr>
<td>Plugin</td>
<td>☝ No</td>
</tr>
<tr>
<td>Generic</td>
<td>☑ Yes</td>
</tr>
<tr>
<td>Lambda (non-containerized)</td>
<td>☑ Yes</td>
</tr>
<tr>
<td>Lambda (containerized)</td>
<td>☑ Yes</td>
</tr>
</tbody>
</table>

When you configure the user and group, you specify them separated by a colon (:) in the following format: `user:group`. The group is optional. If you don't specify a group, the AWS IoT Greengrass Core software defaults to the primary group of the user. You can use name or ID to identify the user and group.

You can configure the user and group for each component and for each core device.

- **Configure for a component**

  You can configure each component to run with a user and group specific to that component. When you create a deployment, you can specify the user and group for each component in the deployment.
The AWS IoT Greengrass Core software runs components as this user and group if you specify them. Otherwise, it defaults to run components as the default user and group that you configure for the core device. For more information, see Create deployments (p. 384).

- **Configure defaults for a core device**

You can configure a default system user and group that the AWS IoT Greengrass Core software uses to run components. When the AWS IoT Greengrass Core software runs a component, it uses the user and group that you specify for that component. If that component doesn't specify a user and group, then the AWS IoT Greengrass Core software runs the component as the default user and group that you configure for the core device. For more information, see Configure the default user and group (p. 123).

**Note**

If you don't configure a user to run components and you run the AWS IoT Greengrass Core software as root, then the software won't run components. You must specify a default user to run components if you run as root.

If you don't configure a user to run components and you run the AWS IoT Greengrass Core software as a non-root user, then the software runs components as that user.

You can also run components as a system user that doesn't exist, also called an unknown user, to increase security. On Linux operating systems, a process can signal any other process that is run by the same user. An unknown user doesn't run other processes, so you can run components as an unknown user to prevent components from signaling other components on the core device. To run components as an unknown user, specify a user ID that doesn't exist on the core device. You can also specify a group ID that doesn't exist to run as an unknown group.

**Configure the default user and group**

You can use a deployment to configure the default user and group on a core device. In this deployment, you update the nucleus component (p. 136) configuration.

**Note**

You can also set the default user and group when you run the AWS IoT Greengrass Core software with the `--component-default-user` option. For more information, see Install the AWS IoT Greengrass Core software (p. 45).

To configure the default user and group, create a deployment (p. 384) that specifies the following configuration update for the aws.greengrass.Nucleus component.

```json
{
  "runWithDefault": {
    "posixUser": "ggc_user:ggc_group"
  }
}
```

The following example defines a deployment that configures `ggc_user` as the default user and `ggc_group` as the default group. The `merge` configuration update requires a serialized JSON object.

```json
{
  "components": {
    "aws.greengrass.Nucleus": {
      "version": "2.4.0",
      "configurationUpdate": {
        "merge": "{\"runWithDefault\":{"\nposixUser\":\"ggc_user:ggc_group\"}}"
      }
    }
  }
}
```
Configure system resource limits for components

This feature is available for v2.4.0 and later of the Greengrass nucleus component (p. 136).

You can configure the maximum amount of CPU and RAM usage that each component’s processes can use on the core device.

The following table shows the types of components that support system resource limits. For more information, see Component types (p. 323).

<table>
<thead>
<tr>
<th>Component type</th>
<th>Configure system resource limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nucleus</td>
<td>☒ No</td>
</tr>
<tr>
<td>Plugin</td>
<td>☒ No</td>
</tr>
<tr>
<td>Generic</td>
<td>☑ Yes</td>
</tr>
<tr>
<td>Lambda (non-containerized)</td>
<td>☑ Yes</td>
</tr>
<tr>
<td>Lambda (containerized)</td>
<td>☒ No</td>
</tr>
</tbody>
</table>

Important
System resource limits aren’t supported when you run AWS IoT Greengrass Core software in a Docker container (p. 93).

You can configure system resource limits for each component and for each core device.

- **Configure for a component**
  
  You can configure each component with system resource limits specific to that component. When you create a deployment, you can specify the system resource limits for each component in the deployment. If the component supports system resource limits, the AWS IoT Greengrass Core software applies the limits to the component’s processes. If you don’t specify system resource limits for a component, the AWS IoT Greengrass Core software uses any defaults that you have configured for the core device. For more information, see Create deployments (p. 384).

- **Configure defaults for a core device**
  
  You can configure the default system resource limits that the AWS IoT Greengrass Core software applies to components that support these limits. When the AWS IoT Greengrass Core software runs a component, it applies the system resource limits that you specify for that component. If that component doesn’t specify system resource limits, the AWS IoT Greengrass Core software applies the default system resource limits that you configure for the core device. If you don’t specify default system resource limits, the AWS IoT Greengrass Core software doesn’t apply any system resource limits by default. For more information, see Configure default system resource limits (p. 124).

Configure default system resource limits

You can deploy the Greengrass nucleus component (p. 136) to configure the default system resource limits for a core device. To configure the default system resource limits, create a deployment (p. 384) that specifies the following configuration update for the aws.greengrass.Nucleus component.
Connect on port 443 or through a network proxy

AWS IoT Greengrass core devices communicate with AWS IoT Core using the MQTT messaging protocol with TLS client authentication. By convention, MQTT over TLS uses port 8883. However, as a security measure, restrictive environments might limit inbound and outbound traffic to a small range of TCP ports. For example, a corporate firewall might open port 443 for HTTPS traffic, but close other ports that are used for less common protocols, such as port 8883 for MQTT traffic. Other restrictive environments might require all traffic to go through an HTTP proxy before connecting to the internet.

**Note**
Greengrass core devices that run Greengrass nucleus component (p. 136) v2.0.3 and earlier use port 8443 to connect to the AWS IoT Greengrass data plane endpoint. These devices must be able to connect to this endpoint on port 8443. For more information, see Allow device traffic through a proxy or firewall (p. 640).

To enable communication in these scenarios, AWS IoT Greengrass provides the following configuration options:

- **MQTT communication over port 443.** If your network allows connections to port 443, you can configure the Greengrass core device to use port 443 for MQTT traffic instead of the default port 8883. This can be a direct connection to port 443 or a connection through a network proxy server. Unlike the default configuration, which uses certificate-based client authentication, MQTT on port 443 uses the device service role (p. 627) for authentication.

  For more information, see Configure MQTT over port 443 (p. 126).

- **HTTPS communication over port 443.** The AWS IoT Greengrass Core software sends HTTPS traffic over port 8443 by default, but you can configure it to use port 443. AWS IoT Greengrass uses the Application Layer Protocol Network (ALPN) TLS extension to enable this connection. As with the default configuration, HTTPS on port 443 uses certificate-based client authentication.

  **Important**
  To use ALPN and enable HTTPS communication over port 443, your core device must run Java 8 update 252 or later. All updates of Java version 9 and later also support ALPN.

  For more information, see Configure HTTPS over port 443 (p. 126).
• **Connection through a network proxy.** You can configure a network proxy server to act as an intermediary for connecting to the Greengrass core device. AWS IoT Greengrass supports only basic authentication and HTTP proxies. AWS IoT Greengrass doesn’t support HTTPS proxies.

The AWS IoT Greengrass Core software passes the proxy configuration to components through the `ALL_PROXY`, `HTTP_PROXY`, `HTTPS_PROXY`, and `NO_PROXY` environment variables. Components must use these settings to connect through the proxy. Components use common libraries (such as boto3, cURL, and the python `requests` package) that typically use these environment variables by default to make connections. If a component also specifies these environment variables by default to make connections. If a component also specifies these environment variables by default, AWS IoT Greengrass doesn’t override them.

For more information, see Configure a network proxy (p. 127).

### Configure MQTT over port 443

You can use a deployment to configure MQTT over port 443 on a single core device or a group of core devices. In this deployment, you update the nucleus component (p. 136) configuration. The nucleus restarts when you update its `mqtt` configuration.

To configure MQTT over port 443, create a deployment (p. 384) that specifies the following configuration update for the `aws.greengrass.Nucleus` component.

```json
{
  "mqtt": {
    "port": 443
  }
}
```

The following example defines a deployment that configures MQTT over port 443. The `merge` configuration update requires a serialized JSON object.

```json
{
  "components": {
    "aws.greengrass.Nucleus": {
      "version": "2.4.0",
      "configurationUpdate": {
        "merge": "{"mqtt":{"port":443}}"
      }
    }
  }
}
```

### Configure HTTPS over port 443

This feature requires Greengrass nucleus (p. 136) v2.0.4 or later.

You can use a deployment to configure HTTPS over port 443 on a single core device or a group of core devices. In this deployment, you update the nucleus component (p. 136) configuration.

To configure HTTPS over port 443, create a deployment (p. 384) that specifies the following configuration update for the `aws.greengrass.Nucleus` component.

```json
{
  "greengrassDataPlanePort": 443
}
```

The following example defines a deployment that configures HTTPS over port 443. The `merge` configuration update requires a serialized JSON object.
Connect on port 443 or through a network proxy

```json
{
  "components": {
    "aws.greengrass.Nucleus": {
      "version": "2.4.0",
      "configurationUpdate": {
        "merge": "{"greengrassDataPlanePort\":443}"
      }
    }
  }
}
```

Configure a network proxy

Follow the procedure in this section to configure Greengrass core devices to connect to the internet through an HTTP network proxy. For more information about the endpoints and ports that core devices use, see Allow device traffic through a proxy or firewall (p. 640).

**Important**

If your core device runs a version of the Greengrass nucleus (p. 136) earlier than v2.4.0, your device's role must allow the following permissions to use a network proxy:

- `iot:Connect`
- `iot:Publish`
- `iot:Receive`
- `iot:Subscribe`

This is necessary because the device uses AWS credentials from the token exchange service to authenticate MQTT connections to AWS IoT. The device uses MQTT to receive and install deployments from the AWS Cloud, so your device won't work unless you define these permissions on its role. Devices typically use X.509 certificates to authenticate MQTT connections, but devices can't do this to authenticate when they use a proxy.

For more information about how to configure the device role, see Authorize core devices to interact with AWS services (p. 627).

You can use a deployment to configure a network proxy on a single core device or a group of core devices. In this deployment, you update the nucleus component (p. 136) configuration. The nucleus restarts when you update its networkProxy configuration.

To configure a network proxy, create a deployment (p. 384) for the `aws.greengrass.Nucleus` component that merges the following configuration update. This configuration update contains the networkProxy object (p. 128).

```json
{
  "networkProxy": {
    "proxy": {
      "url": "https://my-proxy-server:1100",
      "username": "Mary_Major",
      "password": "pass@word1357"
    }
  }
}
```

The following example defines a deployment that configures a network proxy. The merge configuration update requires a serialized JSON object.

```json
{
```

---

127
"components": {  
  "aws.greengrass.Nucleus": {  
    "version": "2.4.0",  
    "configurationUpdate": {  
      "merge": "{\"networkProxy\":{"noProxyAddresses":"http://192.168.0.1,www.example.com","proxy":{"url":"https://my-proxy-server:1100","username":"Mary_Major","password":"pass@word1357"}}}"  
    }  
  }  
}

The networkProxy object

Use the networkProxy object to specify information about the network proxy. This object contains the following information:

noProxyAddresses

(Optional) A comma-separated list of IP addresses or host names that are exempt from the proxy.

proxy

The proxy to which to connect. This object contains the following information:

url

The URL of the proxy server in the format scheme://userinfo@host:port.

- scheme – The scheme, which must be http or https.
- userinfo – (Optional) The user name and password information. If you specify this in the url, the Greengrass core device ignores the username and password fields.
- host – The host name or IP address of the proxy server.
- port – (Optional) The port number. If you don’t specify the port, then the Greengrass core device uses the following default values:
  - http – 80
  - https – 443

username

(Optional) The user name to use to authenticate to the proxy server.

credential

(Optional) The password to use to authenticate to the proxy server.

Configure MQTT timeouts and cache settings

In the AWS IoT Greengrass environment, components can use MQTT to communicate with AWS IoT Core. The AWS IoT Greengrass Core software manages MQTT messages for components. When the core device loses connection to the AWS Cloud, the software caches MQTT messages to retry later when the connection restores. You can configure settings such as message timeouts and the size of the cache. For more information, see the mqtt and mqtt.spooler configuration parameters of the Greengrass nucleus component (p. 136).

AWS IoT Core imposes service quotas on its MQTT message broker. These quotas might apply to messages that you send between core devices and AWS IoT Core. For more information, see AWS IoT Core message broker service quotas in the AWS General Reference.
Update the AWS IoT Greengrass Core software (OTA)

The AWS IoT Greengrass Core software comprises the Greengrass nucleus component and other optional components that you can deploy to your devices to perform over-the-air (OTA) updates of the software. This feature is built in to the AWS IoT Greengrass Core software.

OTA updates make it more efficient to:

- Fix security vulnerabilities.
- Address software stability issues.
- Deploy new or improved features.

Topics

- Requirements (p. 129)
- Considerations for core devices (p. 129)
- Greengrass nucleus update behavior (p. 129)
- Perform an OTA update (p. 131)

Requirements

The following requirements apply to deploy OTA updates of the AWS IoT Greengrass Core software:

- The Greengrass core device must have a connection to the AWS Cloud to receive the deployment.
- The Greengrass core device must be correctly configured and provisioned with certificates and keys for authentication with AWS IoT Core and AWS IoT Greengrass.
- The AWS IoT Greengrass Core software must be set up and running as a system service. OTA updates don’t work if you run the nucleus from the JAR file, Greengrass.jar. For more information, see Configure AWS IoT Greengrass as a system service (p. 119).

Considerations for core devices

Before perform an OTA update, be aware of the impact on the core devices that you update:

- The Greengrass nucleus shuts down.
- All components running on the core device also shut down. If those components write to local resources, they might leave those resources in an incorrect state unless shut down properly. Components can use interprocess communication (p. 396) to tell the nucleus component to defer the update until they clean up the resources that they use.
- While the nucleus component is shut down, the core device loses its connections with the AWS Cloud and local devices.
- Long-lived Lambda functions that run as components lose their dynamic state information and drop all pending work.

Greengrass nucleus update behavior

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all component dependencies for that component. Because of this, new patch versions of AWS-provided...
public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.

When the version of the Greengrass nucleus component changes, the AWS IoT Greengrass Core software—which includes the nucleus and all other components on your device—restarts to apply the changes. Because of the impact on core devices when the nucleus component is updated, you might want to control when a new nucleus patch version is deployed to your devices. To do so, you must directly include the Greengrass nucleus component in your deployment. Directly including a component means that you include a specific version of that component in your deployment configuration and do not rely on component dependencies to deploy that component to your devices. For more information about defining dependencies in your component recipes, see Recipe format (p. 342).

Review the following table to understand the update behavior for the Greengrass nucleus component based on your actions and deployment configurations.

<table>
<thead>
<tr>
<th>Action</th>
<th>Deployment configuration</th>
<th>Nucleus update behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add new devices to a thing group targeted by an existing deployment</td>
<td>The deployment does not directly include Greengrass nucleus. The deployment directly includes at least one AWS-provided component, or includes a custom component that depends on an AWS-provided component or on the Greengrass nucleus.</td>
<td>On new devices, installs the latest patch version of nucleus that meets all component dependency requirements. On existing devices, does not update the installed version of the nucleus.</td>
</tr>
<tr>
<td>Add new devices to a thing group targeted by an existing deployment</td>
<td>The deployment directly includes a specific version of the Greengrass nucleus.</td>
<td>On new devices, installs the specified nucleus version. On existing devices, does not update the installed version of the nucleus.</td>
</tr>
<tr>
<td>Create a new deployment or revise an existing deployment.</td>
<td>The deployment does not directly include Greengrass nucleus. The deployment directly includes at least one AWS-provided component, or includes a custom component that depends on an AWS-provided component or on the Greengrass nucleus.</td>
<td>On all targeted devices, installs the latest patch version of the nucleus that meets all component dependency requirements, including on any new devices that you add to the targeted thing group.</td>
</tr>
<tr>
<td>Create a new deployment or revise an existing deployment.</td>
<td>The deployment directly includes a specific version of the Greengrass nucleus.</td>
<td>On all targeted devices, installs the specified nucleus version, including any new devices that you add to the targeted thing group.</td>
</tr>
</tbody>
</table>
Perform an OTA update

To perform an OTA update, create a deployment (p. 384) that includes the nucleus component (p. 136) and the version to install.

Uninstall the AWS IoT Greengrass Core software

You can uninstall the AWS IoT Greengrass Core software to remove it from a device that you don't want to use as a Greengrass core device. You might also do this to clean up an installation that fails.

To uninstall the AWS IoT Greengrass Core software

1. If you run the software as a system service, you must stop, disable, and remove the service. Run the following commands.

   ```bash
   sudo systemctl stop greengrass.service && sudo systemctl disable greengrass.service
   sudo rm /etc/systemd/system/greengrass.service
   sudo systemctl daemon-reload && sudo systemctl reset-failed
   ```

2. Remove the root folder from the device. Replace `/greengrass/v2` with the path to the root folder.

   ```bash
   sudo rm -rf /greengrass/v2
   ```

3. Delete the core device from the AWS IoT Greengrass service. This step removes the core device's status information from the AWS Cloud. Be sure to complete this step if you plan to reinstall the AWS IoT Greengrass Core software to a core device with the same name.
   - To delete a core device from the AWS IoT Greengrass console, do the following:
     a. Navigate to the AWS IoT Greengrass console.
     b. Choose Core devices.
     c. Choose the core device to delete.
     d. Choose Delete.
     e. In the confirmation modal, choose Delete.
   - To delete a core device with the AWS Command Line Interface, use the `DeleteCoreDevice` operation. Run the following command, and replace `MyGreengrassCore` with the name of the core device.

     ```bash
     aws greengrassv2 delete-core-device --core-device-thing-name MyGreengrassCore
     ```
AWS-provided components

AWS IoT Greengrass provides and maintains prebuilt components that you can deploy to your devices. These components include features (such as stream manager), AWS IoT Greengrass V1 connectors (such as CloudWatch metrics), and local development tools (such as the AWS IoT Greengrass CLI). You can deploy these components (p. 383) to your devices for their standalone functionality, or you can use them as dependencies in your custom Greengrass components (p. 322).

Note
Several AWS-provided components depend on specific minor versions of the Greengrass nucleus. Because of this dependency, you need to update these components when you update the Greengrass nucleus to a new minor version. For information about the specific versions of the nucleus that each component depends on, see the corresponding component topic. For more information about updating the nucleus, see Update the AWS IoT Greengrass Core software (OTA) (p. 129).

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Depends on nucleus</th>
<th>Component type (p. 323)</th>
<th>Open source (p. 761)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>The nucleus of the AWS IoT Greengrass Core software. Use this component to</td>
<td>Yes</td>
<td>Nucleus</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>configure and update the software on your core devices.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client device auth (p. 146)</td>
<td>Enables local IoT devices, called client devices, to connect to the core</td>
<td>Yes</td>
<td>Plugin</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>device.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CloudWatch metrics (p. 153)</td>
<td>Publishes custom metrics to Amazon CloudWatch.</td>
<td>Yes</td>
<td>Lambda</td>
<td>No</td>
</tr>
<tr>
<td>Device Defender (p. 170)</td>
<td>Notifies administrators of changes in the state of the Greengrass core</td>
<td>Yes</td>
<td>Lambda</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>device to identify unusual behavior.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Docker application manager (p. 162)</td>
<td>Enables AWS IoT Greengrass to download Docker images</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
<td>Depends on nucleus</td>
<td>Component type (p. 323)</td>
<td>Open source (p. 761)</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------------</td>
<td>-------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Greengrass CLI (p. 166)</td>
<td>Provides a command-line interface that you can use to create local deployments and interact with the Greengrass core device and its components.</td>
<td>Yes</td>
<td>Plugin</td>
<td>Yes</td>
</tr>
<tr>
<td>IP detector (p. 177)</td>
<td>Reports MQTT broker connectivity information to AWS IoT Greengrass, so client devices can discover how to connect.</td>
<td>Yes</td>
<td>Plugin</td>
<td>Yes</td>
</tr>
<tr>
<td>Kinesis Data Firehose (p. 180)</td>
<td>Publishes data through Amazon Kinesis Data Firehose delivery streams to destinations in the AWS Cloud.</td>
<td>Yes</td>
<td>Lambda</td>
<td>No</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>Handles processes and environment configuration for Lambda functions.</td>
<td>No</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>Lambda manager (p. 190)</td>
<td>Handles interprocess communication and scaling for Lambda functions.</td>
<td>Yes</td>
<td>Plugin</td>
<td>No</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
<td>Depends on nucleus</td>
<td>Component type (p. 323)</td>
<td>Open source (p. 761)</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------</td>
<td>-------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>Provides artifacts for each Lambda runtime.</td>
<td>No</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>Local debug console (p. 199)</td>
<td>Provides a local console that you can use to debug and manage the Greengrass core device and its components.</td>
<td>Yes</td>
<td>Plugin</td>
<td>Yes</td>
</tr>
<tr>
<td>Legacy subscription router (p. 194)</td>
<td>Manages subscriptions for Lambda functions that run on AWS IoT Greengrass V1.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>Log manager (p. 206)</td>
<td>Collects and uploads logs on the Greengrass core device.</td>
<td>Yes</td>
<td>Plugin</td>
<td>Yes</td>
</tr>
<tr>
<td>Machine learning components (p. 216)</td>
<td>Provides machine learning models and sample inference code that you can use to perform machine learning inference on Greengrass core devices.</td>
<td>See Machine learning components (p. 216).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modbus-RTU protocol adapter (p. 264)</td>
<td>Polls information from local Modbus RTU devices.</td>
<td>Yes</td>
<td>Lambda</td>
<td>No</td>
</tr>
<tr>
<td>MQTT bridge (p. 280)</td>
<td>Relays MQTT messages between client devices, local AWS IoT Greengrass publish/subscribe, and AWS IoT Core.</td>
<td>No</td>
<td>Plugin</td>
<td>Yes</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
<td>Depends on nucleus</td>
<td>Component type (p. 323)</td>
<td>Open source (p. 761)</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td>--------------------</td>
<td>------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>MQTT broker (Moquette) (p. 283)</td>
<td>Handles MQTT messages between client devices and the core device.</td>
<td>No</td>
<td>Plugin</td>
<td>Yes</td>
</tr>
<tr>
<td>Secret manager (p. 285)</td>
<td>Deploys secrets from AWS Secrets Manager secrets so that you can securely use credentials, such as passwords, in custom components on the Greengrass core device.</td>
<td>Yes</td>
<td>Plugin</td>
<td>Yes</td>
</tr>
<tr>
<td>Secure tunneling (p. 289)</td>
<td>Enables AWS IoT secure tunneling connections that you can use to establish bidirectional communications with Greengrass core devices that are behind restricted firewalls.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>Shadow manager (p. 294)</td>
<td>Enables interaction with shadows on the core device. It manages shadow document storage and also the synchronization of local shadow states with the AWS IoT Device Shadow service.</td>
<td>Yes</td>
<td>Plugin</td>
<td>Yes</td>
</tr>
<tr>
<td>Amazon SNS (p. 298)</td>
<td>Publishes messages to Amazon SNS topics.</td>
<td>Yes</td>
<td>Lambda</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 1: Component Details

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Depends on nucleus</th>
<th>Component type (p. 323)</th>
<th>Open source (p. 761)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream manager (p. 306)</td>
<td>Streams high-volume data from local sources to the AWS Cloud.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>Provides AWS credentials that you can use to interact with AWS services.</td>
<td>No</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>IoT SiteWise OPC-UA collector (p. 313)</td>
<td>Collects data from OPC-UA servers.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>IoT SiteWise publisher (p. 315)</td>
<td>Publishes data to the AWS Cloud.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>IoT SiteWise processor (p. 317)</td>
<td>Processes data on the Greengrass core devices.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
</tbody>
</table>

Greengrass nucleus

The Greengrass nucleus component (aws.greengrass.Nucleus) is a mandatory component and the minimum requirement to run the AWS IoT Greengrass Core software on a device. You can configure this component to customize and update your AWS IoT Greengrass Core software remotely. Deploy this component to configure settings such as proxy, device role, and AWS IoT thing configuration on your core devices.

Important
When the version of the nucleus component changes, or when you change certain configuration parameters, the AWS IoT Greengrass Core software—which includes the nucleus and all other components on your device—restarts to apply the changes.

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all component dependencies for that component. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly. To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment (p. 384). For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA) (p. 129).

Topics
- Versions (p. 137)
- Requirements (p. 137)
- Dependencies (p. 137)
- Installation (p. 137)
• Configuration (p. 137)
• Local log file (p. 143)
• Changelog (p. 144)

Versions

This component has the following versions:

• 2.4.x
• 2.3.x
• 2.2.x
• 2.1.x
• 2.0.x

Requirements

Devices must meet certain requirements to install and run the Greengrass nucleus and the AWS IoT Greengrass Core software. For more information, see AWS IoT Greengrass Core software requirements (p. 42).

Dependencies

The Greengrass nucleus does not include any component dependencies. However, several AWS-provided components include the nucleus as a dependency. For more information, see AWS-provided components (p. 132).

For more information about component dependencies, see the component recipe reference (p. 343).

Installation

You can download an installer that sets up the Greengrass nucleus component on your device. This installer sets up your device as a Greengrass core device. There are two types of installations that you can perform: a quick installation that creates required AWS resources for you, or a manual installation where you create the AWS resources yourself. For more information, see Install the AWS IoT Greengrass Core software (p. 45).

You can also follow a tutorial to install the Greengrass nucleus and explore Greengrass component development. For more information, see Getting started with AWS IoT Greengrass V2 (p. 24).

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component. Some parameters require that the AWS IoT Greengrass Core software restarts to take effect. For more information about why and how to configure this component, see Configure the AWS IoT Greengrass Core software (p. 119).

`iotRoleAlias`

The AWS IoT role alias that points to a token exchange IAM role. The AWS IoT credentials provider assumes this role to allow the Greengrass core device to interact with AWS services. For more information, see Authorize core devices to interact with AWS services (p. 627).
When you run the AWS IoT Greengrass Core software with the --provision true option, the software provisions a role alias and sets its value in the nucleus component.

**networkProxy**

(Optional) The network proxy to use for all connections. For more information, see Connect on port 443 or through a network proxy (p. 125).

**Important**

When you deploy a change to this configuration parameter, the AWS IoT Greengrass Core software restarts for the change to take effect.

This object contains the following information:

**noProxyAddresses**

(Optional) A comma-separated list of IP addresses or host names that are exempt from the proxy.

**proxy**

The proxy to which to connect. This object contains the following information:

**url**

The URL of the proxy server in the format scheme://userinfo@host:port.

- **scheme** – The scheme, which must be http.
- **userinfo** – (Optional) The user name and password information. If you specify this information in the `url`, the Greengrass core device ignores the `username` and `password` fields.
- **host** – The host name or IP address of the proxy server.
- **port** – (Optional) The port number. If you don't specify the port, then the Greengrass core device uses the following default value:
  - http – 80

**username**

(Optional) The user name that authenticates the proxy server.

**password**

(Optional) The password that authenticates the proxy server.

**mqtt**

(Optional) The MQTT configuration for the Greengrass core device. For more information, see Connect on port 443 or through a network proxy (p. 125).

**Important**

When you deploy a change to this configuration parameter, the AWS IoT Greengrass Core software restarts for the change to take effect.

This object contains the following information:

**port**

(Optional) The port to use for MQTT connections.

Default: 8883

**keepAliveTimeoutMs**

(Optional) The amount of time in milliseconds between each PING message that the client sends to keep the MQTT connection alive.

Default: 60000 (60 seconds)
pingTimeoutMs

(Optional) The amount of time in milliseconds that the client waits to receive a PINGACK message from the server. If the wait exceeds the timeout, the core device closes and reopens the MQTT connection.

Default: 30000 (30 seconds)

maxInFlightPublishes

(Optional) The maximum number of unacknowledged MQTT QoS 1 messages that can be in flight at the same time.

This feature is available for v2.1.0 and later of this component.

Default: 5

Valid range: Maximum value of 100

maxMessageSizeInBytes

(Optional) The maximum size of an MQTT message. If a message exceeds this size, the Greengrass nucleus rejects the message with an error.

This feature is available for v2.1.0 and later of this component.

Default: 131072 (128 KB)

Valid range: Maximum value of 2621440 (2.5 MB)

maxPublishRetry

(Optional) The maximum number of times to retry a message that fails to publish. You can specify -1 to retry unlimited times.

This feature is available for v2.1.0 and later of this component.

Default: 100

spooler

(Optional) The MQTT spooler configuration for the Greengrass core device. This object contains the following information:

maxSizeInBytes

(Optional) The maximum size of the cache where the core device stores unprocessed MQTT messages in memory. If the cache is full, the core device discards the oldest messages to add new messages.

Default: 2621440 (2.5 MB)

keepQos0WhenOffline

(Optional) You can spool MQTT QoS 0 messages that the core device receives while it's offline. If you set this option to true, the core device spools QoS 0 messages that it can't send while it's offline. If you set this option to false, the core device discards these messages. The core device always spools QoS 1 messages unless the spool is full.

Default: false

jvmOptions

(Optional) The JVM options to use to run the AWS IoT Greengrass Core software. For information about recommended JVM options for running AWS IoT Greengrass Core software, see Control memory allocation with JVM options (p. 121).
Important
When you deploy a change to this configuration parameter, the AWS IoT Greengrass Core software restarts for the change to take effect.

**iotDataEndpoint**

The AWS IoT data endpoint for your AWS account.

When you run the AWS IoT Greengrass Core software with the `--provision true` option, the software gets your data and credentials endpoints from AWS IoT and sets them in the nucleus component.

**iotCredEndpoint**

The AWS IoT credentials endpoint for your AWS account.

When you run the AWS IoT Greengrass Core software with the `--provision true` option, the software gets your data and credentials endpoints from AWS IoT and sets them in the nucleus component.

**greengrassDataPlanePort**

This feature is available in v2.0.4 and later of this component.

(Optional) The port to use for data plane connections. For more information, see Connect on port 443 or through a network proxy (p. 125).

Important
You must specify a port where the device can make outbound connections. If you specify a port that is blocked, the device won't be able to connect to AWS IoT Greengrass to receive deployments.

Choose from the following options:

- 443
- 8443

Default: 8443

**awsRegion**

The AWS Region to use.

**runWithDefault**

The system user and group to use to run components.

Important
When you deploy a change to this configuration parameter, the AWS IoT Greengrass Core software restarts for the change to take effect.

This object contains the following information:

**posixUser**

The name or ID of the system user and system group that the core device uses to run generic and Lambda components. Specify the user and group separated by a colon (:), where the group is optional. If you omit the group, the AWS IoT Greengrass Core software defaults to the primary group of the user that you specify. For example, you can specify `ggc_user` or `ggc_user:ggc_group`. For more information, see Configure the user and group that run components (p. 122).

When you run the AWS IoT Greengrass Core software with the `--component-default-user ggc_user:ggc_group` option, the software sets this parameter in the nucleus component.

You can't specify `root` or `0` for the user or the group.
**systemResourceLimits**

This feature is available in v2.4.0 and later of this component.

The system resource limits to apply to generic and non-containerized Lambda component processes by default. You can override system resource limits for individual components when you create a deployment. For more information, see Configure system resource limits for components (p. 124).

This object contains the following information:

**cpus**

The maximum amount of CPU time that each component's processes can use on the core device. A core device's total CPU time is equivalent to the device's number of CPU cores. For example, on a core device with 4 CPU cores, you can set this value to 2 to limit each component's processes to 50 percent usage of each CPU core. On a device with 1 CPU core, you can set this value to 0.25 to limit each component's processes to 25 percent usage of the CPU. If you set this value to a number greater than the number of CPU cores, the AWS IoT Greengrass Core software doesn't limit the components' CPU usage.

**memory**

The maximum amount of RAM (in kilobytes) that each component's processes can use on the core device.

**logging**

(Optional) The logging configuration for the core device. This object contains the following information:

**level**

(Optional) The minimum level of information to upload. Choose from the following log levels, listed here in level order:

- DEBUG
- INFO
- WARN
- ERROR

Default: INFO

**format**

(Optional) The data format of the logs. Choose from the following options:

- TEXT
- JSON

Default: TEXT

**outputType**

(Optional) The output type for logs. Choose from the following options:

- FILE – The AWS IoT Greengrass Core software outputs logs to files in the directory that you specify in `outputDirectory`.
- CONSOLE – The AWS IoT Greengrass Core software prints logs to `stdout`. Choose this option to view logs as the core device prints them.

Default: FILE

**fileSizeKB**

(Optional) The maximum size of each log file (in kilobytes). After a log file reaches exceeds this maximum file size, the AWS IoT Greengrass Core software creates a new log file.
This parameter applies only when you specify FILE for outputType.

Default: 1024

**totalLogsSizeKB**

(Optional) The maximum total size of all log files (in kilobytes). After the total size of all log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes the oldest log files.

This parameter is equivalent to the system logs disk space limit (p. 209) parameter (diskSpaceLimit) of the log manager component (p. 206). If you specify this parameter on both components, the AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total log size.

This parameter applies only when you specify FILE for outputType.

Default: 10240

**outputDirectory**

(Optional) The output directory for log files.

This parameter applies only when you specify FILE for outputType.

Default: /greengrass/v2/logs, where /greengrass/v2 is the AWS IoT Greengrass root folder.

**fleetstatus**

This parameter is available in v2.1.0 and later of this component.

(Optional) The fleet status configuration for the core device.

This object contains the following information:

**periodicStatusPublishIntervalSeconds**

(Optional) The amount of time (in seconds) between which the core device publishes device status to the AWS Cloud.

Minimum: 86400

Default: 86400

**telemetry**

(Optional) The system health telemetry configuration for the core device. For more information about telemetry metrics and how to act on telemetry data, see Gather system health telemetry data from AWS IoT Greengrass core devices (p. 652).

This object contains the following information:

**enabled**

(Optional) You can enable or disable telemetry.

Default: true

**periodicAggregateMetricsIntervalSeconds**

(Optional) The interval (in seconds) over which the core device aggregates metrics.

If you set this value lower than the minimum supported value, the nucleus uses the default value instead.

Minimum: 3600

Default: 3600
periodicPublishMetricsIntervalSeconds

(Optional) The amount of time (in seconds) between which the core device publishes telemetry metrics to the AWS Cloud.

If you set this value lower than the minimum supported value, the nucleus uses the default value instead.

Minimum: 86400
Default: 86400

deploymentPollingFrequencySeconds

(Optional) The period in seconds at which to poll for deployment notifications.

Default: 15

componentStoreMaxSizeBytes

(Optional) The maximum size on disk of the component store, which comprises component recipes and artifacts.

Default: 10000000000 (10 GB)

platformOverride

(Optional) A dictionary of attributes that identify the core device's platform. Use this to define custom platform attributes that component recipes can use to identify the correct lifecycle and artifacts for the component. For example, you might define a hardware capability attribute to deploy only the minimal set of artifacts for a component to run. For more information, see the manifest platform parameter (p. 345) in the component recipe.

You can also use this parameter to override the os and architecture platform attributes of the core device.

Example: Configuration merge update

```
{
  "iotRoleAlias": "GreengrassCoreTokenExchangeRoleAlias",
  "networkProxy": { 
    "proxy": { 
      "url": "http://my-proxy-server:1100",
      "username": "Mary_Major",
      "password": "pass@word1357"
    }
  },
  "mqtt": { 
    "port": 443
  },
  "greengrassDataPlanePort": 443,
  "jvmOptions": "-Xmx64m",
  "runWithDefault": { 
    "posixUser": "ggc_user:ggc_group"
  }
}
```

Local log file

This component uses the following log file.

```
/greengrass/v2/logs/greengrass.log
```
Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.0</td>
<td>New features</td>
</tr>
<tr>
<td></td>
<td>- Adds support for system resource limits. You can configure the maximum amount of CPU and RAM usage that each component's processes can use on the core device. For more information, see Configure system resource limits for components (p. 124).</td>
</tr>
<tr>
<td></td>
<td>- Adds IPC operations to pause and resume components. For more information, see PauseComponent (p. 449) and ResumeComponent (p. 450).</td>
</tr>
<tr>
<td></td>
<td>- Adds support for provisioning plugins. You can specify a JAR file to run during installation to provision required AWS resources for a Greengrass core device. The Greengrass nucleus includes an interface that you can implement to develop custom provisioning plugins. For more information, see Install AWS IoT Greengrass Core software with custom resource provisioning (p. 83).</td>
</tr>
<tr>
<td></td>
<td>- Adds the optional thing-name-policy argument to the AWS IoT Greengrass Core software installer. You can use this option to specify an existing or custom AWS IoT policy when you install the AWS IoT Greengrass Core software with automatic resource provisioning (p. 46).</td>
</tr>
<tr>
<td></td>
<td>Bug fixes and improvements</td>
</tr>
<tr>
<td></td>
<td>- Updates logging configuration on startup. This fixes an issue where the logging configuration wasn't applied on startup.</td>
</tr>
<tr>
<td></td>
<td>- Updates the nucleus loader symlink to point to the component store in the Greengrass root folder during installation. This update enables you to delete the JAR file and other nucleus artifacts that you download when you install the AWS IoT Greengrass Core software.</td>
</tr>
<tr>
<td></td>
<td>- Additional minor fixes and improvements. For more information, see the changelog on GitHub.</td>
</tr>
<tr>
<td>2.3.0</td>
<td>New features</td>
</tr>
<tr>
<td></td>
<td>- Adds support for deployment configuration documents up to 10 MB, up from 7 KB (for deployments that target things) or 31 KB (for deployments that target thing groups).</td>
</tr>
</tbody>
</table>

To use this feature, a core device's AWS IoT policy must allow the greengrass:GetDeploymentConfiguration permission. If you used the AWS IoT Greengrass Core software installer to provision resources (p. 46), your core device's AWS IoT policy allows greengrass:*, which includes this permission. For more information, see Device authentication and authorization for AWS IoT Greengrass (p. 610).
### Version 2.2.0
**New features**
- Adds IPC operations for local shadow management.

**Bug fixes and improvements**
- Reduces the size of the JAR file.
- Reduces memory usage.
- Fixes issues where the log configuration wasn't updated in certain cases.
- Additional minor fixes and improvements. For more information, see the changelog on GitHub.

### Version 2.1.0
**New features**
- Supports downloading Docker images from private repositories in Amazon ECR.
- Adds the following parameters to customize the MQTT configuration on core devices:
  - `maxInflightPublishes` – The maximum number of unacknowledged MQTT QoS 1 messages that can be in flight at the same time.
  - `maxPublishRetry` – The maximum number of times to retry a message that fails to publish.
- Adds the `fleetstatusservice` configuration parameter to configure the interval at which the core device publishes device status to the AWS Cloud.
- Additional minor fixes and improvements. For more information, see the changelog on GitHub.

**Bug fixes and improvements**
- Fixes an issue that caused shadow deployments to be duplicated when the nucleus restarts.
- Fixes an issue that caused the nucleus to crash when it encountered a service load exception.
- Improves component dependency resolution to fail a deployment that includes a circular dependency.
- Fixes an issue that prevented a plugin component from being redeployed if that component had been previously removed from the core device.
- Fix an issue that caused the `HOME` environment variable to be set to the `/greengrass/v2/work` directory for Lambda components or for components that run as root. The `HOME` variable is now correctly set to the home directory for the user that runs the component.
- Additional minor fixes and improvements. For more information, see the changelog on GitHub.

### Version 2.0.5
**Bug fixes and improvements**
- Correctly routes traffic through a configured network proxy when downloading AWS-provided components.
- Use the correct Greengrass data plane endpoint in AWS China Regions.

---

**Version** | **Changes**
--- | ---
2.2.0 | **New features**
- Adds IPC operations for local shadow management.

**Bug fixes and improvements**
- Reduces the size of the JAR file.
- Reduces memory usage.
- Fixes issues where the log configuration wasn't updated in certain cases.
- Additional minor fixes and improvements. For more information, see the changelog on GitHub.

2.1.0 | **New features**
- Supports downloading Docker images from private repositories in Amazon ECR.
- Adds the following parameters to customize the MQTT configuration on core devices:
  - `maxInflightPublishes` – The maximum number of unacknowledged MQTT QoS 1 messages that can be in flight at the same time.
  - `maxPublishRetry` – The maximum number of times to retry a message that fails to publish.
- Adds the `fleetstatusservice` configuration parameter to configure the interval at which the core device publishes device status to the AWS Cloud.
- Additional minor fixes and improvements. For more information, see the changelog on GitHub.

**Bug fixes and improvements**
- Fixes an issue that caused shadow deployments to be duplicated when the nucleus restarts.
- Fixes an issue that caused the nucleus to crash when it encountered a service load exception.
- Improves component dependency resolution to fail a deployment that includes a circular dependency.
- Fixes an issue that prevented a plugin component from being redeployed if that component had been previously removed from the core device.
- Fix an issue that caused the `HOME` environment variable to be set to the `/greengrass/v2/work` directory for Lambda components or for components that run as root. The `HOME` variable is now correctly set to the home directory for the user that runs the component.
- Additional minor fixes and improvements. For more information, see the changelog on GitHub.

2.0.5 | **Bug fixes and improvements**
- Correctly routes traffic through a configured network proxy when downloading AWS-provided components.
- Use the correct Greengrass data plane endpoint in AWS China Regions.
<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.4</td>
<td>New features</td>
</tr>
<tr>
<td></td>
<td>• Enables HTTPS traffic over port 443. You can use the new greengrassDataPlanePort configuration parameter for version 2.0.4 of the nucleus component to configure HTTPS communication to travel over port 443 instead of the default port 8443. For more information, see Configure HTTPS over port 443 (p. 126).</td>
</tr>
<tr>
<td></td>
<td>• Adds the work path recipe variable. You can use this recipe variable to get the path to components' work folders, which you can use to share files between components and their dependencies. For more information, see the work path recipe variable (p. 355).</td>
</tr>
<tr>
<td></td>
<td>Bug fixes and improvements</td>
</tr>
<tr>
<td></td>
<td>• Prevents the creation of the token exchange AWS Identity and Access Management (IAM) role policy if a role policy already exists.</td>
</tr>
<tr>
<td></td>
<td>As a result of this change, the installer now requires the iam:GetPolicy and sts:GetCallerIdentity when run with --provision true. For more information, see Minimal IAM policy for installer to provision resources (p. 629).</td>
</tr>
<tr>
<td></td>
<td>• Correctly handles the cancellation of a deployment that has not yet been registered successfully.</td>
</tr>
<tr>
<td></td>
<td>• Updates the configuration to remove older entries with newer timestamps when rolling back a deployment.</td>
</tr>
<tr>
<td></td>
<td>• Additional minor fixes and improvements. For more information, see the changelog on GitHub.</td>
</tr>
<tr>
<td>2.0.3</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

# Client device auth

The client device auth component (aws.greengrass.clientdevices.Auth) authenticates client devices and authorizes client device actions.

**Note**

Client devices are local IoT devices that connect to a Greengrass core device to send MQTT messages and data to process. For more information, see Interact with local IoT devices (p. 477).

**Topics**

- Versions (p. 146)
- Type (p. 147)
- Requirements (p. 147)
- Dependencies (p. 147)
- Configuration (p. 148)
- Local log file (p. 152)
- Changelog (p. 152)

**Versions**

This component has the following versions:
Type

This component is a plugin component (aws.greengrass.plugin). The Greengrass nucleus (p. 136) runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

- The Greengrass service role (p. 630) must be associated to your AWS account and allow the iot:DescribeCertificate permission.
- The core device's AWS IoT policy must allow the following permissions:
  - greengrass:PutCertificateAuthorities
  - greengrass:VerifyClientDeviceIdentity
  - greengrass:VerifyClientDeviceIoTCertificateAssociation
  - greengrass:GetConnectivityInfo

For more information, see AWS IoT policies for data plane operations (p. 611) and Minimal AWS IoT policy to support client devices (p. 616).

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall (p. 640).

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iot.region.amazonaws.com</td>
<td>443</td>
<td>Yes</td>
<td>Used to get information about AWS IoT thing certificates.</td>
</tr>
</tbody>
</table>

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 152) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.
2.0.2

The following table lists the dependencies for version 2.0.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.2.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.1

The following table lists the dependencies for version 2.0.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.2.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.0

The following table lists the dependencies for version 2.0.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.2.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

deviceGroups

Device groups are groups of client devices that have permissions to connect and communicate with a core device. Use selection rules to identify groups of client devices, and define authorization policies that specify the permissions for each device group.

This object contains the following information:

formatVersion

The format version for this configuration object.

Choose from the following options:

- 2021-03-05

definitions

The device groups for this core device. Each definition specifies a selection rule to evaluate if a client device is a member of the group. Each definition also specifies the permissions policy to apply to client devices who match the selection rule. If a client device is a member of multiple device groups, the device's permissions are the union of each group's permissions policy.
This object contains the following information:

**groupNameKey**

The name of this device group. Replace `groupNameKey` with a name that helps you identify this device group.

This object contains the following information:

**selectionRule**

The selection rule that specifies which client devices are members of this device group. When a client device connects, the core device evaluates the selection rule to determine if the client device is a member of this device group. If the client device is a member, the core device uses this device group's policy to authorize the client device's actions.

Use the * wildcard to match multiple client devices with one selection rule clause. You can use this wildcard at the end of the thing name to match client devices whose names start with a string that you specify. You can also use this wildcard to match all client devices.

**Note**

To select a value that contains a colon character (\`:`), escape the colon with a backslash character (\`\`). In formats such as JSON, you must escape backslash characters, so you enter two backslash characters before the colon character. For example, specify `thingName: MyTeam:\:ClientDevice1` to select a thing whose name is `MyTeam:ClientDevice1`.

You can specify the following selector:

- **thingName** – The name of a client device's AWS IoT thing.

**Example Example selection rule**

The following selection rule matches client devices whose names are `MyClientDevice1` or `MyClientDevice2`.

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

**Example Example selection rule (use wildcards)**

The following selection rule matches client devices whose names start with `MyClientDevice`.

```
thingName: MyClientDevice*
```

**Example Example selection rule (match all devices)**

The following selection rule matches all client devices.

```
thingName: *
```

**policyName**

The permissions policy that applies to client devices in this device group. Specify the name of a policy that you define in the `policies` object.
Configuration

policies

The device group authorization policies for client devices that connect to the core device. Each authorization policy specifies a set of actions and the resources on which a client device can perform those actions.

This object contains the following information:

policyNameKey

The name of this authorization policy. Replace policyNameKey with a name that helps you identify this authorization policy. You use this policy name to define which policy applies to a device group.

This object contains the following information:

statementNameKey

The name of this policy statement. Replace statementNameKey with a name that helps you identify this policy statement.

This object contains the following information:

operations

The list of operations to allow for the resources in this policy.

You can include any of the following operations:

- mqtt:connect – Grants permission to connect to the core device. Client devices must have this permission to connect to a core device.

This operation supports the following resources:

- mqtt:clientId:deviceClientId – Restrict access based on the client ID that a client device uses to connect to the core device's MQTT broker. Replace deviceClientId with the client ID to use.

- mqtt:publish – Grants permission to publish MQTT messages to topics.

This operation supports the following resources:

- mqtt:topic:mqttTopic – Restrict access based on the MQTT topic where a client device publishes a message. Replace mqttTopic with the topic to use.

This resource doesn't support MQTT topic wildcards.

- mqtt:subscribe – Grants permission to subscribe to MQTT topic filters to receive messages.

This operation supports the following resources:

- mqtt:topicfilter:mqttTopicFilter – Restrict access based on the MQTT topics where a client device can subscribe to messages. Replace mqttTopicFilter with the topic filter to use.

This resource supports the + and # MQTT topic wildcards. For more information, see MQTT topics in the AWS IoT Core Developer Guide.

The client device can subscribe to the exact topic filters that you allow. For example, if you allow the client device to subscribe to the mqtt:topicfilter:client/+status resource, the client device can subscribe to client/+status but not client/client1/status.

You can specify the * wildcard to allow access to all actions.
resources

The list of resources to allow for the operations in this policy. Specify resources that correspond to the operations in this policy. For example, you might specify a list of MQTT topic resources (mqtt:topic:mqttTopic) in a policy that specifies the mqtt:publish operation.

You can specify the * wildcard to allow access to all resources. You can't use the * wildcard to match partial resource identifiers. For example, you can specify "resources": "+", but you can't specify "resources": "mqtt:clientId:*".

statementDescription

(Optional) A description for this policy statement.

Example Example: Configuration merge update (using a restrictive policy)

The following example configuration specifies to allow client devices whose names start with MyClientDevice to connect and publish/subscribe on all topics.

```json
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}```
Example: Configuration merge update (using a permissive policy)

The following example configuration specifies to allow client devices whose names start with `MyClientDevice` to connect and publish/subscribe on all topics.

```json
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyPermissivePolicy"
      }
    },
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "description": "Allow client devices to perform all actions.",
          "operations": ["*
          
        ],
        "resources": ["*
        
      }
    }
  }
}
```

Local log file

This component uses the same log file as the Greengrass nucleus (p. 136) component.

`/greengrass/v2/logs/greengrass.log`

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

  ```bash
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.2</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.0.1</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
</tbody>
</table>
CloudWatch metrics

The Amazon CloudWatch metrics component (aws.greengrass.Cloudwatch) publishes custom metrics from Greengrass core devices to Amazon CloudWatch. The component enables components to publish CloudWatch metrics, which you can use to monitor and analyze the Greengrass core device's environment. For more information, see Using Amazon CloudWatch metrics in the Amazon CloudWatch User Guide.

To publish a CloudWatch metric with this component, publish a message to a topic where this component subscribes. By default, this component subscribes to the cloudwatch/metric/put local publish/subscribe (p. 410) topic. You can specify other topics, including AWS IoT Core MQTT topics, when you deploy this component.

This component batches metrics that are in the same namespace and publishes them to CloudWatch at regular intervals.

Note
This component provides similar functionality to the CloudWatch metrics connector in AWS IoT Greengrass V1. For more information, see CloudWatch metrics connector in the AWS IoT Greengrass V1 Developer Guide.

Topics
- Versions (p. 153)
- Type (p. 153)
- Requirements (p. 154)
- Dependencies (p. 155)
- Configuration (p. 157)
- Input data (p. 159)
- Output data (p. 161)
- Licenses (p. 161)
- Local log file (p. 162)
- Changelog (p. 162)
- See also (p. 162)

Versions

This component has the following versions:

- 2.0.x

For information about changes in each version of the component, see the changelog (p. 162).

Type

This component is a Lambda component (aws.greengrass.lambda). The Greengrass nucleus (p. 136) runs this component's Lambda function using the Lambda launcher component (p. 188).
Requirements

This component has the following requirements:

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Requirements to run Lambda functions (p. 43).
- Python version 3.7 installed on the core device and added to the PATH environment variable.
- The Greengrass device role (p. 627) must allow the cloudwatch:PutMetricData action, as shown in the following example IAM policy.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "cloudwatch:PutMetricData"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

For more information, see Amazon CloudWatch permissions reference in the Amazon CloudWatch User Guide.

- To receive output data from this component, you must merge the following configuration update for the legacy subscription router component (p. 194) when you deploy this component. The legacy subscription router component (aws.greengrass.LegacySubscriptionRouter) is a dependency of this component. This configuration specifies the topic where this component publishes responses.

Legacy subscription router v2.1.x

```json
{
  "subscriptions": {
    "aws-greengrass-cloudwatch": {
      "id": "aws-greengrass-cloudwatch",
      "source": "component:aws.greengrass.Cloudwatch",
      "subject": "cloudwatch/metric/put/status",
      "target": "cloud"
    }
  }
}
```

Legacy subscription router v2.0.x

```json
{
  "subscriptions": {
    "aws-greengrass-cloudwatch": {
      "id": "aws-greengrass-cloudwatch",
      "subject": "cloudwatch/metric/put/status",
      "target": "cloud"
    }
  }
}
```
• Replace `region` with the AWS Region that you use.
• Replace `version` with the version of the Lambda function that this component runs. To find the Lambda function version, you must view the recipe for the version of this component that you want to deploy. Open this component’s details page in the AWS IoT Greengrass console, and look for the Lambda function key-value pair. This key-value pair contains the name and version of the Lambda function.

**Important**
You must update the Lambda function version on the legacy subscription router every time you deploy this component. This ensures that you use the correct Lambda function version for the component version that you deploy.

For more information, see Create deployments (p. 384).

### Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall (p. 640).

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>monitoring.region.amazonaws.com</code></td>
<td>443</td>
<td>Yes</td>
<td>Upload CloudWatch metrics.</td>
</tr>
</tbody>
</table>

### Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

#### 2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

#### 2.0.6

The following table lists the dependencies for version 2.0.6 of this component.
### Dependencies

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.0.4</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

**2.0.5**

The following table lists the dependencies for version 2.0.5 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

**2.0.4**

The following table lists the dependencies for version 2.0.4 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

**2.0.3**

The following table lists the dependencies for version 2.0.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.3 &lt;2.1.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>&gt;=1.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>&gt;=1.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>&gt;=1.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>
For more information about component dependencies, see the component recipe reference (p. 343).

**Configuration**

This component provides the following configuration parameters that you can customize when you deploy the component.

**Note**
This component's default configuration includes Lambda function parameters. We recommend that you edit only the following parameters to configure this component on your devices.

**lambdaParams**

An object that contains the parameters for this component's Lambda function. This object contains the following information:

**EnvironmentVariables**

An object that contains the Lambda function's parameters. This object contains the following information:

**PUBLISH_INTERVAL**

(Optional) The maximum number of seconds to wait before the component publishes batched metrics for a given namespace. To configure the component to publish metrics as it receives them, which means without batching, specify 0.

The component publishes to CloudWatch after it receives 20 metrics in the same namespace or after the interval that you specify.

**Note**
The component doesn't guarantee the order in which events publish.

This value can be at most 900 seconds.

Default: 10 seconds

**MAX_METRICS_TO_RETAIN**

(Optional) The maximum number of metrics across all namespaces to save in memory before the component replaces them with newer metrics.

This limit applies when the core device doesn't have a connection to the internet, so the component buffers the metrics to publish later. When the buffer is full, the component replaces the oldest metrics with newer ones. Metrics in a given namespace replace only metrics in the same namespace.

**Note**
If the host process for the component is interrupted, the component doesn't save metrics. This can happen during a deployment or when the core device restarts, for example.

This value must be at least 2,000 metrics.

Default: 5,000 metrics

**PUBLISH_REGION**

(Optional) The AWS Region to which to publish CloudWatch metrics. This value overrides the default Region for the core device. This parameter is required only for cross-Region metrics.

**containerMode**

(Optional) The containerization mode for this component. Choose from the following options:
Configuration

- **NoContainer** – The component doesn't run in an isolated runtime environment.
- **GreengrassContainer** – The component runs in an isolated runtime environment inside the AWS IoT Greengrass container.

Default: `GreengrassContainer`

**containerParams**

(Optional) An object that contains the container parameters for this component. The component uses these parameters if you specify `GreengrassContainer` for `containerMode`.

This object contains the following information:

**memorySize**

(Optional) The amount of memory (in kilobytes) to allocate to the component.

Defaults to 64 MB (65,535 KB).

**pubsubTopics**

(Optional) An object that contains the topics where the component subscribes to receive messages. You can specify each topic and whether the component subscribes to MQTT topics from AWS IoT Core or local publish/subscribe topics.

This object contains the following information:

0 – This is an array index as a string.

An object that contains the following information:

**type**

(Optional) The type of publish/subscribe messaging that this component uses to subscribe to messages. Choose from the following options:

- **Pubsub** – Subscribe to local publish/subscribe messages. If you choose this option, the topic can't contain MQTT wildcards. For more information about how to send messages from custom component when you specify this option, see Publish/subscribe local messages (p. 410).
- **IotCore** – Subscribe to AWS IoT Core MQTT messages. If you choose this option, the topic can contain MQTT wildcards. For more information about how to send messages from custom components when you specify this option, see Publish/subscribe AWS IoT Core MQTT messages (p. 433).

Default: Pubsub

**topic**

(Optional) The topic to which the component subscribes to receive messages. If you specify `IotCore` for `type`, you can use MQTT wildcards (+ and #) in this topic.

**Example Example: Configuration merge update (container mode)**

```
{
    "containerMode": "GreengrassContainer"
}
```

**Example Example: Configuration merge update (no container mode)**

```
{
    "containerMode": "NoContainer"
}
```
Input data

This component accepts metrics on the following topic and publishes the metrics to CloudWatch. By default, this component subscribes to local publish/subscribe messages. For more information about how to publish messages to this component from your custom components, see Publish/subscribe local messages (p. 410).

Default topic (local publish/subscribe): cloudwatch/metric/put

The message accepts the following properties. Input messages must be in JSON format.

request

The metric in this message.

The request object contains the metric data to publish to CloudWatch. The metric values must meet the specifications of the PutMetricData operation.

Type: object that contains the following information:

namespace

The user-defined namespace for the metric data in this request. CloudWatch uses namespaces as containers for metric data points.

Note
You can't specify a namespace that begins with the reserved string AWS/.

Type: string

Valid pattern: [^:].*

metricData

The data for the metric.

Type: object that contains the following information:

metricName

The name of the metric.

Type: string

value

The value for the metric.

Note
CloudWatch rejects values that are too small or too large. The value must be between $8.515920e-109$ and $1.174271e+108$ (Base 10) or $2e-360$ and $2e360$ (Base 2). CloudWatch doesn't support special values such as NaN, +Infinity, and -Infinity.

Type: double

dimensions

(Optional) The for the metric. Dimensions provide additional information about the metric and its data. A metric can define up to 10 dimensions.

Type: array of objects that each contain the following information:
Input data

name
(Optional) The dimension name.
Type: string
value
(Optional) The dimension value.
Type: string
timestamp
(Optional) The time at which the metric data was received, expressed in seconds in Unix epoch time.
Defaults to the time at which the component receives the message.
Type: double
Note
When you send multiple metrics within a single Lambda function, we recommend that you retrieve the timestamp separately for each metric. Don't use a variable to store the timestamp.
unit
(Optional) The unit of the metric.
Type: string
Valid values: Seconds, Microseconds, Milliseconds, Bytes, Kilobytes, Megabytes, Gigabytes, Terabytes, Bits, Kilobits, Megabits, Gigabits, Terabits, Percent, Count, Bytes/Second, Kilobytes/Second, Megabytes/Second, Gigabytes/Second, Terabytes/Second, Bits/Second, Kilobits/Second, Megabits/Second, Gigabits/Second, Terabits/Second, Count/Second, None
Defaults to None.

Note
All quotas that apply to the CloudWatch PutMetricData API apply to metrics that you publish with this component. The following quotas are especially important:

- 40 KB limit on the API payload
- 20 metrics per API request
- 150 transactions per second (TPS) for the PutMetricData API

For more information, see CloudWatch service quotas in the CloudWatch User Guide.

Example Example input

```json
{
  "request": {
    "namespace": "Greengrass",
    "metricData": {
      "metricName": "latency",
      "dimensions": [
        {
          "name": "hostname",
          "value": "test_hostname"
        }
      ]
    }
  }
}```
Output data

This component publishes responses as output data on the following MQTT topic by default. You must specify this topic as the subject in the configuration for the legacy subscription router component (p. 194). For more information about how to subscribe to messages on this topic in your custom components, see Publish/subcribe AWS IoT Core MQTT messages (p. 433).

Default topic (AWS IoT Core MQTT): cloudwatch/metric/put/status

Example Example output: Success

The response includes the namespace of the metric data and the RequestId field from the CloudWatch response.

```json
{
  "response": {
    "cloudwatch_rid": "70573243-d723-11e8-b095-75ff2EXAMPLE",
    "namespace": "Greengrass",
    "status": "success"
  }
}
```

Example Example output: Failure

```json
{
  "response": {
    "namespace": "Greengrass",
    "error": "InvalidInputException",
    "error_message": "cw metric is invalid",
    "status": "fail"
  }
}
```

**Note**

If the component detects a retryable error, such as a connection error, it retries the publish in the next batch.

**Licenses**

This component includes the following third-party software/licensing:

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF License
- docutils/PSF License, GNU General Public License (GPL), Python Software Foundation License, Public Domain
- jmespath/MIT License
- s3transfer/Apache License 2.0
**Local log file**

This component uses the following log file.

```
greengrass/v2/logs/aws.greengrass.Cloudwatch.log
```

**To view this component's logs**

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.Cloudwatch.log
```

**Changelog**

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.7</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.0.6</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.0.5</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>2.0.4</td>
<td>Version updated for Greengrass nucleus version 2.1.0 release.</td>
</tr>
<tr>
<td>2.0.3</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

**See also**

- Using Amazon CloudWatch metrics in the Amazon CloudWatch User Guide
- PutMetricData in the Amazon CloudWatch API Reference

**Docker application manager**

The Docker application manager component (`aws.greengrass.DockerApplicationManager`) enables AWS IoT Greengrass to download Docker images from public image registries. It also enables AWS IoT Greengrass to manage credentials to download images from private repositories in Amazon Elastic Container Registry (Amazon ECR).

When you develop a custom component that runs a Docker container, include the Docker application manager as a dependency to download the Docker images that are specified as artifacts in your component. For more information, see Run a Docker container (p. 331).
Versions

This component has the following versions:

- 2.0.x

Type

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

- **Docker Engine** 1.9.1 or later installed and running on your Greengrass core device. Version 20.10 is the latest version that is verified to work with the connector. You must install Docker directly on the core device before you deploy custom components that run Docker containers.
- The Docker daemon started and running on the core device before you deploy this component.
- **Docker images stored in one of the following supported image sources:**
  - Public and private image repositories in Amazon Elastic Container Registry (Amazon ECR)
  - Public Docker Hub repository
  - Public Docker Trusted Registry
  - Docker images included as artifacts in your custom Docker container components. Use the following URI formats to specify your Docker images:
    - Private Amazon ECR image: docker:account-id.dkr.ecr.region.amazonaws.com/repository/image[:tag[@digest]]
    - Public Amazon ECR image: docker:public.ecr.aws/repository/image[:tag[@digest]]
    - Public Docker Hub image: docker:name[:tag[@digest]]

For more information, see Run a Docker container (p. 331).

**Note**

If you don't specify the image tag or image digest in the artifact URI for an image, then the Docker application manager pulls the latest available version of that image when you deploy your custom Docker container component. To ensure that all of your core devices run the same version of an image, we recommend that you include the image tag or image digest in the artifact URI.

- Root user permissions or Docker configured for you to run it as a non-root user. Adding a user to the docker group enables you to call docker commands without sudo. To add ggc_user, or the non-
root user that you use to run AWS IoT Greengrass, to the docker group that you configure, run `sudo usermod -aG docker user-name`.

- Docker configured to use a proxy server. This is required only if you are running AWS IoT Greengrass behind a network proxy.
- If your Docker images are stored in an Amazon ECR private registry, then you must include the token exchange service component as a dependency in the Docker container component. Also, the Greengrass device role (p. 627) must allow the `ecr:GetAuthorizationToken`, `ecr:BatchGetImage`, and `ecr:GetDownloadUrlForLayer` actions, as shown in the following example IAM policy.

```
{
   "Version": "2012-10-17",
   "Statement": [
       {
           "Action": [
               "ecr:GetAuthorizationToken",
               "ecr:BatchGetImage",
               "ecr:GetDownloadUrlForLayer"
           ],
           "Resource": [
               "*"
           ],
           "Effect": "Allow"
       }
   ]
}
```

### Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall (p. 640).

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecr.region.amazonaws.com</td>
<td>443</td>
<td>No</td>
<td>Required if you download Docker images from Amazon ECR.</td>
</tr>
<tr>
<td>hub.docker.com</td>
<td>443</td>
<td>No</td>
<td>Required if you download Docker images from Docker Hub.</td>
</tr>
<tr>
<td>registry.hub.docker.com/v1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its
dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 166) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.2

The following table lists the dependencies for version 2.0.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.1

The following table lists the dependencies for version 2.0.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.0

The following table lists the dependencies for version 2.0.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component doesn't have any configuration parameters.

Local log file

This component uses the following log file.

`/greengrass/v2/logs/aws.greengrass.DockerApplicationManager.log`
To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

  ```bash
  sudo tail -f /greengrass/v2/logs/aws.greengrass.DockerApplicationManager.log
  ```

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.3</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.0.2</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.0.1</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>2.0.0</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

See also

- Run a Docker container (p. 331)

Greengrass CLI

The Greengrass CLI component (aws.greengrass.Cli) provides a local command-line interface that you can use on core devices to develop and debug components locally. The Greengrass CLI lets you create local deployments and restart components on the core device, for example.

You can install this component when you install the AWS IoT Greengrass Core software. For more information, see Getting started with AWS IoT Greengrass V2 (p. 24).

After you install this component, run the following command to view its help documentation. When this component installs, it adds a symbolic link to `greengrass-cli` in the `/greengrass/v2/bin` folder. You can run the Greengrass CLI from this path or add it to your `PATH` environment variable to run `greengrass-cli` without its absolute path.

  ```bash
  /greengrass/v2/bin/greengrass-cli help
  ```

The following command restarts a component named `com.example.HelloWorld`, for example.

  ```bash
  sudo /greengrass/v2/bin/greengrass-cli component restart \ 
  --names "com.example.HelloWorld"
  ```

For more information, see Greengrass Command Line Interface (p. 595).

Topics

- Versions (p. 167)
- Type (p. 167)
• Requirements (p. 167)
• Dependencies (p. 167)
• Configuration (p. 168)
• Local log file (p. 169)
• Changelog (p. 169)

Versions

This component has the following versions:

• 2.4.x
• 2.3.x
• 2.2.x
• 2.1.x
• 2.0.x

Type

This component is a plugin component (aws.greengrass.plugin). The Greengrass nucleus (p. 136) runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

• You must be authorized to use the Greengrass CLI to interact with the AWS IoT Greengrass Core software. Do one of the following to use the Greengrass CLI:
  • Use a root user (sudo).
  • Use the system user that runs the AWS IoT Greengrass Core software.
  • Use a system user that's in a system group that you specify in the AuthorizedPosixGroups (p. 168) configuration parameter.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 169) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.4.0

The following table lists the dependencies for version 2.4.0 of this component.
### Compatibility

#### Dependency

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

#### 2.3.0

The following table lists the dependencies for version 2.3.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

#### 2.2.0

The following table lists the dependencies for version 2.2.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

#### 2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

#### 2.0.x

The following table lists the dependencies for version 2.0.x of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.x &lt;2.1.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

**Note**

The minimum compatible version of the Greengrass nucleus corresponds to the patch version of the Greengrass CLI component.

For more information about component dependencies, see the component recipe reference (p. 343).

### Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

**AuthorizedPosixGroups**

(Optional) A string that contains a comma-separated list of system groups. You authorize these system groups to use the Greengrass CLI to interact with the AWS IoT Greengrass Core software. You
can specify group names or group IDs. For example, `group1,1002,group3` authorizes three system groups (`group1`, `1002`, and `group3`) to use the Greengrass CLI.

If you don’t specify any groups to authorize, you can use the Greengrass CLI as the root user (`sudo`) or as the system user that runs the AWS IoT Greengrass Core software.

**Example Example: Configuration merge update**

The following example configuration specifies to authorize three system groups (`group1`, `1002`, and `group3`) to use the Greengrass CLI.

```json
{
    "AuthorizedPosixGroups": "group1,1002,group3"
}
```

**Local log file**

This component uses the same log file as the Greengrass nucleus (p. 136) component.

```
/greengrass/v2/logs/greengrass.log
```

**To view this component's logs**

- Run the following command on the core device to view this component's log file in real time.
  Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

**Changelog**

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2.4.0   | New features  
|         | • Adds support for system resource limits. When you create a local deployment, you can configure the maximum amount of CPU and RAM usage that each component's processes can use on the core device. For more information, see [Configure system resource limits for components](p. 124) and the [deployment create command](p. 600). |
| 2.3.0   | Version updated for Greengrass nucleus version 2.3.0 release. |
| 2.2.0   | Version updated for Greengrass nucleus version 2.2.0 release. |
| 2.1.0   | Version updated for Greengrass nucleus version 2.1.0 release. |
| 2.0.5   | Version updated for Greengrass nucleus version 2.0.5 release. |
| 2.0.4   | Version updated for Greengrass nucleus version 2.0.4 release. |
| 2.0.3   | Initial version. |
Device Defender

The Device Defender component (aws.greengrass.DeviceDefender) notifies administrators about changes in the state of Greengrass core devices. This can help identify unusual behavior that might indicate a compromised device. For more information, see AWS IoT Device Defender in the AWS IoT Core Developer Guide.

This component reads system metrics from the /proc directory on the core device. Then, it publishes the metrics to AWS IoT Device Defender. For more information about how to read and interpret the metrics that this component reports, see Device metrics document specification in the AWS IoT Core Developer Guide.

**Note**

This component provides similar functionality to the Device Defender connector in AWS IoT Greengrass V1. For more information, see Device Defender connector in the AWS IoT Greengrass V1 Developer Guide.

**Topics**

- Versions (p. 170)
- Type (p. 170)
- Requirements (p. 170)
- Dependencies (p. 171)
- Configuration (p. 173)
- Input data (p. 175)
- Output data (p. 175)
- Local log file (p. 176)
- Licenses (p. 176)
- Changelog (p. 177)

**Versions**

This component has the following versions:

- 2.0.x

**Type**

This component is a Lambda component (aws.greengrass.lambda). The Greengrass nucleus (p. 136) runs this component's Lambda function using the Lambda launcher component (p. 188).

For more information, see Component types (p. 323).

**Requirements**

This component has the following requirements:

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Requirements to run Lambda functions (p. 43).
- Python version 3.7 installed on the core device and added to the PATH environment variable.
- AWS IoT Device Defender configured to use the Detect feature to keep track of violations. For more information, see Detect in the AWS IoT Core Developer Guide.
- The psutil library installed on the core device. Version 5.7.0 is the latest version that is verified to work with the component.
- The cbor library installed on the core device. Version 1.0.0 is the latest version that is verified to work with the component.
- To receive output data from this component, you must merge the following configuration update for the legacy subscription router component (p. 194) when you deploy this component. The legacy subscription router component (aws.greengrass.LegacySubscriptionRouter) is a dependency of this component. This configuration specifies the topic where this component publishes responses.

Legacy subscription router v2.1.x

```
{
  "subscriptions": {
    "aws-greengrass-device-defender": {
      "id": "aws-greengrass-device-defender",
      "source": "component:aws.greengrass.DeviceDefender",
      "subject": "$aws/things/+/defender/metrics/json",
      "target": "cloud"
    }
  }
}
```

Legacy subscription router v2.0.x

```
{
  "subscriptions": {
    "aws-greengrass-device-defender": {
      "id": "aws-greengrass-device-defender",
      "subject": "$aws/things/+/defender/metrics/json",
      "target": "cloud"
    }
  }
}
```

- Replace region with the AWS Region that you use.
- Replace version with the version of the Lambda function that this component runs. To find the Lambda function version, you must view the recipe for the version of this component that you want to deploy. Open this component's details page in the AWS IoT Greengrass console, and look for the Lambda function key-value pair. This key-value pair contains the name and version of the Lambda function.

  **Important**
  You must update the Lambda function version on the legacy subscription router every time you deploy this component. This ensures that you use the correct Lambda function version for the component version that you deploy.

  For more information, see Create deployments (p. 384).

**Dependencies**

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 177) of this component and the semantic version constraints that define the component.
versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.0.6

The following table lists the dependencies for version 2.0.6 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.0.5

The following table lists the dependencies for version 2.0.5 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.0.4

The following table lists the dependencies for version 2.0.4 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>
The following table lists the dependencies for version 2.0.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.3 &lt;2.1.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>&gt;=1.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>&gt;=1.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>&gt;=1.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

### Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

**Note**

This component's default configuration includes Lambda function parameters. We recommend that you edit only the following parameters to configure this component on your devices.

**lambdaParams**

An object that contains the parameters for this component's Lambda function. This object contains the following information:

**EnvironmentVariables**

An object that contains the Lambda function's parameters. This object contains the following information:

**PROCFS_PATH**

(Optional) The path to the `/proc` folder.
- To run this component in a container, use the default value, `/host–proc`. The component runs in a container by default.
- To run this component in no container mode, specify `/proc` for this parameter.

Default: `/host–proc`. This is the default path where this component mounts the `/proc` folder in the container.

**Note**

This component has read-only access to this folder.
SAMPLE_INTERVAL_SECONDS
(Optional) The amount of time in seconds between each cycle where the component
gathers and reports metrics.

The minimum value is 300 seconds (5 minutes).

Default: 300 seconds

containerMode
(Optional) The containerization mode for this component. Choose from the following options:

- GreengrassContainer – The component runs in an isolated runtime environment inside the
  AWS IoT Greengrass container.
- NoContainer – The component doesn't run in an isolated runtime environment.

If you specify this option, you must specify /proc for the PROCFS_PATH environment variable
parameter.

Default: GreengrassContainer

containerParams
(Optional) An object that contains the container parameters for this component. The component
uses these parameters if you specify GreengrassContainer for containerMode.

This object contains the following information:

memorySize
(Optional) The amount of memory (in kilobytes) to allocate to the component.

Defaults to 50,000 KB.

pubsubTopics
(Optional) An object that contains the topics where the component subscribes to receive messages.
You can specify each topic and whether the component subscribes to MQTT topics from AWS IoT
Core or local publish/subscribe topics.

This object contains the following information:

0 – This is an array index as a string.

An object that contains the following information:

type
(Optional) The type of publish/subscribe messaging that this component uses to subscribe
to messages. Choose from the following options:

- Pubsub – Subscribe to local publish/subscribe messages. If you choose this option, the
topic can’t contain MQTT wildcards. For more information about how to send messages
from custom component when you specify this option, see Publish/subscribe local
messages (p. 410).
- IotCore – Subscribe to AWS IoT Core MQTT messages. If you choose this option, the
topic can contain MQTT wildcards. For more information about how to send messages
from custom components when you specify this option, see Publish/subscribe AWS IoT
Core MQTT messages (p. 433).

Default: Pubsub

topic
(Optional) The topic to which the component subscribes to receive messages. If you specify
IotCore for type, you can use MQTT wildcards (+ and #) in this topic.
Example Example: Configuration merge update (container mode)

```json
{
    "lambdaExecutionParameters": {
        "EnvironmentVariables": {
            "PROCFS_PATH": "/host_proc"
        },
        "containerMode": "GreengrassContainer"
    }
}
```

Example Example: Configuration merge update (no container mode)

```json
{
    "lambdaExecutionParameters": {
        "EnvironmentVariables": {
            "PROCFS_PATH": "/proc"
        },
        "containerMode": "NoContainer"
    }
}
```

## Input data

This component doesn't accept messages as input data.

## Output data

This component publishes security metrics to the following reserved topic for AWS IoT Device Defender. This component replaces `coreDeviceName` with the name of the core device when it publishes the metrics.

**Topic (AWS IoT Core MQTT):** `$aws/things/coreDeviceName/defender/metrics/json`

Example Example output

```json
{
    "header": {
        "report_id": 1529963534,
        "version": "1.0"
    },
    "metrics": {
        "listening_tcp_ports": {
            "ports": [
                {
                    "interface": "eth0",
                    "port": 24800
                },
                {
                    "interface": "eth0",
                    "port": 22
                },
                {
                    "interface": "eth0",
                    "port": 53
                }
            ],
            "total": 3
        }
    }
}
```
"listening_udp_ports": {
    "ports": [
    {
        "interface": "eth0",
        "port": 5353
    },
    {
        "interface": "eth0",
        "port": 67
    }
    ],
    "total": 2
},
"network_stats": {
    "bytes_in": 1157864729406,
    "bytes_out": 1170821865,
    "packets_in": 693092175031,
    "packets_out": 738917180
},
"tcp_connections": {
    "established_connections":{
        "connections": [
        {
            "local_interface": "eth0",
            "local_port": 80,
            "remote_addr": "192.168.0.1:8000"
        },
        {
            "local_interface": "eth0",
            "local_port": 80,
            "remote_addr": "192.168.0.1:8000"
        }
        ],
    "total": 2
    }
}

For more information about the metrics that this component reports, see Device metrics document specification in the AWS IoT Core Developer Guide.

Local log file

This component uses the following log file.

/greengrass/v2/logs/aws.greengrass.DeviceDefender.log

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace /greengrass/v2 with the path to the AWS IoT Greengrass root folder.

    sudo tail -f /greengrass/v2/logs/aws.greengrass.DeviceDefender.log

Licenses

This component is released under the Greengrass Core Software License Agreement.
Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.7</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.0.6</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.0.5</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>2.0.4</td>
<td>Version updated for Greengrass nucleus version 2.1.0 release.</td>
</tr>
<tr>
<td>2.0.3</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

IP detector

The IP detector component (aws.greengrass.clientdevices.IPDetector) does the following:

- Monitors the Greengrass core device's network connectivity information. This information includes the core device's network endpoints and the port where an MQTT broker operates.
- Updates the core device's connectivity information in the AWS IoT Greengrass cloud service.

Client devices can use Greengrass cloud discovery to retrieve associated core devices' connectivity information. Then, client devices can try to connect to each core device until they successfully connect.

**Note**
Client devices are local IoT devices that connect to a Greengrass core device to send MQTT messages and data to process. For more information, see [Interact with local IoT devices](p. 477).

The IP detector component replaces a core device's existing connectivity information with the information it detects. Because this component removes existing information, you can either use the IP detector component, or manually manage connectivity information.

**Note**
The IP detector component detects only IPv4 addresses.

Topics

- Versions (p. 177)
- Type (p. 178)
- Requirements (p. 178)
- Dependencies (p. 178)
- Configuration (p. 179)
- Local log file (p. 179)
- Changelog (p. 179)

Versions

This component has the following versions:

- 2.0.x
Type

This component is a plugin component (aws.greengrass.plugin). The Greengrass nucleus (p. 136) runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

- The Greengrass service role (p. 630) must be associated to your AWS account and allow the iot:GetThingShadow and iot:UpdateThingShadow permissions.
- The core device's AWS IoT policy must allow the greengrass:UpdateConnectivityInfo permission. For more information, see AWS IoT policies for data plane operations (p. 611) and Minimal AWS IoT policy to support client devices (p. 616).
- The core device's MQTT broker component, such as the Moquette MQTT broker (p. 283), must run on port 8883.
- If you have a complex network setup, the IP detector component might not be able to identify the endpoints where client devices can connect to the core device. If the IP detector component can't manage the endpoints, you must manually manage the core device endpoints instead. For example, if the core device is behind a router that forwards the MQTT broker port to it, you must specify the router's IP address as an endpoint for the core device. For more information, see Manage core device endpoints (p. 492).

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 179) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.0.2

The following table lists the dependencies for version 2.0.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.2.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.1

The following table lists the dependencies for version 2.0.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.2.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>
2.0.0

The following table lists the dependencies for version 2.0.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.2.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

**includeIPv4LoopbackAddrs**

(Optional) You can enable this option to detect and report IPv4 loopback addresses, which are IP addresses where a device can communicate with itself, such as localhost. Use this option in test environments where the core device and client device run on the same system.

Default: false

**includeIPv4LinkLocalAddrs**

(Optional) You can enable this option to detect and report IPv4 link-local addresses. Use this option if the core device's network doesn't have Dynamic Host Configuration Protocol (DHCP) or statically assigned IP addresses.

Default: false

Local log file

This component uses the same log file as the Greengrass nucleus (p. 136) component.

`/greengrass/v2/logs/greengrass.log`

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

  ```
sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.2</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.0.1</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
</tbody>
</table>
Kinesis Data Firehose

The Kinesis Data Firehose component (aws.greengrass.KinesisFirehose) publishes data through Amazon Kinesis Data Firehose delivery streams to destinations, such as Amazon S3, Amazon Redshift, and Amazon OpenSearch Service. For more information, see What is Amazon Kinesis Data Firehose? in the Amazon Kinesis Data Firehose Developer Guide.

To publish to a Kinesis delivery stream with this component, publish a message to a topic where this component subscribes. By default, this component subscribes to the kinesisfirehose/message and kinesisfirehose/message/binary/# local publish/subscribe (p. 410) topics. You can specify other topics, including AWS IoT Core MQTT topics, when you deploy this component.

Note
This component provides similar functionality to the Kinesis Data Firehose connector in AWS IoT Greengrass V1. For more information, see Kinesis Data Firehose connector in the AWS IoT Greengrass V1 Developer Guide.

Topics
- Versions (p. 180)
- Type (p. 180)
- Requirements (p. 180)
- Dependencies (p. 182)
- Configuration (p. 184)
- Input data (p. 186)
- Output data (p. 187)
- Local log file (p. 187)
- Licenses (p. 187)
- Changelog (p. 188)
- See also (p. 188)

Versions
This component has the following versions:
- 2.0.x

Type
This component is a Lambda component (aws.greengrass.lambda). The Greengrass nucleus (p. 136) runs this component's Lambda function using the Lambda launcher component (p. 188).

For more information, see Component types (p. 323).

Requirements
This component has the following requirements:
Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Requirements to run Lambda functions (p. 43).

- Python version 3.7 installed on the core device and added to the PATH environment variable.
- The Greengrass device role (p. 627) must allow the firehose:PutRecord and firehose:PutRecordBatch actions, as shown in the following example IAM policy.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "firehose:PutRecord",
        "firehose:PutRecordBatch"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:firehose:region:account-id:deliverystream/stream-name"
      ]
    }
  ]
}
```

You can dynamically override the default delivery stream in the input message payload for this component. If your application uses this feature, the IAM policy must include all target streams as resources. You can grant granular or conditional access to resources (for example, by using a wildcard * naming scheme).

- To receive output data from this component, you must merge the following configuration update for the legacy subscription router component (p. 194) when you deploy this component. The legacy subscription router component (aws.greengrass.LegacySubscriptionRouter) is a dependency of this component. This configuration specifies the topic where this component publishes responses.

Legacy subscription router v2.1.x

```json
{
  "subscriptions": {
    "aws-greengrass-kinesisfirehose": {
      "id": "aws-greengrass-kinesisfirehose",
      "source": "component:aws.greengrass.KinesisFirehose",
      "subject": "kinesisfirehose/message/status",
      "target": "cloud"
    }
  }
}
```

Legacy subscription router v2.0.x

```json
{
  "subscriptions": {
    "aws-greengrass-kinesisfirehose": {
      "id": "aws-greengrass-kinesisfirehose",
      "subject": "kinesisfirehose/message/status",
      "target": "cloud"
    }
  }
}
```

- Replace region with the AWS Region that you use.
• Replace `version` with the version of the Lambda function that this component runs. To find the Lambda function version, you must view the recipe for the version of this component that you want to deploy. Open this component’s details page in the AWS IoT Greengrass console, and look for the `Lambda function` key-value pair. This key-value pair contains the name and version of the Lambda function.

**Important**
You must update the Lambda function version on the legacy subscription router every time you deploy this component. This ensures that you use the correct Lambda function version for the component version that you deploy.

For more information, see Create deployments (p. 384).

### Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall (p. 640).

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>firehose.{region}.amazonaws.com</td>
<td>443</td>
<td>Yes</td>
<td>Upload data to Kinesis Data Firehose.</td>
</tr>
</tbody>
</table>

### Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 188) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

#### 2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

#### 2.0.6

The following table lists the dependencies for version 2.0.6 of this component.
## Dependencies

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

### 2.0.5

The following table lists the dependencies for version 2.0.5 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

### 2.0.4

The following table lists the dependencies for version 2.0.4 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

### 2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.3 &lt;2.1.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>&gt;=1.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>&gt;=1.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>&gt;=1.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>
For more information about component dependencies, see the component recipe reference (p. 343).

**Configuration**

This component provides the following configuration parameters that you can customize when you deploy the component.

**Note**

This component's default configuration includes Lambda function parameters. We recommend that you edit only the following parameters to configure this component on your devices.

**lambdaParams**

An object that contains the parameters for this component's Lambda function. This object contains the following information:

**EnvironmentVariables**

An object that contains the Lambda function's parameters. This object contains the following information:

**DEFAULT_DELIVERY_STREAM_ARN**

The ARN of the default Kinesis Data Firehose delivery stream where the component sends data. You can override the destination stream with the `delivery_stream_arn` property in the input message payload.

**Note**

The core device role must allow the required actions on all target delivery streams. For more information, see Requirements (p. 180).

**PUBLISH_INTERVAL**

(Optional) The maximum number of seconds to wait before the component publishes batched data to Kinesis Data Firehose. To configure the component to publish metrics as it receives them, which means without batching, specify 0. This value can be at most 900 seconds.

Default: 10 seconds

**DELIVERY_STREAM_QUEUE_SIZE**

(Optional) The maximum number of records to retain in memory before the component rejects new records for the same delivery stream.

This value must be at least 2,000 records.

Default: 5,000 records

**containerMode**

(Optional) The containerization mode for this component. Choose from the following options:

- **NoContainer** – The component doesn't run in an isolated runtime environment.
- **GreengrassContainer** – The component runs in an isolated runtime environment inside the AWS IoT Greengrass container.

Default: GreengrassContainer

**containerParams**

(Optional) An object that contains the container parameters for this component. The component uses these parameters if you specify GreengrassContainer for `containerMode`.

This object contains the following information:
memorySize

(Optional) The amount of memory (in kilobytes) to allocate to the component.

Defaults to 64 MB (65,535 KB).

pubsubTopics

(Optional) An object that contains the topics where the component subscribes to receive messages. You can specify each topic and whether the component subscribes to MQTT topics from AWS IoT Core or local publish/subscribe topics.

This object contains the following information:

0 – This is an array index as a string.

An object that contains the following information:

type

(Optional) The type of publish/subscribe messaging that this component uses to subscribe to messages. Choose from the following options:

- Pubsub – Subscribe to local publish/subscribe messages. If you choose this option, the topic can’t contain MQTT wildcards. For more information about how to send messages from custom component when you specify this option, see Publish/subscribe local messages (p. 410).
- IotCore – Subscribe to AWS IoT Core MQTT messages. If you choose this option, the topic can contain MQTT wildcards. For more information about how to send messages from custom components when you specify this option, see Publish/subscribe AWS IoT Core MQTT messages (p. 433).

Default: Pubsub

topic

(Optional) The topic to which the component subscribes to receive messages. If you specify IotCore for type, you can use MQTT wildcards (+ and #) in this topic.

Example Example: Configuration merge update (container mode)

```json
{
    "lambdaExecutionParameters": {
        "EnvironmentVariables": {
        }
    },
    "containerMode": "GreengrassContainer"
}
```

Example Example: Configuration merge update (no container mode)

```json
{
    "lambdaExecutionParameters": {
        "EnvironmentVariables": {
        }
    },
    "containerMode": "NoContainer"
}
```
Input data

This component accepts stream content on the following topics and sends the content to the target delivery stream. The component accepts two types of input data:

- JSON data on the kinesisfirehose/message topic.
- Binary data on the kinesisfirehose/message/binary/# topic.

Default topic for JSON data (local publish/subscribe): kinesisfirehose/message

The message accepts the following properties. Input messages must be in JSON format.

**request**

The data to send to the delivery stream and the target delivery stream, if different from the default stream.

Type: object that contains the following information:

**data**

The data to send to the delivery stream.

Type: string

**delivery_stream_arn**

(Optional) The ARN of the target Kinesis Data Firehose delivery stream. Specify this property to override the default delivery stream.

Type: string

**id**

An arbitrary ID for the request. Use this property to map an input request to an output response. When you specify this property, the component sets the id property in the response object to this value.

Type: string

Example Example input

```json
{
    "request": {
        "data": "Data to send to the delivery stream."
    },
    "id": "request123"
}
```

Default topic for binary data (local publish/subscribe): kinesisfirehose/message/binary/#

Use this topic to send a message that contains binary data. The component doesn't parse binary data. The component streams the data as is.

To map the input request to an output response, replace the # wildcard in the message topic with an arbitrary request ID. For example, if you publish a message to kinesisfirehose/message/binary/request123, the id property in the response object is set to request123.

If you don't want to map a request to a response, you can publish your messages to kinesisfirehose/message/binary/. Be sure to include the trailing slash (/).
Output data

This component publishes responses as output data on the following MQTT topic by default. You must specify this topic as the subject in the configuration for the legacy subscription router component (p. 194). For more information about how to subscribe to messages on this topic in your custom components, see Publish/subscribe AWS IoT Core MQTT messages (p. 433).

**Default topic (AWS IoT Core MQTT):** kinesisfirehose/message/status

**Example Example output**

The response contains the status of each data record sent in the batch.

```json
{
    "response": [
        {
            "ErrorCode": "error",
            "ErrorMessage": "test error",
            "id": "request123",
            "status": "fail"
        },
        {
            "firehose_record_id": "xyz2",
            "id": "request456",
            "status": "success"
        },
        {
            "firehose_record_id": "xyz3",
            "id": "request890",
            "status": "success"
        }
    ]
}
```

**Note**

If the component detects a retryable error, such as a connection error, it retries the publish in the next batch.

Local log file

This component uses the following log file.

```
/greengrass/v2/logs/aws.greengrass.KinesisFirehose.log
```

**To view this component’s logs**

- Run the following command on the core device to view this component’s log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.KinesisFirehose.log
```

Licenses

This component includes the following third-party software/licensing:
This component is released under the Greengrass Core Software License Agreement.

**Changelog**

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.7</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.0.6</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.0.5</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>2.0.4</td>
<td>Version updated for Greengrass nucleus version 2.1.0 release.</td>
</tr>
<tr>
<td>2.0.3</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

**See also**

- What is Amazon Kinesis Data Firehose? in the Amazon Kinesis Data Firehose Developer Guide

**Lambda launcher**

The Lambda launcher component (aws.greengrass.LambdaLauncher) starts and stops AWS Lambda functions on AWS IoT Greengrass core devices. This component also sets up any containerization and runs processes as the users that you specify.

**Note**

When you deploy a Lambda function component to a core device, the deployment also includes this component. For more information, see Run AWS Lambda functions (p. 361).

**Topics**

- Versions (p. 189)
- Type (p. 189)
- Requirements (p. 189)
- Dependencies (p. 189)
- Configuration (p. 189)
- Local log file (p. 190)
- Changelog (p. 190)
Versions

This component has the following versions:

- 2.0.x

Type

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Requirements to run Lambda functions (p. 43).

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 190) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.0.4 - 2.0.8

The following table lists the dependencies for versions 2.0.4 to 2.0.8 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lambda manager (p. 190)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lambda manager (p. 190)</td>
<td>&gt;=2.0.3 &lt;2.1.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component doesn't have any configuration parameters.
Local log file

This component uses the following log file.

```
/greengrass/v2/logs/lambdaFunctionComponentName.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder, and replace `lambdaFunctionComponentName` with the name of the Lambda function component that this component launches.

```
sudo tail -f /greengrass/v2/logs/lambdaFunctionComponentName.log
```

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.8</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.0.7</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.0.6</td>
<td>General performance improvements and bug fixes.</td>
</tr>
<tr>
<td>2.0.4</td>
<td>Bug fixes and improvements</td>
</tr>
<tr>
<td></td>
<td>• Fixes an issue where the component doesn't correctly pass <code>AddGroupOwner</code> to the Lambda function container.</td>
</tr>
<tr>
<td>2.0.3</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

Lambda manager

The Lambda manager component (aws.greengrass.LambdaManager) manages work items and interprocess communication for AWS Lambda functions that run on the Greengrass core device.

**Note**

When you deploy a Lambda function component to a core device, the deployment also includes this component. For more information, see Run AWS Lambda functions (p. 361).

**Topics**

- Versions (p. 191)
- Type (p. 191)
- Requirements (p. 191)
- Dependencies (p. 191)
- Configuration (p. 192)
- Local log file (p. 192)
- Changelog (p. 193)
Versions

This component has the following versions:

- 2.1.x
- 2.0.x

Type

This component is a plugin component (aws.greengrass.plugin). The Greengrass nucleus (p. 136) runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Requirements to run Lambda functions (p. 43).

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 193) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>
2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.3 &lt;2.1.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

getResultTimeoutInSeconds

(Optional) The maximum amount of time in seconds that Lambda functions can run before they time out.

Default: 60

Local log file

This component uses the same log file as the Greengrass nucleus (p. 136) component.

/greengrass/v2/logs/greengrass.log

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace /greengrass/v2 with the path to the AWS IoT Greengrass root folder.

  sudo tail -f /greengrass/v2/logs/greengrass.log
Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.3</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>2.1.0</td>
<td>Version updated for Greengrass nucleus version 2.1.0 release.</td>
</tr>
<tr>
<td>2.0.3</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

Lambda runtimes

The Lambda runtimes component (aws.greengrass.LambdaRuntimes) provides the runtimes that Greengrass core devices use to run AWS Lambda functions.

Note
When you deploy a Lambda function component to a core device, the deployment also includes this component. For more information, see Run AWS Lambda functions (p. 361).

Topics
- Versions (p. 193)
- Type (p. 193)
- Requirements (p. 193)
- Dependencies (p. 194)
- Configuration (p. 194)
- Local log file (p. 194)
- Changelog (p. 194)

Versions

This component has the following versions:

- 2.0.x

Type

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:
• Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Requirements to run Lambda functions (p. 43).

Dependencies
This component doesn't have any dependencies.

Configuration
This component doesn't have any configuration parameters.

Local log file
This component doesn't output logs.

Changelog
The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.7</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.0.6</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.0.5</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>2.0.4</td>
<td>Version updated for Greengrass nucleus version 2.1.0 release.</td>
</tr>
<tr>
<td>2.0.3</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

Legacy subscription router
The legacy subscription router (aws.greengrass.LegacySubscriptionRouter) manages subscriptions on the Greengrass core device. Subscriptions are a feature of AWS IoT Greengrass V1 that define the topics that Lambda functions can use for MQTT messaging on a core device. For more information, see Managed subscriptions in the MQTT messaging workflow in the AWS IoT Greengrass V1 Developer Guide.

You can use this component to enable subscriptions for connector components and Lambda function components that use the AWS IoT Greengrass Core SDK.

Note
This component is required only if your Lambda function uses the publish() function in the AWS IoT Greengrass Core SDK. If you update your Lambda function code to use the interprocess communication (IPC) interface in the V2 AWS IoT Device SDK, you don't need to deploy the legacy subscription router component. For more information, see the following interprocess communication (p. 396) services:

• Publish/subscribe local messages (p. 410)
• Publish/subscribe AWS IoT Core MQTT messages (p. 433)
Versions

This component has the following versions:

- 2.1.x
- 2.0.x

Type

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 199) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.
### Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

**v2.1.x**

**subscriptions**

(Optional) The subscriptions to enable on the core device. This is an object, where each key is a unique ID, and each value is an object that defines the subscription for that connector. You must configure a subscription when you deploy a V1 connector component or a Lambda function that uses the AWS IoT Greengrass Core SDK.

Each subscription object contains the following information:

**id**

The unique ID of this subscription. This ID must match the key for this subscription object.

**source**

The Lambda function that uses the AWS IoT Greengrass Core SDK to publish MQTT messages on the topics that you specify in `subject`. Specify one of the following:

- The name of a Lambda function component on the core device. Specify the component name with the `component:` prefix, such as `component:com.example.HelloWorldLambda`.
- The Amazon Resource Name (ARN) of a Lambda function on the core device.

**Important**

If the version of the Lambda function changes, you must configure the subscription with the new version of the function. Otherwise, this component won't route the messages until the version matches the subscription.
You must specify an Amazon Resource Name (ARN) that includes the version of the function to import. You can't use version aliases like $LATEST.

To deploy a subscription for a V1 connector component, specify the name of the component or the ARN of the connector component's Lambda function.

subject

The MQTT topic or topic filter on which the source and target can publish and receive messages. This value supports the + and # topic wildcards.

target

The target that receives the MQTT messages on the topics that you specify in subject. The subscription specifies that the source function publishes MQTT messages to AWS IoT Core or to a Lambda function on the core device. Specify one of the following:

• cloud. The source function publishes MQTT messages to AWS IoT Core.
• The name of a Lambda function component on the core device. Specify the component name with the component: prefix, such as component:com.example.HelloWorldLambda.
• The Amazon Resource Name (ARN) of a Lambda function on the core device.

Important

If the version of the Lambda function changes, you must configure the subscription with the new version of the function. Otherwise, this component won't route the messages until the version matches the subscription. You must specify an Amazon Resource Name (ARN) that includes the version of the function to import. You can't use version aliases like $LATEST.

Default: No subscriptions

Example Example configuration update (defining a subscription to AWS IoT Core)

The following example specifies that the com.example.HelloWorldLambda Lambda function component publishes MQTT message to AWS IoT Core on the hello/world topic.

```json
{
   "subscriptions": {
      "Greengrass_HelloWorld_to_cloud": {
         "id": "Greengrass_HelloWorld_to_cloud",
         "source": "component:com.example.HelloWorldLambda",
         "subject": "hello/world",
         "target": "cloud"
      }
   }
}
```

Example Example configuration update (defining a subscription to another Lambda function)

The following example specifies that the com.example.HelloWorldLambda Lambda function component publishes MQTT messages to the com.example.MessageRelay Lambda function component on the hello/world topic.

```json
{
   "subscriptions": {
      "Greengrass_HelloWorld_to_MessageRelay": {
         "id": "Greengrass_HelloWorld_to_MessageRelay",
         "source": "component:com.example.HelloWorldLambda",
         "subject": "hello/world",
```
v2.0.x

subscriptions

(Optional) The subscriptions to enable on the core device. This is an object, where each key is a unique ID, and each value is an object that defines the subscription for that connector. You must configure a subscription when you deploy a V1 connector component or a Lambda function that uses the AWS IoT Greengrass Core SDK.

Each subscription object contains the following information:

id

The unique ID of this subscription. This ID must match the key for this subscription object.

source

The Lambda function that uses the AWS IoT Greengrass Core SDK to publish MQTT messages on the topics that you specify in subject. Specify the following:

• The Amazon Resource Name (ARN) of a Lambda function on the core device.
  
  Important
  
  If the version of the Lambda function changes, you must configure the subscription with the new version of the function. Otherwise, this component won't route the messages until the version matches the subscription. You must specify an Amazon Resource Name (ARN) that includes the version of the function to import. You can't use version aliases like $LATEST.

To deploy a subscription for a V1 connector component, specify the ARN of the connector component's Lambda function.

subject

The MQTT topic or topic filter on which the source and target can publish and receive messages. This value supports the + and # topic wildcards.

target

The target that receives the MQTT messages on the topics that you specify in subject. The subscription specifies that the source function publishes MQTT messages to AWS IoT Core or to a Lambda function on the core device. Specify one of the following:

• cloud. The source function publishes MQTT messages to AWS IoT Core.
  
  Important
  
  If the version of the Lambda function changes, you must configure the subscription with the new version of the function. Otherwise, this component won't route the messages until the version matches the subscription. You must specify an Amazon Resource Name (ARN) that includes the version of the function to import. You can't use version aliases like $LATEST.

Default: No subscriptions

Example Example configuration update (defining a subscription to AWS IoT Core)

The following example specifies that the Greengrass_HelloWorld function publishes MQTT message to AWS IoT Core on the hello/world topic.
"subscriptions": {
    "Greengrass_HelloWorld_to_cloud": {
        "id": "Greengrass_HelloWorld_to_cloud",
        "subject": "hello/world",
        "target": "cloud"
    }
}

**Example configuration update (defining a subscription to another Lambda function)**

The following example specifies that the Greengrass_HelloWorld function publishes MQTT messages to the Greengrass_MessageRelay on the hello/world topic.

```
"subscriptions": {
    "Greengrass_HelloWorld_to_MessageRelay": {
        "id": "Greengrass_HelloWorld_to_MessageRelay",
        "subject": "hello/world",
    }
}
```

**Local log file**

This component doesn't output logs.

**Changelog**

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.3</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
</tbody>
</table>
| 2.1.0   | Bug fixes and improvements  
|         | • Adds support to specify component names instead of ARNs for source and target. If you specify a component name for a subscription, you don't need to reconfigure the subscription each time the version of the Lambda function changes. |
| 2.0.3   | Initial version. |

**Local debug console**

The local debug console component (aws.greengrass.LocalDebugConsole) provides a local dashboard that displays information about your AWS IoT Greengrass core devices and its components. You can use this dashboard to debug your core device and manage local components.
Important
Don't use this component in production environments. This component is intended for use only in development environments. Anyone with access to the AWS IoT Greengrass CLI on the core device can access information and perform operations that this component exposes.

Topics
• Versions (p. 200)
• Type (p. 200)
• Requirements (p. 200)
• Dependencies (p. 200)
• Configuration (p. 202)
• Usage (p. 203)
• Local log file (p. 206)
• Changelog (p. 206)

Versions
This component has the following versions:
• 2.2.x
• 2.1.x
• 2.0.x

Type
This component is a plugin component (aws.greengrass.plugin). The Greengrass nucleus (p. 136) runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

For more information, see Component types (p. 323).

Requirements
This component has the following requirements:

• You use a user name and password to sign in to the dashboard. The username, which is debug, is provided for you. You must use the AWS IoT Greengrass CLI to create a temporary password that authenticates you with the dashboard on a core device. You must be able to use the AWS IoT Greengrass CLI to use the local debug console. For more information, see the Greengrass CLI requirements (p. 167). For more information about how to generate the password and sign in, see Local debug console component usage (p. 203).

Dependencies
When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released
dependencies (p. 206) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.2.2

The following table lists the dependencies for version 2.2.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.5.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Greengrass CLI (p. 166)</td>
<td>&gt;=2.1.0 &lt;2.5.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.2.1

The following table lists the dependencies for version 2.2.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.4.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Greengrass CLI (p. 166)</td>
<td>&gt;=2.1.0 &lt;2.4.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.2.0

The following table lists the dependencies for version 2.2.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.3.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Greengrass CLI (p. 166)</td>
<td>&gt;=2.1.0 &lt;2.3.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Greengrass CLI (p. 166)</td>
<td>&gt;=2.1.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.3 &lt;2.1.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>
## Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

### v2.1.x - v2.2.x

**httpsEnabled**

(Optional) You can enable HTTPS communication for the local debug console. If you enable HTTPS communication, the local debug console creates a self-signed certificate. Web browsers show security warnings for websites that use self-signed certificates, so you must manually verify the certificate. Then, you can bypass the warning. For more information, see Usage (p. 203).

Default: true

**port**

(Optional) The port at which to provide the local debug console.

Default: 1441

**websocketPort**

(Optional) The websocket port to use for the local debug console.

Default: 1442

### Example Example: Configuration merge update

The following example configuration specifies to open the local debug console on non-default ports and disable HTTPS.

```json
{
  "httpsEnabled": false,
  "port": "10441",
  "websocketPort": "10442"
}
```

### v2.0.x

**port**

(Optional) The port at which to provide the local debug console.

Default: 1441

**websocketPort**

(Optional) The websocket port to use for the local debug console.

Default: 1442

For more information about component dependencies, see the component recipe reference (p. 343).
Example Example: Configuration merge update

The following example configuration specifies to open the local debug console on non-default ports.

```
{
  "port": "10441",
  "websocketPort": "10442"
}
```

Usage

To use the local debug console, create a session from the Greengrass CLI. When you create a session, the Greengrass CLI provides a user name and temporary password that you can use to sign in to the local debug console.

Follow these instructions to open the local debug console on your core device or on your development computer.

v2.1.x - v2.2.x

In versions 2.1.0 and later, the local debug console uses HTTPS by default. When HTTPS is enabled, the local debug console creates a self-signed certificate to secure the connection. Your web browser shows a security warning when you open the local debug console because of this self-signed certificate. When you create a session with the Greengrass CLI, the output includes the certificate's fingerprints, so you can verify that the certificate is legitimate and the connection is secure.

You can disable HTTPS. For more information, see Local debug console configuration (p. 202).

To open the local debug console

1. (Optional) To view the local debug console on your development computer, you can forward the console's port over SSH. However, you must first enable the AllowTcpForwarding option in your core device's SSH configuration file. This option is enabled by default. Run the following command on your development computer to view the dashboard at http://localhost:1441 on your development computer.

   ```
   ssh -L 1441:localhost:1441 -L 1442:localhost:1442 username@core-device-ip-address
   ```

   **Note**
   You can change the default ports from 1441 and 1442. For more information, see Local debug console configuration (p. 202).

2. Create a session to use the local debug console. When you create a session, you generate a password that you use to authenticate. The local debug console requires a password to increase security, because you can use this component to view important information and perform operations on the core device. The local debug console also creates a certificate to secure the connection if you enable HTTPS in the component configuration. HTTPS is enabled by default.

   Use the AWS IoT Greengrass CLI to create the session. This command generates a random 43-character password that expires after 8 hours. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass V2 root folder.

   ```
   sudo /greengrass/v2/bin/greengrass-cli get-debug-password
   ```

   The command output looks like the following example if you have configured the local debug console to use HTTPS. You use the certificate fingerprints to verify that the connection is secure when you open the local debug console.
Username: debug
Password: bEDp3MOHdJ8oUw5de_sCBi2XaGyu3a8xRExample
Password expires at: 2021-04-01T17:01:43.92199931-07:00
The local debug console is configured to use TLS security. The certificate is self-signed so you will need to bypass your web browser's security warnings to open the console.

Before you bypass the security warning, verify that the certificate fingerprint matches the following fingerprints.

SHA-256: 15 0B 2C E2 54 8B 22 DE 08 46 54 8A B1 2B 25 DE FB 02 7D 01 4E 4A 56 67 96
DA 6C B1 D2 C4 1B
SHA-1: BC 3E 16 04 D3 80 70 DA E0 47 25 F9 90 FA D6 02 80 3E B5 C1

The debug view component creates a session that lasts for 8 hours. After that, you must generate a new password to view the local debug console again.

3. Open and sign in to the dashboard. View the dashboard on your Greengrass core device, or on your development computer if you forward the port over SSH. Do one of the following:
   - If you enabled HTTPS in the local debug console, which is the default setting, do the following:
     a. Open https://localhost:1441 on your core device, or on your development computer if you forwarded the port over SSH.
        
        Your browser might show a security warning about an invalid security certificate.
     b. If your browser shows a security warning, verify the certificate is legitimate and bypass the security warning. Do the following:
        i. Find the SHA-256 or SHA-1 fingerprint for the certificate, and verify that it matches the SHA-256 or SHA-1 fingerprint that the get-debug-password command previously printed. Your browser might provide one or both fingerprints. Consult your browser's documentation to view the certificate and its fingerprints. In some browsers, the certificate fingerprint is called a thumbprint.
   
   Note
   If the certificate fingerprint doesn't match, go to Step 2 (p. 203) to create a new session. If the certificate fingerprint still doesn't match, your connection might be insecure.
   
   i. If the certificate fingerprint matches, bypass your browser's security warning to open the local debug console. Consult your browser's documentation to bypass the browser security warning.
     c. Sign in to the website using the user name and password that the get-debug-password command printed earlier.
        The local debug console opens.
     d. If the local debug console shows an error that says it can't connect to the WebSocket due to a failed TLS handshake, you must bypass the self-signed security warning for the WebSocket URL.

   Error connecting to WebSocket
   The connection was closed due to a failure to perform a TLS handshake.
   Try opening https://localhost:1442 and bypass any warnings, then reload this page. The WebSocket connection uses the same certificate as this page.

   Do the following:
   i. Open https://localhost:1442 in the same browser where you opened the local debug console.
   ii. Verify the certificate and bypass the security warning.
Your browser might show an HTTP 404 page after you bypass the warning.


The local debug console shows information about the core device.

- If you disabled HTTPS in the local debug console, do the following:
  
a. Open http://localhost:1441 on your core device, or open it on your development computer if you forwarded the port over SSH.
  
b. Sign in to the website using the user name and password that the get-debug-password command previously printed.

The local debug console opens.

v2.0.x

To open the local debug console

1. (Optional) To view the local debug console on your development computer, you can forward the console's port over SSH. However, you must first enable the AllowTcpForwarding option in your core device's SSH configuration file. This option is enabled by default. Run the following command on your development computer to view the dashboard at http://localhost:1441 on your development computer.

   ssh -L 1441:localhost:1441 -L 1442:localhost:1442 username@core-device-ip-address

   Note
   You can change the default ports from 1441 and 1442. For more information, see Local debug console configuration (p. 202).

2. Create a session to use the local debug console. When you create a session, you generate a password that you use to authenticate. The local debug console requires a password to increase security, because you can use this component to view important information and perform operations on the core device.

   Use the AWS IoT Greengrass CLI to create the session. This command generates a random 43-character password that expires after 8 hours. Replace /greengrass/v2 with the path to the AWS IoT Greengrass V2 root folder.

   sudo /greengrass/v2/bin/greengrass-cli get-debug-password

   The command output looks like the following example.

   Username: debug
   Password: bEDp3MOHdj8ou2w5de_sCBI2XAguy3a8XxREXAMPLE
   Password will expire at: 2021-04-01T17:01:43.921999931-07:00

   The debug view component creates a session lasts for 4 hours, and then you must generate a new password to view the local debug console again.

3. Open http://localhost:1441 on your core device, or open it on your development computer if you forwarded the port over SSH.

4. Sign in to the website using the user name and password that the get-debug-password command previously printed.

   The local debug console opens.
Local log file

This component uses the same log file as the Greengrass nucleus (p. 136) component.

```
/greengrass/v2/logs/greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.2</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.2.0</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>2.1.0</td>
<td>New features</td>
</tr>
<tr>
<td></td>
<td>• Uses HTTPS to secure your connection to the local debug console. HTTPS is enabled by default.</td>
</tr>
<tr>
<td></td>
<td>• Bug fixes and improvements</td>
</tr>
<tr>
<td></td>
<td>• You can dismiss flashbar messages in the configuration editor.</td>
</tr>
<tr>
<td>2.0.3</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

Log manager

The log manager component (aws.greengrass.LogManager) uploads logs from AWS IoT Greengrass core devices to Amazon CloudWatch Logs. You can upload logs from the Greengrass nucleus, other Greengrass components, and other applications and services that aren't Greengrass components.

For information about the log groups and log streams to which this component uploads logs, see Usage (p. 214).

Topics

- Versions (p. 207)
- Type (p. 207)
- Requirements (p. 207)
- Dependencies (p. 208)
- Configuration (p. 209)
- Usage (p. 214)
- Local log file (p. 215)
- Changelog (p. 216)
Versions

This component has the following versions:

- 2.1.x
- 2.0.x

Type

This component is a plugin component (aws.greengrass.plugin). The Greengrass nucleus (p. 136) runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

- The Greengrass device role (p. 627) must allow the logs:CreateLogGroup, logs:CreateLogStream, logs:PutLogEvents, and logs:DescribeLogStreams actions, as shown in the following example IAM policy.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents",
                "logs:DescribeLogStreams"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

**Note**

The Greengrass device role (p. 627) that you create when you install the AWS IoT Greengrass Core software includes the permissions in this example policy by default.

For more information, see Using identity-based policies (IAM policies) for CloudWatch Logs in the Amazon CloudWatch Logs User Guide.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall (p. 640).
Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 216) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.1.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>
2.0.x

The following table lists the dependencies for version 2.0.x of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.3 &lt;2.1.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

**Configuration**

This component provides the following configuration parameters that you can customize when you deploy the component.

**2.1.x**

logsUploaderConfiguration

(Optional) The configuration for logs that the log manager component uploads. This object contains the following information:

systemLogsConfiguration

(Optional) The configuration for AWS IoT Greengrass Core software system logs. Specify this configuration to enable the log manager component to manage system logs. This object contains the following information:

uploadToCloudWatch

(Optional) You can upload system logs to CloudWatch Logs.

Default: false

minimumLogLevel

(Optional) The minimum level of information to upload. Choose from the following log levels, listed here in level order:

- DEBUG
- INFO
- WARN
- ERROR

Default: INFO

diskSpaceLimit

(Optional) The maximum total size of all system log files, in the unit you specify in diskSpaceLimitUnit. After the total size of all log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes the oldest log files.

This parameter is equivalent to the system logs disk space limit (p. 142) parameter (totalLogsSizeKB) of the Greengrass nucleus component (p. 136). If you specify this parameter on both components, the AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total log size.

diskSpaceLimitUnit

(Optional) The unit for the diskSpaceLimit. Choose from the following options:

- KB – kilobytes
- MB – megabytes
• GB – gigabytes
  Default: KB

deleteLogFileAfterCloudUpload

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

  Default: false

cOMPONENTLOGSConfiguration

(Optional) A list of log configurations for components on the core device. Each configuration in this list defines the log configuration for a component or application. The log manager component uploads these component logs to CloudWatch Logs.

Each object contains the following information:

  componentName

  The name of the component for this logs configuration. You can specify the name of a Greengrass component or another value to identify this log group.

  minimumLogLevel

  (Optional) The minimum level of information to upload. Choose from the following log levels, listed here in level order:

  • DEBUG
  • INFO
  • WARN
  • ERROR

  Default: INFO

logFileDirectoryPath

(Optional) The path to the folder that contains this component's log files.

You don’t need to specify this parameter for Greengrass components that print to standard output (stdout) and standard error (stderr).

  Default: /greengrass/v2/logs.

logFileRegex

(Optional) A regular expression that specifies the log file name format that the component or application uses. The log manager component uses this regular expression to identify log files in the folder at logFileDirectoryPath.

You don’t need to specify this parameter for Greengrass components that print to standard output (stdout) and standard error (stderr).

If your component or application rotates log files, specify a regex that matches the rotated log file names. For example, you might specify hello_world\w*.log to upload logs for a Hello World application. The \w* pattern matches zero or more word characters, which includes alphanumeric characters and underscores. This regex matches log files with and without timestamps in their name. In this example, the log manager uploads the following log files:

  • hello_world.log – The most recent log file for the Hello World application.
  • hello_world_2020_12_15_17_0.log – An older log file for the Hello World application.

  Default: componentName\w*.log, where componentName is the name of the component for this log configuration.
diskSpaceLimit

(Optional) The maximum total size of all log files for this component, in the unit you specify in diskSpaceLimitUnit. After the total size of this component's log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes the oldest log files.

diskSpaceLimitUnit

(Optional) The unit for the diskSpaceLimit. Choose from the following options:
- KB – kilobytes
- MB – megabytes
- GB – gigabytes

Default: KB

deleteLogFileAfterCloudUpload

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: false

periodicUploadIntervalSec

(Optional) The period in seconds at which the log manager component checks for new log files to upload.

Default: 300 (5 minutes)

Example: Configuration merge update

The following example configuration specifies to upload system logs and com.example.HelloWorld component logs to CloudWatch Logs.

```json
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true",
      "minimumLogLevel": "INFO",
      "diskSpaceLimit": "10",
      "diskSpaceLimitUnit": "MB",
      "deleteLogFileAfterCloudUpload": "false"
    },
    "componentLogsConfiguration": [
      {
        "componentName": "com.example.HelloWorld",
        "minimumLogLevel": "INFO",
        "diskSpaceLimit": "10",
        "diskSpaceLimitUnit": "KB",
        "deleteLogFileAfterCloudUpload": "false"
      }
    ],
    "periodicUploadIntervalSec": "300"
  }
}
```

v2.0.x

logsUploaderConfiguration

(Optional) The configuration for logs that the log manager component uploads. This object contains the following information:
sysToemLogsConfiguration

(Optional) The configuration for AWS IoT Greengrass Core software system logs. Specify this configuration to enable the log manager component to manage system logs. This object contains the following information:

uploadToCloudWatch

(Optional) You can upload system logs to CloudWatch Logs.

Default: false

minimumLogLevel

(Optional) The minimum level of information to upload. Choose from the following log levels, listed here in level order:

- DEBUG
- INFO
- WARN
- ERROR

Default: INFO

diskSpaceLimit

(Optional) The maximum total size of all system log files, in the unit you specify in diskSpaceLimitUnit. After the total size of all log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes the oldest log files.

This parameter is equivalent to the system logs disk space limit (p. 142) parameter (totalLogsSizeKB) of the Greengrass nucleus component (p. 136). If you specify this parameter on both components, the AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total log size.

diskSpaceLimitUnit

(Optional) The unit for the diskSpaceLimit. Choose from the following options:

- KB – kilobytes
- MB – megabytes
- GB – gigabytes

Default: KB

deleteLogFileAfterCloudUpload

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: false

componentLogsConfiguration

(Optional) A list of log configurations for components on the core device. Each configuration in this list defines the log configuration for a component or application. The log manager component uploads these component logs to CloudWatch Logs

Each object contains the following information:

cOMPONENTNAME

The name of the component for this logs configuration. You can specify the name of a Greengrass component or another value to identify this log group.
minimumLogLevel

(Optional) The minimum level of information to upload. Choose from the following log levels, listed here in level order:

- DEBUG
- INFO
- WARN
- ERROR

Default: INFO

logFileDirectoryPath

The path to the folder that contains this component's log files.

To upload a Greengrass component's logs, specify \greengrass/v2/logs, and replace \greengrass/v2 with your Greengrass root folder.

logFileRegex

A regular expression that specifies the log file name format that the component or application uses. The log manager component uses this regular expression to identify log files in the folder at logFileDirectoryPath.

To upload a Greengrass component's logs, specify \com.example.HelloWorld\w*\.log to upload logs for a Hello World component. The \w* pattern matches zero or more word characters, which includes alphanumeric characters and underscores. This regex matches log files with and without timestamps in their name. In this example, the log manager uploads the following log files:

- com.example.HelloWorld_2020_12_15_17_0.log – An older log file for the Hello World component. The Greengrass nucleus adds a rotating timestamp to the log files.

diskSpaceLimit

(Optional) The maximum total size of all log files for this component, in the unit you specify in diskSpaceLimitUnit. After the total size of this component's log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes the oldest log files.

diskSpaceLimitUnit

(Optional) The unit for the diskSpaceLimit. Choose from the following options:

- KB – kilobytes
- MB – megabytes
- GB – gigabytes

Default: KB

deleteLogFileAfterCloudUpload

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: false

periodicUploadIntervalSec

(Optional) The period in seconds at which the log manager component checks for new log files to upload.
Example Example: Configuration merge update

The following example configuration specifies to upload system logs and `com.example.HelloWorld` component logs to CloudWatch Logs.

```json
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true",
      "minimumLogLevel": "INFO",
      "diskSpaceLimit": "10",
      "diskSpaceLimitUnit": "MB",
      "deleteLogFileAfterCloudUpload": "false"
    },
    "componentLogsConfiguration": [
      {
        "componentName": "com.example.HelloWorld",
        "minimumLogLevel": "INFO",
        "logFileDirectoryPath": "/greengrass/v2/logs",
        "logFileRegex": "com.example.HelloWorld\w*.log",
        "diskSpaceLimit": "10",
        "diskSpaceLimitUnit": "KB",
        "deleteLogFileAfterCloudUpload": "false"
      }
    ],
    "periodicUploadIntervalSec": "300"
  }
}
```

Usage

The log manager component uploads to the following log groups and log streams.

2.1.x

Log group name

/aws/greengrass/componentType/region/componentName

The log group name uses the following variables:

- componentType – The type of the component, which can be one of the following:
  - GreengrassSystemComponent – The component is part of the Greengrass nucleus (p. 136).
  - UserComponent – The component isn't part of the Greengrass nucleus. The log manager uses this type for Greengrass components and other applications on the device.
- region – The AWS Region that the core device uses.
- componentName – The name of the component. For system logs, this value is System.

Log stream name

/date/thing/thingName

The log stream name uses the following variables:
Local log file

This component uses the same log file as the Greengrass nucleus (p. 136) component.

```
/greengrass/v2/logs/greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time.
  Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
```
Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.3</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
</tbody>
</table>
| 2.1.1   | Bug fixes and improvements  
• Fixes an issue where the system log configuration wasn't updated in certain cases. |
| 2.1.0   | Bug fixes and improvements  
• Use defaults for `logFileDirectoryPath` and `logFileRegex` that work for Greengrass components that print to standard output (stdout) and standard error (stderr).  
• Correctly route traffic through a configured network proxy when uploading logs to CloudWatch Logs.  
• Correctly handle colon characters (:) in log stream names. CloudWatch Logs log stream names don't support colons.  
• Simplify log stream names by removing thing group names from the log stream.  
• Remove an error log message that prints during normal behavior. |
| 2.0.x   | Initial version. |

Machine learning components

AWS IoT Greengrass provides the following machine learning components that you can deploy to supported devices to perform machine learning inference (p. 558) using models trained in Amazon SageMaker or with your own pre-trained models that are stored in Amazon S3.

AWS-provided machine learning components are broadly categorized as follows:

- **Model component**—Contains machine learning models as Greengrass artifacts.
- **Runtime component**—Contains the script that installs the machine learning framework and its dependencies on the Greengrass core device.
- **Inference component**—Contains the inference code and includes component dependencies to install the machine learning framework and download pre-trained machine learning models.

You can use the sample inference code and pre-trained models in the AWS-provided machine learning components to perform image classification and object detection using DLR and TensorFlow Lite. To perform custom machine learning inference with your own models that are stored in Amazon S3, or to use a different machine learning framework, you can use the recipes of these public components as templates to create custom machine learning components. For more information, see Customize your machine learning components (p. 581).
AWS IoT Greengrass also includes an AWS-provided component to manage the installation and lifecycle of the SageMaker Edge Manager agent on Greengrass core devices. With SageMaker Edge Manager, you can use Amazon SageMaker Neo-compiled models directly on your core device. For more information, see Use Amazon SageMaker Edge Manager on Greengrass core devices (p. 573).

The following table lists the machine learning components that are available in AWS IoT Greengrass.

**Note**
Several AWS-provided components depend on specific minor versions of the Greengrass nucleus. Because of this dependency, you need to update these components when you update the Greengrass nucleus to a new minor version. For information about the specific versions of the nucleus that each component depends on, see the corresponding component topic. For more information about updating the nucleus, see Update the AWS IoT Greengrass Core software (OTA) (p. 129).

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Depends on nucleus</th>
<th>Component type (p. 323)</th>
<th>Open source (p. 761)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SageMaker Edge Manager (p. 220)</td>
<td>Deploys the Amazon SageMaker Edge Manager agent on the Greengrass core device.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>DLR image classification (p. 226)</td>
<td>Inference component that uses the DLR image classification model store and the DLR runtime component as dependencies to install DLR, download sample image classification models, and perform image classification inference on supported devices.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>DLR object detection (p. 232)</td>
<td>Inference component that uses the DLR object detection model store and the DLR runtime component as dependencies to install DLR, download sample object detection</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
</tbody>
</table>
# Machine learning components

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Depends on nucleus</th>
<th>Component type (p. 323)</th>
<th>Open source (p. 761)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLR image classification model store (p. 238)</td>
<td>Model component that contains sample ResNet-50 image classification models as Greengrass artifacts.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>DLR object detection model store (p. 241)</td>
<td>Model component that contains sample YOLOv3 object detection models as Greengrass artifacts.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>DLR installer (p. 243)</td>
<td>Runtime component that contains an installation script that is used to install DLR and its dependencies on the Greengrass core device.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
</tbody>
</table>
## Machine learning components

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Depends on nucleus</th>
<th>Component type (p. 323)</th>
<th>Open source (p. 761)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow Lite image classification (p. 247)</td>
<td>Inference component that uses the TensorFlow Lite image classification model store and the TensorFlow Lite runtime component as dependencies to install TensorFlow Lite, download sample image classification models, and perform image classification inference on supported devices.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>TensorFlow Lite object detection (p. 252)</td>
<td>Inference component that uses the TensorFlow Lite object detection model store and the TensorFlow Lite runtime component as dependencies to install TensorFlow Lite, download sample object detection models, and perform object detection inference on supported devices.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>TensorFlow Lite image classification model store (p. 256)</td>
<td>Model component that contains a sample MobileNet v1 model as a Greengrass artifact.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
</tbody>
</table>
### SageMaker Edge Manager

The Amazon SageMaker Edge Manager component (`aws.greengrass.SageMakerEdgeManager`) installs the SageMaker Edge Manager agent binary.

SageMaker Edge Manager provides model management for edge devices so you can optimize, secure, monitor, and maintain machine learning models on fleets of edge devices. The SageMaker Edge Manager component installs and manages the lifecycle of the SageMaker Edge Manager agent on your core device. You can also use SageMaker Edge Manager to package and use SageMaker Neo-compiled models as model components on Greengrass core devices. For more information about using SageMaker Edge Manager agent on your core device, see Use Amazon SageMaker Edge Manager on Greengrass core devices (p. 573).

SageMaker Edge Manager component v1.0.x installs Edge Manager agent binary v1.20210512.96da6cc. For more information about Edge Manager agent binary versions, see Edge Manager Agent.

**Note**
The SageMaker Edge Manager component is available only in the following AWS Regions:

- US East (Ohio)
- US East (N. Virginia)
- US West (Oregon)
- EU (Frankfurt)
- EU (Ireland)
- Asia Pacific (Tokyo)

**Topics**

- Versions (p. 221)
• Type (p. 221)
• Requirements (p. 221)
• Dependencies (p. 222)
• Configuration (p. 223)
• Local log file (p. 225)
• Changelog (p. 226)

Versions
This component has the following versions:

• 1.0.x

Type
This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Requirements
This component has the following requirements:

• A Greengrass core device running on a Debian-based Linux platform (x86_64 or Armv8). If you don't have one, see Getting started with AWS IoT Greengrass V2 (p. 24).
• Python 3.6 or later, including pip for your version of Python, installed on your core device.
• The Greengrass device role (p. 627) configured with the following:
  • A trust relationship that allows credentials.iot.amazonaws.com and sagemaker.amazonaws.com to assume the role, as shown in the following IAM policy example.

```
{
   "Version": "2012-10-17",
   "Statement": [  
       {  
           "Effect": "Allow",
           "Principal": {  
               "Service": "credentials.iot.amazonaws.com"
           },  
           "Action": "sts:AssumeRole"
       },  
       {  
           "Effect": "Allow",
           "Principal": {  
               "Service": "sagemaker.amazonaws.com"
           },  
           "Action": "sts:AssumeRole"
       }
   ]
}
```

• The AmazonSageMakerEdgeDeviceFleetPolicy IAM managed policy.
• The s3:PutObject action, as shown in the following IAM policy example.

```
{
   "Version": "2012-10-17",
   "Statement": [  
       {  
           "Effect": "Allow",
           "Principal": {  
               "Service": "sagemaker.amazonaws.com"
           },  
           "Action": "s3:PutObject"
       }
   ]
}
```
"Statement": [
  {
    "Action": [
      "s3:PutObject"
    ],
    "Resource": [
      "*
    ],
    "Effect": "Allow"
  }
]

• An Amazon S3 bucket created in the same AWS account and AWS Region as your Greengrass core device. SageMaker Edge Manager requires an S3 bucket to create an edge device fleet, and to store sample data from running inference on your device. For information about creating S3 buckets, see Getting started with Amazon S3.

• A SageMaker edge device fleet that uses the same AWS IoT role alias as your Greengrass core device. For more information, see Create an edge device fleet (p. 577).

• Your Greengrass core device registered as an edge device in your SageMaker Edge device fleet. The edge device name must match the AWS IoT thing name for your core device. For more information, see Register your Greengrass core device (p. 577).

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall (p. 640).

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>edge.sagemaker.region.amazonaws.com</code></td>
<td>443</td>
<td>Yes</td>
<td>Check device registration status and send metrics to SageMaker.</td>
</tr>
<tr>
<td><code>*.s3.amazonaws.com</code></td>
<td>443</td>
<td>Yes</td>
<td>Upload capture data to the S3 bucket that you specify. You can replace * with the name of each bucket where you upload data.</td>
</tr>
</tbody>
</table>

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its...
dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 226) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

1.0.3

The following table lists the dependencies for version 1.0.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>&gt;=0.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

1.0.1 - 1.0.2

The following table lists the dependencies for versions 1.0.1 and 1.0.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>&gt;=0.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

1.0.0

The following table lists the dependencies for version 1.0.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>&gt;=0.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

Note

This section describes the configuration parameters that you set in the component. For more information about the corresponding SageMaker Edge Manager configuration, see Edge Manager Agent in the Amazon SageMaker Developer Guide.

DeviceFleetName

The name of the SageMaker Edge Manager device fleet that contains your Greengrass core device.
You must specify a value for this parameter in the configuration update when you deploy this component.

**BucketName**

The name of the S3 bucket to which you upload captured inference data. The bucket name must contain the string `sagemaker`.

If you set `CaptureDataDestination` to `Cloud`, or if you set `CaptureDataPeriodicUpload` to `true`, then you must specify a value for this parameter in the configuration update when you deploy this component.

**Note**

Capture data is an SageMaker feature that you use to upload inference input, inference results, and additional inference data to an S3 bucket or a local directory for future analysis. For more information about using capture data with SageMaker Edge Manager, see Manage Model in the Amazon SageMaker Developer Guide.

**CaptureDataBatchSize**

(Optional) The size of a batch of capture data requests that the agent handles. This value must be less than the buffer size that you specify in `CaptureDataBufferSize`. We recommend that you don't exceed half the buffer size.

The agent handles a request batch when the number of requests in the buffer meets the `CaptureDataBatchSize` number, or when the `CaptureDataPushPeriodSeconds` interval elapses, whichever occurs first.

Default: 10

**CaptureDataBufferSize**

(Optional) The maximum number of capture data requests stored in the buffer.

Default: 30

**CaptureDataDestination**

(Optional) The destination where you store captured data. This parameter can have the following values:

- **Disk**— Writes captured data to the component's work directory.
- **Cloud**— Uploads captured data to the S3 bucket that you specify in `BucketName`.

If you specify `Disk`, you can also choose to periodically upload the captured data to your S3 bucket by setting `CaptureDataPeriodicUpload` to `true`.

Default: `Cloud`

**CaptureDataPeriodicUpload**

(Optional) String value that specifies whether to periodically upload captured data. Supported values are `true` and `false`.

Set this parameter to `true` if you set `CaptureDataDestination` to `Disk`, and you also want the agent to periodically upload the captured data to your S3 bucket.

Default: `false`

**CaptureDataPeriodicUploadPeriodSeconds**

(Optional) The interval in seconds at which SageMaker Edge Manager agent uploads captured data to the S3 bucket. Use this parameter if you set `CaptureDataPeriodicUpload` to `true`.

Default: 8
CaptureDataPushPeriodSeconds

(Optional) The interval in seconds at which SageMaker Edge Manager agent handles a batch of capture data requests from the buffer.

The agent handles a request batch when the number of requests in the buffer meets the CaptureDataBatchSize number, or when the CaptureDataPushPeriodSeconds interval elapses, whichever occurs first.

Default: 4

CaptureDataBase64EmbedLimit

(Optional) The maximum size in bytes of captured data that SageMaker Edge Manager agent uploads.

Default: 3072

FolderPrefix

(Optional) The name of the folder to which the agent writes the captured data. If you set CaptureDataDestination to Disk, the agent creates the folder in the component's work directory. If you set CaptureDataDestination to Cloud, or if you set CaptureDataPeriodicUpload to true, the agent creates the folder in your S3 bucket.

Default: sme-capture

SagemakerEdgeLogVerbose

(Optional) String value that specifies whether to enable debug logging. Supported values are true and false.

Default: false

UnixSocketName

(Optional) The location of the SageMaker Edge Manager socket file descriptor on the core device.

Default: /tmp/aws.greengrass.SageMakerEdgeManager.sock

Example Example: Configuration merge update

The following example configuration specifies that the core device is part of the MyEdgeDeviceFleet and that the agent writes capture data both to the device and to an S3 bucket. This configuration also enables debug logging.

```json
{
  "DeviceFleetName": "MyEdgeDeviceFleet",
  "BucketName": "DOC-EXAMPLE-BUCKET",
  "CaptureDataDestination": "Disk",
  "CaptureDataPeriodicUpload": "true",
  "SagemakerEdgeLogVerbose": "true"
}
```

Local log file

This component uses the following log file.

```
/greengrass/v2/logs/aws.greengrass.SageMakerEdgeManager.log
```
To view this component’s logs

- Run the following command on the core device to view this component’s log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```bash
sudo tail -f /greengrass/v2/logs/aws.greengrass.SageMakerEdgeManager.log
```

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0.3</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>1.0.2</td>
<td>Bug fixes and improvements</td>
</tr>
<tr>
<td></td>
<td>Updates the installation script in the component lifecycle. Your core devices must now have Python 3.6 or later, including <code>pip</code> for your version of Python, installed on the device before you deploy this component.</td>
</tr>
<tr>
<td>1.0.1</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>1.0.0</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

DLR image classification

The DLR image classification component (aws.greengrass.DLRImageClassification) contains sample inference code to perform image classification inference using Deep Learning Runtime and resnet-50 models. This component uses the variant DLR image classification model store (p. 238) and the DLR installer (p. 243) components as dependencies to download DLR and the sample models.

To use this inference component with a custom-trained DLR model, create a custom version (p. 583) of the dependent model store component. To use your own custom inference code, you can use the recipe of this component as a template to create a custom inference component (p. 587).

Topics

- Versions (p. 226)
- Type (p. 227)
- Requirements (p. 227)
- Dependencies (p. 227)
- Configuration (p. 228)
- Local log file (p. 231)
- Changelog (p. 231)

Versions

This component has the following versions:

- 2.1.x
• 2.0.x

**Type**

This component is a generic component (aws.greengrass.generic). The *Greengrass nucleus* (p. 136) runs the component's lifecycle scripts.

For more information, see *Component types (p. 323).*

**Requirements**

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, *GNU C Library* (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies:

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

**Dependencies**

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 231) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the *AWS IoT Greengrass console.* On the component details page, look for the *Dependencies* list.

**2.1.4 - 2.1.5**

The following table lists the dependencies for versions 2.1.4 to 2.1.5 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
<tr>
<td>DLR image classification model store (p. 238)</td>
<td>~2.1.0</td>
<td>Hard</td>
</tr>
<tr>
<td>DLR (p. 243)</td>
<td>~1.6.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

**2.1.3**

The following table lists the dependencies for version 2.1.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
<tr>
<td>DLR image classification model store (p. 238)</td>
<td>~2.1.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>
### Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

#### 2.1.x

**accessControl**

(Optional) The object that contains the *authorization policy (p. 403)* that allows the component to publish messages to the default notifications topic.
Default:

```
{
  "aws.greengrass.ipc.mqttproxy": {
    "aws.greengrass.DLRImageClassification:mqttproxy:1": {
      "policyDescription": "Allows access to publish via topic ml/dlr/image-classification.",
      "operations": [
        "aws.greengrass#PublishToIoTCore"
      ],
      "resources": [
        "ml/dlr/image-classification"
      ]
    }
  }
}
```

**PublishResultsOnTopic**

(Optional) The topic on which you want to publish the inference results. If you modify this value, then you must also modify the value of resources in the accessControl parameter to match your custom topic name.

Default: ml/dlr/image-classification

**Accelerator**

The accelerator that you want to use. Supported values are `cpu` and `gpu`.

The sample models in the dependent model component support only CPU acceleration. To use GPU acceleration with a different custom model, create a custom model component (p. 583) to override the public model component.

Default: cpu

**ImageDirectory**

(Optional) The path of the folder on the device where inference components read images. You can modify this value to any location on your device to which you have read/write access.

Default: /greengrass/v2/packages/artifacts-unarchived/component-name/image_classification/sample_images/

*Note*

If you set the value of `UseCamera` to true, then this configuration parameter is ignored.

**ImageName**

(Optional) The name of the image that the inference component uses as an input to a make prediction. The component looks for the image in the folder specified in `ImageDirectory`. By default, the component uses the sample image in the default image directory. AWS IoT Greengrass supports the following image formats: jpeg, jpg, png, and npy.

Default: cat.jpeg

*Note*

If you set the value of `UseCamera` to true, then this configuration parameter is ignored.

**InferenceInterval**

(Optional) The time in seconds between each prediction made by the inference code. The sample inference code runs indefinitely and repeats its predictions at the specified time interval.
For example, you can change this to a shorter interval if you want to use images taken by a camera for real-time prediction.

Default: 3600

ModelResourceKey

(Optional) The models that are used in the dependent public model component. Modify this parameter only if you override the public model component with a custom component.

Default:

```
{
    "armv7l": "DLR-resnet50-armv7l-cpu-ImageClassification",
    "aarch64": "DLR-resnet50-aarch64-cpu-ImageClassification",
    "x86_64": "DLR-resnet50-x86_64-cpu-ImageClassification"
}
```

UseCamera

(Optional) String value that defines whether to use images from a camera connected to the Greengrass core device. Supported values are true and false.

When you set this value to true, the sample inference code accesses the camera on your device and runs inference locally on the captured image. The values of the ImageName and ImageDirectory parameters are ignored. Make sure that the user running this component has read/write access to the location where the camera stores captured images.

Default: false

Note

When you view the recipe of this component, the UseCamera configuration parameter doesn't appear in the default configuration. However, you can modify the value of this parameter in a configuration merge update (p. 390) when you deploy the component. When you set UseCamera to true, you must also create a symlink to enable the inference component to access your camera from the virtual environment that is created by the runtime component. For more information about using a camera with the sample inference components, see Update component configurations (p. 568).

2.0.x

MLRootPath

(Optional) The path of the folder on the device where inference components read images and write inference results. You can modify this value to any location on your device to which the user running this component has read/write access.

Default: /greengrass/v2/work/variant.DLR/greengrass_ml

Default: /greengrass/v2/work/variant.TensorFlowLite/greengrass_ml

Accelerator

The accelerator that you want to use. Supported values are cpu and gpu.

The sample models in the dependent model component support only CPU acceleration. To use GPU acceleration with a different custom model, create a custom model component (p. 583) to override the public model component.

Default: cpu
**ImageName**

(Optional) The name of the image that the inference component uses as an input to a make prediction. The component looks for the image in the folder specified in `ImageDirectory`. The default location is `MLRootPath/images`. AWS IoT Greengrass supports the following image formats: jpeg, jpg, png, and npy.

Default: cat.jpeg

**InferenceInterval**

(Optional) The time in seconds between each prediction made by the inference code. The sample inference code runs indefinitely and repeats its predictions at the specified time interval. For example, you can change this to a shorter interval if you want to use images taken by a camera for real-time prediction.

Default: 3600

**ModelResourceKey**

(Optional) The models that are used in the dependent public model component. Modify this parameter only if you override the public model component with a custom component.

Default:

```
armv7l: "DLR-resnet50-armv7l-cpu-ImageClassification"
x86_64: "DLR-resnet50-x86_64-cpu-ImageClassification"
```

**Local log file**

This component uses the following log file.

```
/greengrass/v2/logs/aws.greengrass.DLRImageClassification.log
```

**To view this component's logs**

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.DLRImageClassification.log
```

**Changelog**

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.5</td>
<td>Component released in all AWS Regions.</td>
</tr>
</tbody>
</table>
| 2.1.4   | Version updated for Greengrass nucleus version 2.4.0 release.  
This version isn't available in Europe (London) (eu-west-2). |
| 2.1.3   | Version updated for Greengrass nucleus version 2.3.0 release. |
| 2.1.2   | Version updated for Greengrass nucleus version 2.2.0 release. |
## DLR object detection

The DLR object detection component (aws.greengrass.DLRObjectDetection) contains sample inference code to perform object detection inference using Deep Learning Runtime and sample pre-trained models. This component uses the variant DLR object detection model store (p. 241) and the DLR installer (p. 243) components as dependencies to download DLR and the sample models.

To use this inference component with a custom-trained DLR model, create a custom version (p. 583) of the dependent model store component. To use your own custom inference code, you can use the recipe of this component as a template to create a custom inference component (p. 587).

### Topics
- Versions (p. 232)
- Type (p. 233)
- Requirements (p. 233)
- Dependencies (p. 233)
- Configuration (p. 234)
- Local log file (p. 237)
- Changelog (p. 237)

### Versions

This component has the following versions:

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.1</td>
<td>New features</td>
</tr>
<tr>
<td></td>
<td>• Use Deep Learning Runtime v1.6.0.</td>
</tr>
<tr>
<td></td>
<td>• Add support for sample image classification on Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano.</td>
</tr>
<tr>
<td></td>
<td>• Enable camera integration for sample inference. Use the new UseCamera configuration parameter to enable the sample inference code to access the camera on your Greengrass core device and run inference locally on the captured image.</td>
</tr>
<tr>
<td></td>
<td>• Add support for publishing inference results to the AWS Cloud. Use the new PublishResultsOnTopic configuration parameter to specify the topic on which you want to publish results.</td>
</tr>
<tr>
<td></td>
<td>• Add the new ImageDirectory configuration parameter that enables you to specify a custom directory for the image on which you want to perform inference.</td>
</tr>
<tr>
<td></td>
<td>Bug fixes and improvements</td>
</tr>
<tr>
<td></td>
<td>• Write inference results to the component log file instead of a separate inference file.</td>
</tr>
<tr>
<td></td>
<td>• Use the AWS IoT Greengrass Core software logging module to log component output.</td>
</tr>
<tr>
<td></td>
<td>• Use the AWS IoT Device SDK to read the component configuration and apply configuration changes.</td>
</tr>
<tr>
<td>2.0.4</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>
Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies:

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 237) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.1.4 - 2.1.5

The following table lists the dependencies for versions 2.1.4 to 2.1.5 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
<tr>
<td>DLR object detection model store (p. 241)</td>
<td>~2.1.0</td>
<td>Hard</td>
</tr>
<tr>
<td>DLR (p. 243)</td>
<td>~1.6.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
<tr>
<td>DLR object detection model store (p. 241)</td>
<td>~2.1.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>
### DLR object detection

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLR (p. 243)</td>
<td>~1.6.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

#### 2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
<tr>
<td>DLR object detection model store (p. 241)</td>
<td>~2.1.0</td>
<td>Hard</td>
</tr>
<tr>
<td>DLR (p. 243)</td>
<td>~1.6.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

#### 2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
<tr>
<td>DLR object detection model store (p. 241)</td>
<td>~2.1.0</td>
<td>Hard</td>
</tr>
<tr>
<td>DLR (p. 243)</td>
<td>~1.6.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

#### 2.0.x

The following table lists the dependencies for version 2.0.x of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>~2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>DLR object detection model store</td>
<td>~2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>DLR</td>
<td>~1.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

### Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

#### 2.1.x

`accessControl`

(Optional) The object that contains the authorization policy (p. 403) that allows the component to publish messages to the default notifications topic.
AWS IoT Greengrass Developer Guide, Version 2

DLR object detection

Default:

```
{
  "aws.greengrass.ipc.mqttproxy": {
    "aws.greengrass.DLRObjectDetection:mqttproxy:1": {
      "policyDescription": "Allows access to publish via topic ml/dlr/object-detection.",
      "operations": [
        "aws.greengrass#PublishToIoTCore"
      ],
      "resources": [
        "ml/dlr/object-detection"
      ]
    }
  }
}
```

PublishResultsOnTopic

(Optional) The topic on which you want to publish the inference results. If you modify this value, then you must also modify the value of resources in the accessControl parameter to match your custom topic name.

Default: ml/dlr/object-detection

Accelerator

The accelerator that you want to use. Supported values are cpu and gpu.

The sample models in the dependent model component support only CPU acceleration. To use GPU acceleration with a different custom model, create a custom model component (p. 583) to override the public model component.

Default: cpu

ImageDirectory

(Optional) The path of the folder on the device where inference components read images. You can modify this value to any location on your device to which you have read/write access.

Default: /greengrass/v2/packages/artifacts-unarchived/component-name/object_detection/sample_images/

Note

If you set the value of UseCamera to true, then this configuration parameter is ignored.

ImageName

(Optional) The name of the image that the inference component uses as an input to a make prediction. The component looks for the image in the folder specified in ImageDirectory. By default, the component uses the sample image in the default image directory. AWS IoT Greengrass supports the following image formats: jpeg, jpg, png, and npy.

Default: objects.jpg

Note

If you set the value of UseCamera to true, then this configuration parameter is ignored.

InferenceInterval

(Optional) The time in seconds between each prediction made by the inference code. The sample inference code runs indefinitely and repeats its predictions at the specified time interval.
For example, you can change this to a shorter interval if you want to use images taken by a camera for real-time prediction.

Default: 3600

ModelResourceKey

(Optional) The models that are used in the dependent public model component. Modify this parameter only if you override the public model component with a custom component.

Default:

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Model Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>armv7l</td>
<td>&quot;DLR-yolo3-armv7l-cpu-ObjectDetection&quot;</td>
</tr>
<tr>
<td>x86_64</td>
<td>&quot;DLR-yolo3-x86_64-cpu-ObjectDetection&quot;</td>
</tr>
<tr>
<td>aarch64</td>
<td>&quot;DLR-yolo3-aarch64-gpu-ObjectDetection&quot;</td>
</tr>
</tbody>
</table>

UseCamera

(Optional) String value that defines whether to use images from a camera connected to the Greengrass core device. Supported values are true and false.

When you set this value to true, the sample inference code accesses the camera on your device and runs inference locally on the captured image. The values of the ImageName and ImageDirectory parameters are ignored. Make sure that the user running this component has read/write access to the location where the camera stores captured images.

Default: false

Note

When you view the recipe of this component, the UseCamera configuration parameter doesn't appear in the default configuration. However, you can modify the value of this parameter in a configuration merge update (p. 390) when you deploy the component. When you set UseCamera to true, you must also create a symlink to enable the inference component to access your camera from the virtual environment that is created by the runtime component. For more information about using a camera with the sample inference components, see Update component configurations (p. 568).

2.0.x

MLRootPath

(Optional) The path of the folder on the device where inference components read images and write inference results. You can modify this value to any location on your device to which the user running this component has read/write access.

Default: /greengrass/v2/work/variant.DLR/greengrass_ml

Default: /greengrass/v2/work/variant.TensorFlowLite/greengrass_ml

Accelerator

Do not modify. Currently, the only supported value for the accelerator is cpu, because the models in the dependent model components are compiled only for the CPU accelerator.

ImageName

(Optional) The name of the image that the inference component uses as an input to a make prediction. The component looks for the image in the folder specified in ImageDirectory. The default location is MLRootPath/images. AWS IoT Greengrass supports the following image formats: jpeg, jpg, png, and npy.
Default: objects.jpg

**InferenceInterval**

(Optional) The time in seconds between each prediction made by the inference code. The sample inference code runs indefinitely and repeats its predictions at the specified time interval. For example, you can change this to a shorter interval if you want to use images taken by a camera for real-time prediction.

Default: 3600

**ModelResourceKey**

(Optional) The models that are used in the dependent public model component. Modify this parameter only if you override the public model component with a custom component.

Default:

```json
{
  armv7l: "DLR-yolo3-armv7l-cpu-ObjectDetection",
  x86_64: "DLR-yolo3-x86_64-cpu-ObjectDetection"
}
```

**Local log file**

This component uses the following log file.

```
/greengrass/v2/logs/aws.greengrass.DLRObjectDetection.log
```

**To view this component's logs**

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.DLRObjectDetection.log
```

**Changelog**

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.5</td>
<td>Component released in all AWS Regions.</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td></td>
<td>This version isn't available in Europe (London) (eu-west-2).</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Bug fixes and improvements</td>
</tr>
<tr>
<td></td>
<td>• Fixes an image scaling issue that resulted in inaccurate bounding boxes in the sample DLR object detection inference results.</td>
</tr>
</tbody>
</table>
## DLR image classification model store

The DLR image classification model store is a machine learning model component that contains pre-trained ResNet-50 models as Greengrass artifacts. The pre-trained models used in this component are fetched from the GluonCV Model Zoo and are compiled using SageMaker Neo Deep Learning Runtime.

The DLR image classification (p. 226) inference component uses this component as a dependency for the model source. To use a custom-trained DLR model, create a custom version (p. 583) of this model component, and include your custom model as a component artifact. You can use the recipe of this component as a template to create custom model components.

**Note**

The name of the DLR image classification model store component varies depending on its version. The component name for version 2.1.x and later versions is `variant.DLR.ImageClassification.ModelStore`. The component name for version 2.0.x is `variant.ImageClassification.ModelStore`.

### Topics
- Versions (p. 239)
- Type (p. 239)
- Requirements (p. 239)
- Dependencies (p. 239)
- Configuration (p. 240)
- Local log file (p. 240)
- Changelog (p. 240)
Versions

This component has the following versions:

- 2.1.x (variant.DLR.ImageClassification.ModelStore)
- 2.0.x (variant.ImageClassification.ModelStore)

Type

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device.
  Run the following command to install the dependencies:

  ```bash
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 240) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.1.4

The following table lists the dependencies for version 2.1.4 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>
2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>~2.0.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

**Configuration**

This component doesn't have any configuration parameters.

**Local log file**

This component doesn't output logs.

**Changelog**

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.4</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
</tbody>
</table>
| 2.1.1   | New features  
  - Add a sample ResNet-50 image classification model for Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano. |
| 2.0.4   | Initial version. |
DLR object detection model store

The DLR object detection model store is a machine learning model component that contains pre-trained YOLOv3 models as Greengrass artifacts. The sample models used in this component are fetched from the GluonCV Model Zoo and compiled using SageMaker Neo Deep Learning Runtime.

The DLR object detection (p. 232) inference component uses this component as a dependency for the model source. To use a custom-trained DLR model, create a custom version (p. 583) of this model component, and include your custom model as a component artifact. You can use the recipe of this component as a template to create custom model components.

**Note**
The name of the DLR object detection model store component varies depending on its version. The component name for version 2.1.x and later versions is variant.DLR.ObjectDetection.ModelStore. The component name for version 2.0.x is variant.ObjectDetection.ModelStore.

**Topics**
- Versions (p. 241)
- Type (p. 241)
- Requirements (p. 241)
- Dependencies (p. 242)
- Configuration (p. 243)
- Local log file (p. 243)
- Changelog (p. 243)

**Versions**

This component has the following versions:

- 2.1.x
- 2.0.x

**Type**

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

**Requirements**

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies:

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```
Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 243) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.1.4

The following table lists the dependencies for version 2.1.4 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>~2.0.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>
### Configuration

This component doesn't have any configuration parameters.

### Local log file

This component doesn't output logs.

### Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.4</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
</tbody>
</table>
| 2.1.1   | New features  
• Add a sample YOLOv3 object detection model for Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano. |
| 2.0.4   | Initial version. |

### DLR installer

The DLR installer component (`variant.DLR`) contains a script that installs Deep Learning Runtime (DLR) and its dependencies in a virtual environment on your device. The DLR image classification ([p. 226](#)) and DLR object detection ([p. 232](#)) components use this component as a dependency for installing DLR. Component version 1.6.x installs DLR v1.6.0 and component version 1.3.x installs DLR v1.3.0.

To use a different runtime, you can use the recipe of this component as a template to create a custom runtime component ([p. 581](#)).

### Topics

- Versions ([p. 243](#))
- Type ([p. 244](#))
- Requirements ([p. 244](#))
- Dependencies ([p. 244](#))
- Configuration ([p. 245](#))
- Usage ([p. 246](#))
- Local log file ([p. 246](#))
- Changelog ([p. 247](#))

### Versions

This component has the following versions:

- 1.6.x
• 1.3.x

Type

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

• On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
• On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies:

    sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev

Endpoints and ports

By default, this component uses an installer script to install packages using the `apt`, `yum`, `brew`, and `pip` commands, depending on what platform the core device uses. This component must be able to perform outbound requests to various package indexes and repositories to run the installer script. To allow this component's outbound traffic through a proxy or firewall, you must identify the endpoints for the package indexes and repositories where your core device connects to install.

Consider the following when you identify endpoints required for this component's install script:

• The endpoints depend on the core device's platform. For example, a core device that runs Ubuntu uses `apt` rather than `yum` or `brew`. Additionally, devices that use the same package index might have different source lists, so they might retrieve packages from different repositories.
• The endpoints might differ between multiple devices that use the same package index, because each device has its own source lists that define where to retrieve packages.
• The endpoints might change over time. Each package index provides the URLs of the repositories where you download packages, and the owner of a package can change what URLs the package index provides.

For more information about the dependencies that this component installs, and how to disable the installer script, see the UseInstaller (p. 246) configuration parameter.

For more information about endpoints and ports required for basic operation, see Allow device traffic through a proxy or firewall (p. 640).

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 247) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.
1.6.5 and 1.6.4

The following table lists the dependencies for version 1.6.5 and 1.6.4 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

1.6.3

The following table lists the dependencies for version 1.6.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

1.6.2

The following table lists the dependencies for version 1.6.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

1.6.1

The following table lists the dependencies for version 1.6.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

1.3.x

The following table lists the dependencies for version 1.3.x of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>~2.0.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

**MLRootPath**

(Optional) The path of the folder on the device where inference components read images and write inference results. You can modify this value to any location on your device to which the user running this component has read/write access.
Use Installer

(Optional) String value that defines whether to use the installer script in this component to install DLR and its dependencies. Supported values are true and false.

Set this value to false if you want to use a custom script for DLR installation, or if you want to include runtime dependencies in a pre-built Linux image. To use this component with the AWS-provided DLR inference components, you will need to install the following libraries, including any dependencies, and make them available the the default Greengrass system user.

- Python 3.7 or later, including pip for your version of Python.
- Deep Learning Runtime v1.6.0
- NumPy.
- OpenCV-Python.
- AWS IoT Device SDK v2 for Python.
- AWS Common Runtime (CRT) Python.
- Picamera, for Raspberry Pi devices only.
- awscam module, for AWS DeepLens devices only.

Default: true

Usage

Use this component with the UseInstaller configuration parameter set to true to install DLR and its dependencies on your device. The component sets up a virtual environment on your device that includes the OpenCV and NumPy libraries that are required for DLR.

Note
The installer script in this component also installs the latest versions of additional system libraries that are required to configure the virtual environment on your device and to use the installed machine learning framework. This might upgrade the existing system libraries on your device. Review the following table for the list of libraries that this component installs for each supported operating system. If you want to customize this installation process, set the UseInstaller configuration parameter to false, and develop your own installer script.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Libraries installed on the device system</th>
<th>Libraries installed in the virtual environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Armv7l</td>
<td>build-essential, cmake, cacertificates, git</td>
<td>setuptools, wheel</td>
</tr>
<tr>
<td>Amazon Linux 2</td>
<td>mesa-libGL</td>
<td>None</td>
</tr>
<tr>
<td>Ubuntu</td>
<td>wget</td>
<td>None</td>
</tr>
</tbody>
</table>

When you deploy your inference component, this runtime component first verifies if your device already has DLR and its dependencies installed, and if not, then it installs them for you.

Local log file

This component uses the following log file.

/greengrass/v2/logs/variant.DLR.log
To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

  ```bash
  sudo tail -f /greengrass/v2/logs/variant.DLR.log
  ```

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6.5</td>
<td>New feature</td>
</tr>
<tr>
<td></td>
<td>• Add the new <code>UseInstaller</code> configuration parameter that lets you disable the installation script in this component.</td>
</tr>
<tr>
<td>1.6.4</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>1.6.3</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>1.6.2</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>1.6.1</td>
<td>New features</td>
</tr>
<tr>
<td></td>
<td>• Install Deep Learning Runtime v1.6.0 and its dependencies.</td>
</tr>
<tr>
<td></td>
<td>• Add support for installing DLR on Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano.</td>
</tr>
<tr>
<td></td>
<td>Bug fixes and improvements</td>
</tr>
<tr>
<td></td>
<td>• Install the AWS IoT Device SDK in the virtual environment to read the component configuration and apply configuration changes.</td>
</tr>
<tr>
<td></td>
<td>• Additional minor bug fixes and improvements.</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Initial version. Installs DLR v1.3.0.</td>
</tr>
</tbody>
</table>

TensorFlow Lite image classification

The TensorFlow Lite image classification component (`aws.greengrass.TensorFlowLiteImageClassification`) contains sample inference code to perform image classification inference using the TensorFlow Lite runtime and a sample pre-trained MobileNet 1.0 quantized model. This component uses the variant TensorFlow Lite image classification model store (p. 256) and the TensorFlow Lite installer (p. 260) components as dependencies to download the TensorFlow Lite runtime and the sample model.

To use this inference component with a custom-trained TensorFlow Lite model, create a custom version (p. 583) of the dependent model store component. To use your own custom inference code, you can use the recipe of this component as a template to create a custom inference component (p. 587).

Topics

- Versions (p. 248)
- Type (p. 248)
- Requirements (p. 248)
• Dependencies (p. 248)
• Configuration (p. 249)
• Local log file (p. 251)
• Changelog (p. 251)

Versions

This component has the following versions:

• 2.1.x

Type

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component’s lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

• On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
• On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device.

Run the following command to install the dependencies:

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

 Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 251) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
<tr>
<td>TensorFlow Lite image classification model store (p. 256)</td>
<td>&gt;=2.1.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
<tr>
<td>TensorFlow Lite (p. 260)</td>
<td>&gt;=2.5.0 &lt;2.6.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>
2.1.2
The following table lists the dependencies for version 2.1.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
<tr>
<td>TensorFlow Lite image classification model store (p. 256)</td>
<td>&gt;=2.1.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
<tr>
<td>TensorFlow Lite (p. 260)</td>
<td>&gt;=2.5.0 &lt;2.6.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.1.1
The following table lists the dependencies for version 2.1.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
<tr>
<td>TensorFlow Lite image classification model store (p. 256)</td>
<td>&gt;=2.1.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
<tr>
<td>TensorFlow Lite (p. 260)</td>
<td>&gt;=2.5.0 &lt;2.6.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.1.0
The following table lists the dependencies for version 2.1.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
<tr>
<td>TensorFlow Lite image classification model store (p. 256)</td>
<td>&gt;=2.1.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
<tr>
<td>TensorFlow Lite (p. 260)</td>
<td>&gt;=2.5.0 &lt;2.6.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

**Configuration**

This component provides the following configuration parameters that you can customize when you deploy the component.

**accessControl**

(Optional) The object that contains the authorization policy (p. 403) that allows the component to publish messages to the default notifications topic.

Default:

```json
{
  "aws.greengrass.ipc.mqttproxy": {
```
"aws.greengrass.TensorFlowLiteImageClassification:mqttproxy:1": {
    "policyDescription": "Allows access to publish via topic ml/tflite/image-classification.",
    "operations": [
        "aws.greengrass#PublishToIoTCore"
    ],
    "resources": [
        "ml/tflite/image-classification"
    ]
}

PublishResultsOnTopic

(Optional) The topic on which you want to publish the inference results. If you modify this value, then you must also modify the value of resources in the accessControl parameter to match your custom topic name.

Default: ml/tflite/image-classification

Accelerator

The accelerator that you want to use. Supported values are cpu and gpu.

The sample models in the dependent model component support only CPU acceleration. To use GPU acceleration with a different custom model, create a custom model component (p. 583) to override the public model component.

Default: cpu

ImageDirectory

(Optional) The path of the folder on the device where inference components read images. You can modify this value to any location on your device to which you have read/write access.

Default: /greengrass/v2/packages/artifacts-unarchived/component-name/image_classification/sample_images/

Note

If you set the value of UseCamera to true, then this configuration parameter is ignored.

ImageName

(Optional) The name of the image that the inference component uses as an input to a make prediction. The component looks for the image in the folder specified in ImageDirectory. By default, the component uses the sample image in the default image directory. AWS IoT Greengrass supports the following image formats: jpeg, jpg, png, and npy.

Default: cat.jpeg

Note

If you set the value of UseCamera to true, then this configuration parameter is ignored.

InferenceInterval

(Optional) The time in seconds between each prediction made by the inference code. The sample inference code runs indefinitely and repeats its predictions at the specified time interval. For example, you can change this to a shorter interval if you want to use images taken by a camera for real-time prediction.

Default: 3600

ModelResourceKey

(Optional) The models that are used in the dependent public model component. Modify this parameter only if you override the public model component with a custom component.
Default:

```json
{
    model: "TensorFlowLite-Mobilenet"
}
```

**UseCamera**

(Optional) String value that defines whether to use images from a camera connected to the Greengrass core device. Supported values are `true` and `false`.

When you set this value to `true`, the sample inference code accesses the camera on your device and runs inference locally on the captured image. The values of the `ImageName` and `ImageDirectory` parameters are ignored. Make sure that the user running this component has read/write access to the location where the camera stores captured images.

Default: `false`

**Note**
When you view the recipe of this component, the `UseCamera` configuration parameter doesn't appear in the default configuration. However, you can modify the value of this parameter in a [configuration merge update](p. 390) when you deploy the component. When you set `UseCamera` to `true`, you must also create a symlink to enable the inference component to access your camera from the virtual environment that is created by the runtime component. For more information about using a camera with the sample inference components, see Update component configurations (p. 568).

**Local log file**

This component uses the following log file.

```
/greengrass/v2/logs/aws.greengrass.TensorFlowLiteImageClassification.log
```

**To view this component's logs**

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.TensorFlowLiteImageClassification.log
```

**Changelog**

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.3</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>2.1.0</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>
TensorFlow Lite object detection

The TensorFlow Lite object detection component (aws.greengrass.TensorFlowLiteObjectDetection) contains sample inference code to perform object detection inference using TensorFlow Lite and a sample pre-trained Single Shot Detection (SSD) MobileNet 1.0 model. This component uses the variant TensorFlow Lite object detection model store (p. 258) and the TensorFlow Lite installer (p. 260) components as dependencies to download TensorFlow Lite and the sample model.

To use this inference component with a custom-trained TensorFlow Lite model, you can create a custom version (p. 583) of the dependent model store component. To use your own custom inference code, use the recipe of this component as a template to create a custom inference component (p. 587).

Topics
• Versions (p. 252)
• Type (p. 252)
• Requirements (p. 252)
• Dependencies (p. 252)
• Configuration (p. 254)
• Local log file (p. 256)
• Changelog (p. 256)

Versions

This component has the following versions:

• 2.1.x

Type

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

• On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
• On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies:

  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its
dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 256) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
<tr>
<td>TensorFlow Lite image classification model store (p. 256)</td>
<td>&gt;=2.1.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
<tr>
<td>TensorFlow Lite (p. 260)</td>
<td>&gt;=2.5.0 &lt;2.6.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
<tr>
<td>TensorFlow Lite image classification model store (p. 256)</td>
<td>&gt;=2.1.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
<tr>
<td>TensorFlow Lite (p. 260)</td>
<td>&gt;=2.5.0 &lt;2.6.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
<tr>
<td>TensorFlow Lite image classification model store (p. 256)</td>
<td>&gt;=2.1.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
<tr>
<td>TensorFlow Lite (p. 260)</td>
<td>&gt;=2.5.0 &lt;2.6.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Dependency</td>
<td>Compatible versions</td>
<td>Dependency type</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>---------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>TensorFlow Lite image classification model store (p. 256)</td>
<td>&gt;=2.1.0 &lt;2.0</td>
<td>Hard</td>
</tr>
<tr>
<td>TensorFlow Lite (p. 260)</td>
<td>&gt;=2.5.0 &lt;2.6.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

**Configuration**

This component provides the following configuration parameters that you can customize when you deploy the component.

**accessControl**

(Optional) The object that contains the authorization policy (p. 403) that allows the component to publish messages to the default notifications topic.

Default:

```json
{
"aws.greengrass.ipc.mqttproxy": {
  "aws.greengrass.TensorFlowLiteObjectDetection:mqttproxy:1": {
    "policyDescription": "Allows access to publish via topic ml/tflite/object-detection.",
    "operations": [
      "aws.greengrass#PublishToIoTCore"
    ],
    "resources": [
      "ml/tflite/object-detection"
    ]
  }
}
```

**PublishResultsOnTopic**

(Optional) The topic on which you want to publish the inference results. If you modify this value, then you must also modify the value of resources in the accessControl parameter to match your custom topic name.

Default: ml/tflite/object-detection

**Accelerator**

The accelerator that you want to use. Supported values are cpu and gpu.

The sample models in the dependent model component support only CPU acceleration. To use GPU acceleration with a different custom model, create a custom model component (p. 583) to override the public model component.

Default: cpu

**ImageDirectory**

(Optional) The path of the folder on the device where inference components read images. You can modify this value to any location on your device to which you have read/write access.

Default: /greengrass/v2/packages/artifacts-unarchived/component-name/object_detection/sample_images/
If you set the value of \texttt{UseCamera} to \texttt{true}, then this configuration parameter is ignored.

\textbf{ImageName}

(Optional) The name of the image that the inference component uses as an input to make a prediction. The component looks for the image in the folder specified in \texttt{ImageDirectory}. By default, the component uses the sample image in the default image directory. AWS IoT Greengrass supports the following image formats: \texttt{jpeg}, \texttt{jpg}, \texttt{png}, and \texttt{npy}.

Default: \texttt{objects.jpg}

\textbf{InferenceInterval}

(Optional) The time in seconds between each prediction made by the inference code. The sample inference code runs indefinitely and repeats its predictions at the specified time interval. For example, you can change this to a shorter interval if you want to use images taken by a camera for real-time prediction.

Default: 3600

\textbf{ModelResourceKey}

(Optional) The models that are used in the dependent public model component. Modify this parameter only if you override the public model component with a custom component.

Default:

\begin{verbatim}
{
  model: "TensorFlowLite-SSD"
}
\end{verbatim}

\textbf{UseCamera}

(Optional) String value that defines whether to use images from a camera connected to the Greengrass core device. Supported values are \texttt{true} and \texttt{false}.

When you set this value to \texttt{true}, the sample inference code accesses the camera on your device and runs inference locally on the captured image. The values of the \texttt{ImageName} and \texttt{ImageDirectory} parameters are ignored. Make sure that the user running this component has read/write access to the location where the camera stores captured images.

Default: \texttt{false}

\textbf{Note}

When you view the recipe of this component, the \texttt{UseCamera} configuration parameter doesn't appear in the default configuration. However, you can modify the value of this parameter in a \texttt{configuration merge update (p. 390)} when you deploy the component. When you set \texttt{UseCamera} to \texttt{true}, you must also create a symlink to enable the inference component to access your camera from the virtual environment that is created by the runtime component. For more information about using a camera with the sample inference components, see Update component configurations (p. 568).

\textbf{Note}

When you view the recipe of this component, the \texttt{UseCamera} configuration parameter doesn't appear in the default configuration. However, you can modify the value of this parameter in a \texttt{configuration merge update (p. 390)} when you deploy the component. When you set \texttt{UseCamera} to \texttt{true}, you must also create a symlink to enable the inference component to access your camera from the virtual environment that is created by the runtime component.
component. For more information about using a camera with the sample inference components, see Update component configurations (p. 568).

Local log file

This component uses the following log file.

```
/greengrass/v2/logs/aws.greengrass.TensorFlowLiteObjectDetection.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.TensorFlowLiteObjectDetection.log
```

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.4</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
</tbody>
</table>
| 2.1.1   | Bug fixes and improvements  
- Fixes an image scaling issue that resulted in inaccurate bounding boxes in the sample TensorFlow Lite object detection inference results. |
| 2.1.0   | Initial version. |

TensorFlow Lite image classification model store

The TensorFlow Lite image classification model store (variant.TensorFlowLite.ImageClassification.ModelStore) is a machine learning model component that contains a pre-trained MobileNet v1 model as a Greengrass artifact. The sample model used in this component is fetched from the TensorFlow Hub and implemented using TensorFlow Lite.

The TensorFlow Lite image classification (p. 247) inference component uses this component as a dependency for the model source. To use a custom-trained TensorFlow Lite model, create a custom version (p. 583) of this model component, and include your custom model as a component artifact. You can use the recipe of this component as a template to create custom model components.

Topics

- Versions (p. 257)
- Type (p. 257)
- Requirements (p. 257)
- Dependencies (p. 257)
- Configuration (p. 258)
• Local log file (p. 258)
• Changelog (p. 258)

Versions

This component has the following versions:

• 2.1.x

Type

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

• On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
• On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies:

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 258) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>
2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

Configuration

This component doesn't have any configuration parameters.

Local log file

This component doesn't output logs.

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.3</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>2.1.0</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

TensorFlow Lite object detection model store

The TensorFlow Lite object detection model store (variant.TensorFlowLite.ObjectDetection.ModelStore) is a machine learning model component that contains a pre-trained Single Shot Detection (SSD) MobileNet model as a Greengrass artifact. The sample model used in this component is fetched from the TensorFlow Hub and implemented using TensorFlow Lite.

The TensorFlow Lite object detection (p. 252) inference component uses this component as a dependency for the model source. To use a custom-trained TensorFlow Lite model, create a custom version (p. 583) of this model component, and include your custom model as a component artifact. You can use the recipe of this component as a template to create custom model components.

Topics

- Versions (p. 259)
• Type (p. 259)
• Requirements (p. 259)
• Dependencies (p. 259)
• Configuration (p. 260)
• Local log file (p. 260)
• Changelog (p. 260)

Versions

This component has the following versions:

• 2.1.x

Type

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component’s lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

• On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
• On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device.
  
  Run the following command to install the dependencies:

  ```sh
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev libavformat-dev
  libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 260) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.
TensorFlow Lite installer

The TensorFlow Lite installer component (variant.TensorFlowLite) contains a script that installs TensorFlow Lite version 2.5.0 and its dependencies in a virtual environment on your device. The TensorFlow Lite image classification (p. 247) and TensorFlow Lite object detection (p. 252) component use this runtime component as a dependency for installing TensorFlow Lite.

To use a different runtime, you can use the recipe of this component as a template to create a custom runtime component (p. 581).
Topics
- Versions (p. 261)
- Type (p. 261)
- Requirements (p. 261)
- Dependencies (p. 262)
- Configuration (p. 263)
- Usage (p. 263)
- Local log file (p. 264)
- Changelog (p. 264)

Versions
This component has the following versions:
- 2.5.x

Type
This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Requirements
This component has the following requirements:
- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device.
  Run the following command to install the dependencies:

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

Endpoints and ports
By default, this component uses an installer script to install packages using the apt, yum, brew, and pip commands, depending on what platform the core device uses. This component must be able to perform outbound requests to various package indexes and repositories to run the installer script. To allow this component's outbound traffic through a proxy or firewall, you must identify the endpoints for the package indexes and repositories where your core device connects to install.

Consider the following when you identify endpoints required for this component's install script:
- The endpoints depend on the core device's platform. For example, a core device that runs Ubuntu uses apt rather than yum or brew. Additionally, devices that use the same package index might have different source lists, so they might retrieve packages from different repositories.
- The endpoints might differ between multiple devices that use the same package index, because each device has its own source lists that define where to retrieve packages.
The endpoints might change over time. Each package index provides the URLs of the repositories where you download packages, and the owner of a package can change what URLs the package index provides.

For more information about the dependencies that this component installs, and how to disable the installer script, see the UseInstaller (p. 263) configuration parameter.

For more information about endpoints and ports required for basic operation, see Allow device traffic through a proxy or firewall (p. 640).

## Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 264) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

### 2.5.4 and 2.5.3

The following table lists the dependencies for version 2.5.4 and 2.5.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

### 2.5.2

The following table lists the dependencies for version 2.5.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

### 2.5.1

The following table lists the dependencies for version 2.5.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

### 2.5.0

The following table lists the dependencies for version 2.5.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>
For more information about component dependencies, see the component recipe reference (p. 343).

**Configuration**

This component provides the following configuration parameters that you can customize when you deploy the component.

**MLRootPath**

(Optional) The path of the folder on the device where inference components read images and write inference results. You can modify this value to any location on your device to which the user running this component has read/write access.

Default: `/greengrass/v2/work/variant.TensorFlowLite/greengrass_ml`

**UseInstaller**

(Optional) String value that defines whether to use the installer script in this component to install TensorFlow Lite and its dependencies. Supported values are `true` and `false`.

Set this value to `false` if you want to use a custom script for TensorFlow Lite installation, or if you want to include runtime dependencies in a pre-built Linux image. To use this component with the AWS-provided TensorFlow Lite inference components, you will need to install the following libraries, including any dependencies, and make them available to the default Greengrass system user.

- Python 3.8 or later, including `pip` for your version of Python
- TensorFlow Lite v2.5.0
- NumPy
- OpenCV-Python
- AWS IoT Device SDK v2 for Python
- AWS Common Runtime (CRT) Python
- Picamera (for Raspberry Pi devices)
- awscam module (for AWS DeepLens devices)

Default: `true`

**Usage**

Use this component with the `UseInstaller` configuration parameter set to `true` to install TensorFlow Lite and its dependencies on your device. The component sets up a virtual environment on your device that includes the OpenCV and NumPy libraries that are required for TensorFlow Lite.

**Note**

The installer script in this component also installs the latest versions of additional system libraries that are required to configure the virtual environment on your device and to use the installed machine learning framework. This might upgrade the existing system libraries on your device. Review the following table for the list of libraries that this component installs for each supported operating system. If you want to customize this installation process, set the `UseInstaller` configuration parameter to `false`, and develop your own installer script.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Libraries installed on the device system</th>
<th>Libraries installed in the virtual environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Armv7l</td>
<td>build-essential, cmake, cacertificates, git</td>
<td>setuptools, wheel</td>
</tr>
</tbody>
</table>
When you deploy your inference component, this runtime component first verifies if your device already has TensorFlow Lite and its dependencies installed. If not, then the runtime component installs them for you.

### Local log file

This component uses the following log file.

```
/greengrass/v2/logs/variant.TensorFlowLite.log
```

### To view this component's logs

- Run the following command on the core device to view this component’s log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/variant.TensorFlowLite.log
```

### Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.4</td>
<td>New feature</td>
</tr>
<tr>
<td></td>
<td>• Add the new UseInstaller configuration parameter that lets you disable the installation script in this component.</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>2.5.0</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

---

**Modbus-RTU protocol adapter**

The Modbus-RTU protocol adapter component (`aws.greengrass.Modbus`) polls information from local Modbus RTU devices.

To request information from a local Modbus RTU device with this component, publish a message to the topic where this component subscribes. In the message, specify the Modbus RTU request to send to a device. Then, this component publishes a response that contains the result of the Modbus RTU request.
Note
This component provides similar functionality to the Modbus RTU protocol adapter connector in AWS IoT Greengrass V1. For more information, see Modbus RTU protocol adapter connector in the AWS IoT Greengrass V1 Developer Guide.

Topics
- Versions (p. 265)
- Type (p. 265)
- Requirements (p. 265)
- Dependencies (p. 266)
- Configuration (p. 268)
- Input data (p. 270)
- Output data (p. 271)
- Modbus RTU requests and responses (p. 273)
- Local log file (p. 279)
- Licenses (p. 279)
- Changelog (p. 279)

Versions
This component has the following versions:
- 2.0.x

Type
This component is a Lambda component (aws.greengrass.lambda). The Greengrass nucleus (p. 136) runs this component's Lambda function using the Lambda launcher component (p. 188).
For more information, see Component types (p. 323).

Requirements
This component has the following requirements:
- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Requirements to run Lambda functions (p. 43).
- Python version 3.7 installed on the core device and added to the PATH environment variable.
- A physical connection between the AWS IoT Greengrass core device and the Modbus devices. The core device must be physically connected to the Modbus RTU network through a serial port, such as a USB port.
- To receive output data from this component, you must merge the following configuration update for the legacy subscription router component (p. 194) when you deploy this component. The legacy subscription router component (aws.greengrass.LegacySubscriptionRouter) is a dependency of this component. This configuration specifies the topic where this component publishes responses.

```

`
Legacy subscription router v2.0.x

```json
{
  "subscriptions": {
    "aws-greengrass-modbus": {
      "id": "aws-greengrass-modbus",
      "subject": "modbus/adapter/response",
      "target": "cloud"
    }
  }
}
```

• Replace `region` with the AWS Region that you use.

• Replace `version` with the version of the Lambda function that this component runs. To find the Lambda function version, you must view the recipe for the version of this component that you want to deploy. Open this component’s details page in the AWS IoT Greengrass console, and look for the Lambda function key-value pair. This key-value pair contains the name and version of the Lambda function.

  **Important**
  You must update the Lambda function version on the legacy subscription router every time you deploy this component. This ensures that you use the correct Lambda function version for the component version that you deploy.

For more information, see Create deployments (p. 384).

## Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 279) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

**2.0.7**

The following table lists the dependencies for version 2.0.7 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>
2.0.6

The following table lists the dependencies for version 2.0.6 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.0.5

The following table lists the dependencies for version 2.0.5 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.0.4

The following table lists the dependencies for version 2.0.4 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.3 &lt;2.1.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>&gt;=1.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>
For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

Note
This component's default configuration includes Lambda function parameters. We recommend that you edit only the following parameters to configure this component on your devices.

lambdaParams
An object that contains the parameters for this component's Lambda function. This object contains the following information:
EnvironmentVariables
An object that contains the Lambda function's parameters. This object contains the following information:
ModbusLocalPort
The absolute path to the physical Modbus serial port on the device, such as /dev/ttyS2.

To run this component in a container, you must define this path as a system device (p. 269) that the component can access. This component runs in a container by default.

Note
This component has read/write access to the device.

containerMode
(Optional) The containerization mode for this component. Choose from the following options:
- GreengrassContainer – The component runs in an isolated runtime environment inside the AWS IoT Greengrass container.

If you specify this option, you specify a system device (p. 269) to give the container access to the Modbus device.
- NoContainer – The component doesn't run in an isolated runtime environment.

Default: GreengrassContainer

containerParams
(Optional) An object that contains the container parameters for this component. The component uses these parameters if you specify GreengrassContainer for containerMode.

This object contains the following information:
memorySize
(Optional) The amount of memory (in kilobytes) to allocate to the component.
Defaults to 512 MB (525,312 KB).

**devices**

(Optional) An object that specifies the system devices that the component can access in a container.

**Important**

To run this component in a container, you must specify the system device that you configure in the `ModbusLocalPort` environment variable.

This object contains the following information:

0 – This is an array index as a string.

An object that contains the following information:

**path**

The path to the system device on the core device. This must have the same value as the value that you configure for `ModbusLocalPort`.

**permission**

(Optional) The permission to access the system device from the container. This value must be `rw`, which specifies that the component has read/write access to the system device.

Default: `rw`

**addGroupOwner**

(Optional) Whether or not to add the system group that runs the component as an owner of the system device.

Default: `true`

**pubsubTopics**

(Optional) An object that contains the topics where the component subscribes to receive messages. You can specify each topic and whether the component subscribes to MQTT topics from AWS IoT Core or local publish/subscribe topics.

This object contains the following information:

0 – This is an array index as a string.

An object that contains the following information:

**type**

(Optional) The type of publish/subscribe messaging that this component uses to subscribe to messages. Choose from the following options:

- **Pubsub** – Subscribe to local publish/subscribe messages. If you choose this option, the topic can’t contain MQTT wildcards. For more information about how to send messages from custom component when you specify this option, see Publish/subscribe local messages (p. 410).
- **IotCore** – Subscribe to AWS IoT Core MQTT messages. If you choose this option, the topic can contain MQTT wildcards. For more information about how to send messages from custom components when you specify this option, see Publish/subscribe AWS IoT Core MQTT messages (p. 433).

Default: Pubsub
topic

(Optional) The topic to which the component subscribes to receive messages. If you specify IotCore for type, you can use MQTT wildcards (+ and #) in this topic.

Example Example: Configuration merge update (container mode)

```json
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "ModbusLocalPort": "/dev/ttyS2"
    }
  },
  "containerMode": "GreengrassContainer",
  "containerParams": {
    "devices": {
      "0": {
        "path": "/dev/ttyS2",
        "permission": "rw",
        "addGroupOwner": true
      }
    }
  }
}
```

Example Example: Configuration merge update (no container mode)

```json
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "ModbusLocalPort": "/dev/ttyS2"
    }
  },
  "containerMode": "NoContainer"
}
```

## Input data

This component accepts Modbus RTU request parameters on the following topic and sends the Modbus RTU request to the device. By default, this component subscribes to local publish/subscribe messages. For more information about how to publish messages to this component from your custom components, see Publish/subscribe local messages (p. 410).

**Default topic (local publish/subscribe):** modbus/adapter/request

The message accepts the following properties. Input messages must be in JSON format.

**request**

The parameters for the Modbus RTU request to send.

The shape of the request message depends on the type of Modbus RTU request that it represents. The following properties are required for all requests.

**Type:** object that contains the following information:

**operation**

The name of the operation to run. For example, specify ReadCoilsRequest to read coils on a Modbus RTU device. For more information about supported operations, see Modbus RTU requests and responses (p. 273).
Output data

Type: string
device

The target device of the request.

This value must be an integer between 0 and 247.

Type: integer

The other parameters to include in the request depend on the operation. This component handles the cyclic redundancy check (CRC) to verify data requests for you.

Note
If you request includes an address property, you must specify its value as an integer. For example, "address": 1.

id

An arbitrary ID for the request. Use this property to map an input request to an output response. When you specify this property, the component sets the id property in the response object to this value.

Type: string

Example Example input: Read coils request

```
{
  "request": {
    "operation": "ReadCoilsRequest",
    "device": 1,
    "address": 1,
    "count": 1
  },
  "id": "MyRequest"
}
```

Output data

This component publishes responses as output data on the following MQTT topic by default. You must specify this topic as the subject in the configuration for the legacy subscription router component (p. 194). For more information about how to subscribe to messages on this topic in your custom components, see Publish/subscribe AWS IoT Core MQTT messages (p. 433).

Default topic (AWS IoT Core MQTT): modbus/adapter/response

The shape of the response message depends on the request operation and the response status. For examples, see Example requests and responses (p. 273).

Every response includes the following properties:

response

The response from the Modbus RTU device.

Type: object that contains the following information:

status

The status of the request. The status can be one of the following values:
Output data

- **Success** – The request was valid, the component sent the request to the Modbus RTU network, and the Modbus RTU network returned a response.

- **Exception** – The request was valid, the component sent the request to the Modbus RTU network, and the Modbus RTU network returned an exception. For more information, see [Response status: Exception](#).

- **No Response** – The request was invalid, and the component caught the error before it sent the request to the Modbus RTU network. For more information, see [Response status: No response](#).

**operation**

The operation that the component requested.

**device**

The device where the component sent the request.

**payload**

The response from the Modbus RTU device. If the status is **No Response**, this object contains only an **error** property with the description of the error (for example, [Input/Output] No Response received from the remote unit).

**id**

The ID of the request, which you can use to identify which response corresponds to which request.

**Note**

A response for a write operation is simply an echo of the request. Although write responses don't include meaningful information, it's a good practice to check the status of the response to see if the request succeeds or fails.

**Example Example output: Success**

```json
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "function_code": 1,
      "bits": [1]
    }
  },
  "id": "MyRequest"
}
```

**Example Example output: Failure**

```json
{
  "response": {
    "status": "fail",
    "error_message": "Internal Error",
    "error": "Exception",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "function_code": 129,
      "exception_code": 2
    }
  }
}
```
Modbus RTU requests and responses

This connector accepts Modbus RTU request parameters as input data and publishes responses as output data.

The following common operations are supported.

<table>
<thead>
<tr>
<th>Operation name in request</th>
<th>Function code in response</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadCoilsRequest</td>
<td>01</td>
</tr>
<tr>
<td>ReadDiscreteInputsRequest</td>
<td>02</td>
</tr>
<tr>
<td>ReadHoldingRegistersRequest</td>
<td>03</td>
</tr>
<tr>
<td>ReadInputRegistersRequest</td>
<td>04</td>
</tr>
<tr>
<td>WriteSingleCoilRequest</td>
<td>05</td>
</tr>
<tr>
<td>WriteSingleRegisterRequest</td>
<td>06</td>
</tr>
<tr>
<td>WriteMultipleCoilsRequest</td>
<td>15</td>
</tr>
<tr>
<td>WriteMultipleRegistersRequest</td>
<td>16</td>
</tr>
<tr>
<td>MaskWriteRegisterRequest</td>
<td>22</td>
</tr>
<tr>
<td>ReadWriteMultipleRegistersRequest</td>
<td>23</td>
</tr>
</tbody>
</table>

Example requests and responses

The following are example requests and responses for supported operations.

Read coils

Request example:

```json
{
    "request": {
        "operation": "ReadCoilsRequest",
        "device": 1,
        "address": 1,
        "count": 1
    },
    "id": "TestRequest"
}
```

Response example:

```json
{
    "response": {
        "status": "success",
        "device": 1,
```
Modbus RTU requests and responses

"operation": "ReadCoilsRequest",
"payload": {
    "function_code": 1,
    "bits": [1]
},
"id": "TestRequest"

Read discrete inputs

Request example:

```
{
    "request": {
        "operation": "ReadDiscreteInputsRequest",
        "device": 1,
        "address": 1,
        "count": 1
    },
    "id": "TestRequest"
}
```

Response example:

```
{
    "response": {
        "status": "success",
        "device": 1,
        "operation": "ReadDiscreteInputsRequest",
        "payload": {
            "function_code": 2,
            "bits": [1]
        }
    },
    "id": "TestRequest"
}
```

Read holding registers

Request example:

```
{
    "request": {
        "operation": "ReadHoldingRegistersRequest",
        "device": 1,
        "address": 1,
        "count": 1
    },
    "id": "TestRequest"
}
```

Response example:

```
{
    "response": {
        "status": "success",
        "device": 1,
        "operation": "ReadHoldingRegistersRequest",
        "payload": {
```
Modbus RTU requests and responses

Read input registers

Request example:

```json
{
    "request": {
        "operation": "ReadInputRegistersRequest",
        "device": 1,
        "address": 1,
        "count": 1
    },
    "id": "TestRequest"
}
```

Write single coil

Request example:

```json
{
    "request": {
        "operation": "WriteSingleCoilRequest",
        "device": 1,
        "address": 1,
        "value": 1
    },
    "id": "TestRequest"
}
```

Response example:

```json
{
    "response": {
        "status": "success",
        "device": 1,
        "operation": "WriteSingleCoilRequest",
        "payload": {
            "function_code": 5,
            "address": 1,
            "value": true
        }
    },
    "id": "TestRequest"
}
```

Write single register

Request example:

```json
{
    "request": {
        "operation": "WriteSingleRegisterRequest",
        "device": 1,
        "address": 1,
        "value": 275
    },
    "id": "TestRequest"
}
```
Write multiple coils

**Request example:**

```json
{
    "request": {
        "operation": "WriteMultipleCoilsRequest",
        "device": 1,
        "address": 1,
        "values": [1, 0, 0, 1]
    },
    "id": "TestRequest"
}
```

**Response example:**

```json
{
    "response": {
        "status": "success",
        "device": 1,
        "operation": "WriteMultipleCoilsRequest",
        "payload": {
            "function_code": 15,
            "address": 1,
            "count": 4
        }
    },
    "id": "TestRequest"
}
```

Write multiple registers

**Request example:**

```json
{
    "request": {
        "operation": "WriteMultipleRegistersRequest",
        "device": 1,
        "address": 1,
        "values": [20, 30, 10]
    },
    "id": "TestRequest"
}
```

**Response example:**

```json
{
    "response": {
        "status": "success",
        "device": 1,
        "operation": "WriteMultipleRegistersRequest",
        "payload": {
            "function_code": 23,
            "address": 1,
            "count": 3
        }
    },
    "id": "TestRequest"
}
```
Modbus RTU requests and responses

Mask write register

**Request example:**

```json
{
    "request": {
        "operation": "MaskWriteRegisterRequest",
        "device": 1,
        "address": 1,
        "and_mask": 175,
        "or_mask": 1
    },
    "id": "TestRequest"
}
```

**Response example:**

```json
{
    "response": {
        "status": "success",
        "device": 1,
        "operation": "MaskWriteRegisterRequest",
        "payload": {
            "function_code": 22,
            "and_mask": 0,
            "or_mask": 8
        }
    },
    "id": "TestRequest"
}
```

Read write multiple registers

**Request example:**

```json
{
    "request": {
        "operation": "ReadWriteMultipleRegistersRequest",
        "device": 1,
        "read_address": 1,
        "read_count": 2,
        "write_address": 3,
        "write_registers": [20,30,40]
    },
    "id": "TestRequest"
}
```

**Response example:**

```json
{
    "response": {
        "status": "success",
        "device": 1,
        "operation": "ReadWriteMultipleRegistersRequest",
        "payload": {
            "function_code": 23,
```
"registers": [10, 20, 10, 20]
},
"id": "TestRequest"
}

**Note**
The response includes the registers that the component reads.

**Response status: Exception**

Exceptions can occur when the request format is valid, but the request is not completed successfully. In this case, the response contains the following information:

- The *status* is set to *Exception*.
- The *function_code* equals the function code of the request + 128.
- The *exception_code* contains the exception code. For more information, see Modbus exception codes.

**Example:**

```json
{
  "response": {
    "status": "fail",
    "error_message": "Internal Error",
    "error": "Exception",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "function_code": 129,
      "exception_code": 2
    }
  },
  "id": "TestRequest"
}
```

**Response status: No response**

This connector performs validation checks on the Modbus request. For example, it checks for invalid formats and missing fields. If the validation fails, the connector doesn't send the request. Instead, it returns a response that contains the following information:

- The *status* is set to *No Response*.
- The *error* contains the error reason.
- The *error_message* contains the error message.

**Examples:**

```json
{
  "response": {
    "status": "fail",
    "error_message": "Invalid address field. Expected <type 'int'>, got <type 'str'>",
    "error": "No Response",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
```

278
If the request targets a nonexistent device or if the Modbus RTU network is not working, you might get a `ModbusIOException`, which uses the No Response format.

```
{
  "response": {
    "status": "fail",
    "error_message": "[Input/Output] No Response received from the remote unit",
    "error": "No Response",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "error": "[Input/Output] No Response received from the remote unit"
    }
  },
  "id": "TestRequest"
}
```

Local log file

This component uses the following log file.

```
/greengrass/v2/logs/aws.greengrass.Modbus.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.Modbus.log
```

Licenses

This component includes the following third-party software/licensing:

- `pymodbus`/BSD License
- `pyserial`/BSD License

This component is released under the [Greengrass Core Software License Agreement](https://aws.amazon.com/IoT-Greengrass/license/).

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.7</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
</tbody>
</table>
MQTT bridge

The MQTT bridge component (aws.greengrass.clientdevices.mqtt.Bridge) relays MQTT messages between client devices, local Greengrass publish/subscribe, and AWS IoT Core. You can use this component to act on MQTT messages from client devices in custom components and sync client devices with the AWS Cloud.

**Note**
Client devices are local IoT devices that connect to a Greengrass core device to send MQTT messages and data to process. For more information, see Interact with local IoT devices (p. 477).

You can use this component to relay messages between the following message brokers:

- Local MQTT – The local MQTT broker handles messages between client devices and a core device.
- Local publish/subscribe – The local Greengrass message broker handles messages between components on a core device. For more information about how to interact with these messages in Greengrass components, see Publish/subscribe local messages (p. 410).
- AWS IoT Core – The AWS IoT Core MQTT broker handles messages between IoT devices and AWS Cloud destinations. For more information about how to interact with these messages in Greengrass components, see Publish/subscribe AWS IoT Core MQTT messages (p. 433).

**Note**
The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings (p. 128).

**Topics**
- Versions (p. 280)
- Type (p. 281)
- Dependencies (p. 281)
- Configuration (p. 281)
- Local log file (p. 283)
- Changelog (p. 283)

**Versions**

This component has the following versions:

- 2.0.x
Type

This component is a plugin component (aws.greengrass.plugin). The Greengrass nucleus (p. 136) runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

For more information, see Component types (p. 323).

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 283) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client device auth (p. 146)</td>
<td>&gt;=2.0.0 &lt;2.1.0</td>
<td>Hard</td>
</tr>
<tr>
<td>MQTT broker (Moquette) (p. 283)</td>
<td>&gt;=2.0.0 &lt;2.1.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

mqttTopicMapping

The topic mappings to bridge. This component subscribes to messages on the source topic and publishes the messages it receives to the destination topic. Each topic mapping defines the topic, source type, and destination type.

This object contains the following information:

- topicMappingNameKey

  The name of this topic mapping. Replace topicMappingNameKey with a name that helps you identify this topic mapping.

  This object contains the following information:

  - topic

    The topic to bridge between the source and target brokers.
If you specify the `LocalMqtt` or `IotCore` source broker, you can use the + and # MQTT topic wildcards to relay messages on all topics that match a topic filter. For more information, see MQTT topics in the AWS IoT Core Developer Guide.

**source**

The source message broker. Choose from the following options:

- `LocalMqtt` – The local MQTT broker where client devices communicate.
- `Pubsub` – The local Greengrass publish/subscribe message broker.
- `IotCore` – The AWS IoT Core MQTT message broker.

**Note**
The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings (p. 128).

**source** and **target** must be different.

**target**

The target message broker. Choose from the following options:

- `LocalMqtt` – The local MQTT broker where client devices communicate.
- `Pubsub` – The local Greengrass publish/subscribe message broker.
- `IotCore` – The AWS IoT Core MQTT message broker.

**Note**
The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings (p. 128).

**source** and **target** must be different.

### Example: Configuration merge update

The following example configuration update specifies to sync the `clients/MyClientDevice1/hello/world` and `clients/MyClientDevice2/hello/world` topics between client devices and AWS IoT Core.

```json
{
  "mqttTopicMapping": {
    "ClientDevice1Mapping": {
      "topic": "clients/MyClientDevice1/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
    "ClientDevice2Mapping": {
      "topic": "clients/MyClientDevice2/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```
Local log file

This component uses the same log file as the Greengrass nucleus (p. 136) component.

```
/greengrass/v2/logs/greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.1</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.0.0</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

MQTT broker (Moquette)

The Moquette MQTT broker component (aws.greengrass.clientdevices.mqtt.Moquette) handles MQTT messages between client devices and a Greengrass core device. This component provides a modified version of the Moquette MQTT broker.

This broker implements the MQTT 3.1.1 protocol. It includes support for QoS 0, QoS 1, QoS 2 retained messages, last will messages, and persistent subscriptions.

Note

Client devices are local IoT devices that connect to a Greengrass core device to send MQTT messages and data to process. For more information, see Interact with local IoT devices (p. 477).

Topics

- Versions (p. 283)
- Type (p. 284)
- Requirements (p. 284)
- Dependencies (p. 284)
- Configuration (p. 284)
- Local log file (p. 285)
- Changelog (p. 285)

Versions

This component has the following versions:
This component is a plugin component (aws.greengrass.plugin). The Greengrass nucleus (p. 136) runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

For more information, see Component types (p. 323).

**Requirements**

This component has the following requirements:

- The core device must be able to accept connections on the port where the MQTT broker operates. This component runs the MQTT broker on port 8883 by default. You can specify a different port when you configure this component.
- If you use the IP detector component (p. 177) to manage MQTT broker endpoints, you must use the default port 8883 for this component.

**Dependencies**

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 285) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client device auth (p. 146)</td>
<td>&gt;=2.0.0 &lt;2.1.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

**Configuration**

This component provides the following configuration parameters that you can customize when you deploy the component.

moquette

(Optional) The Moquette MQTT broker configuration to use. You can configure a subset of Moquette configuration options in this component. For more information, see the inline comments in the Moquette configuration file.

This object contains the following information:
ssl_port

(Optional) The port where the MQTT broker operates.

**Note**
If you use the IP detector component (p. 177) to manage MQTT broker endpoints, you must use the default port 8883 for this component.

Default: 8883

host

(Optional) The interface where the MQTT broker binds. For example, you might change this parameter so that the MQTT broker binds only to a specific local network.

Default: 0.0.0.0 (binds to all network interfaces)

### Example Example: Configuration merge update

The following example configuration specifies to operate the MQTT broker on port 443.

```json
{
  "moquette": {
    "ssl_port": "443"
  }
}
```

### Local log file

This component uses the same log file as the Greengrass nucleus (p. 136) component.

/greengrass/v2/logs/greengrass.log

#### To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

  ```
sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

### Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.1</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.0.0</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

### Secret manager

The secret manager component (`aws.greengrass.SecretManager`) deploys secrets from AWS Secrets Manager to Greengrass core devices. Use this component to securely use credentials, such as passwords,
in custom components on your Greengrass core devices. For more information about Secrets Manager, see What is AWS Secrets Manager? in the AWS Secrets Manager User Guide.

This component encrypts secrets on the core device to keep your credentials and passwords secure until you need to use them.

**Topics**
- Versions (p. 286)
- Type (p. 286)
- Requirements (p. 286)
- Dependencies (p. 287)
- Configuration (p. 288)
- Local log file (p. 289)
- Changelog (p. 289)

**Versions**

This component has the following versions:
- 2.0.x

**Type**

This component is a plugin component (aws.greengrass.plugin). The Greengrass nucleus (p. 136) runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

For more information, see Component types (p. 323).

**Requirements**

This component has the following requirements:

- The Greengrass device role (p. 627) must allow the secretsmanager:GetSecretValue action, as shown in the following example IAM policy.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "secretsmanager:GetSecretValue"
      ],
      "Effect": "Allow",
      "Resource": [
      ]
    }
  ]
}
```
Note
If you use a customer-managed AWS Key Management Service key to encrypt secrets, the device role must also allow the kms:Decrypt action.

For more information about IAM policies for Secrets Manager, see the following in the AWS Secrets Manager User Guide:
- Authentication and access control for AWS Secrets Manager
- Actions, resources, and context keys you can use in an IAM policy or secret policy for AWS Secrets Manager
- Custom components must define an authorization policy that allows aws.greengrass#GetSecretValue to get secrets that you store with this component. In this authorization policy, you can restrict components' access to specific secrets. For more information, see secret manager IPC authorization (p. 455).

Endpoints and ports
This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall (p. 640).

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>secretsmanager.region.amazonaws.com</td>
<td>443</td>
<td>Yes</td>
<td>Download secrets to the core device.</td>
</tr>
</tbody>
</table>

Dependencies
When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 289) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.0.9
The following table lists the dependencies for version 2.0.9 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.8
The following table lists the dependencies for version 2.0.8 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>
2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.6

The following table lists the dependencies for version 2.0.6 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.4 - 2.0.5

The following table lists the dependencies for versions 2.0.4 and 2.0.5 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.3 &lt;2.1.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

cloudSecrets

A list of Secrets Manager secrets to deploy to the core device. You can specify labels to define which versions of each secret to deploy. If you don't specify a version, this component deploys the version with the staging label AWSCURRENT attached. For more information, see Staging labels in the AWS Secrets Manager User Guide.

Each object contains the following information:

- arn
  - The ARN of the secret to deploy.
- labels
  - (Optional) A list of labels to identify the versions of the secret to deploy to the core device.
  - Each label must be a string.

Example Example: Configuration merge update

```json
{
  "cloudSecrets": [
    {
```
Local log file

This component uses the same log file as the Greengrass nucleus (p. 136) component.

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace /greengrass/v2 with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.9</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.0.8</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.0.7</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>2.0.6</td>
<td>Version updated for Greengrass nucleus version 2.1.0 release.</td>
</tr>
</tbody>
</table>
| 2.0.5   | Improvements  
|         | • Add support for AWS China Regions and AWS GovCloud (US) Regions. |
| 2.0.4   | Initial version. |

Secure tunneling

The secure tunneling component (aws.greengrass.SecureTunneling) enables you to use AWS IoT secure tunneling to establish secure bidirectional communication with a Greengrass core device that is behind restricted firewalls.

For example, a Greengrass core device is behind a firewall that prohibits all incoming connections, which blocks SSH connections to the device. Secure tunneling uses MQTT over WebSockets to make these connections so that you can open an SSH connection to the device by using a tunnel that is managed by AWS IoT. For more information about using AWS IoT secure tunneling to connect to remote devices, see AWS IoT secure tunneling in the AWS IoT Developer Guide.

This component subscribes to the AWS IoT Core MQTT message broker on the $aws/things/greengrass-core-device/tunnels/notify topic to receive secure tunneling notifications.
Topics

- Versions (p. 290)
- Type (p. 290)
- Requirements (p. 290)
- Dependencies (p. 291)
- Configuration (p. 292)
- Local log file (p. 293)
- Licenses (p. 293)
- See also (p. 293)
- Changelog (p. 293)

Versions

This component has the following versions:

- 1.0.x

Type

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

- Python 3.5 or later installed on the Greengrass core device and added to the PATH environment variable.
- libcrypt.so.1.1 installed on the Greengrass core device and added to the PATH environment variable.
- The Greengrass core device must allow outbound traffic on port 443.
- The Greengrass core device must have enabled the service that you want to use to communicate with the device. For example, to open an SSH connection to the device, the device must have SSH enabled.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall (p. 640).

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data.tunneling.iot.&lt;region&gt;.amazoneAw438om</td>
<td>443</td>
<td>Yes</td>
<td>Establish secure tunnels.</td>
</tr>
</tbody>
</table>
## Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 293) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

### 1.0.4

The following table lists the dependencies for version 1.0.4 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

### 1.0.3

The following table lists the dependencies for version 1.0.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

### 1.0.2

The following table lists the dependencies for version 1.0.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

### 1.0.1

The following table lists the dependencies for version 1.0.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

### 1.0.0

The following table lists the dependencies for version 1.0.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.3 &lt;2.1.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).
Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

**OS_DIST_INFO**

(Optional) The operating system of your core device. By default, the component attempts to automatically identify the operating system running on your core device. If the component fails to start with the default value, then use this value to specify the operating system. For a list of supported operating systems for this component, see Requirements (p. 42).

This value can be one of the following: auto, ubuntu, amzn2, raspberrypi.

Default: auto

**accessControl**

(Optional) The object that contains the authorization policy (p. 403) that allows the component to subscribe to the secure tunneling notifications topic.

**Note**

Do not modify this configuration parameter if your deployment targets a thing group. If your deployment targets an individual core device, and you want to restrict this component's subscription to the topic for that device, then in the `resources` value in the authorization policy, replace the MQTT topic wildcard (+) with the thing name for that core device.

Default:

```
{
  "aws.greengrass.ipc.mqttproxy": {
    "aws.iot.SecureTunneling:mqttproxy:1": {
      "policyDescription": "Access to tunnel notification pubsub topic",
      "operations": [
        "aws.greengrass#SubscribeToIoTCore"
      ],
      "resources": [
        "$aws/things/+/tunnels/notify"
      ]
    }
  }
}
```

**Example Example: Configuration merge update**

The following example configuration specifies to allow this component to open secure tunnels on a core device named *MyGreengrassCore* that runs Ubuntu.

```
{
  "OS_DIST": "ubuntu",
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
      "aws.iot.SecureTunneling:mqttproxy:1": {
        "policyDescription": "Access to tunnel notification pubsub topic",
        "operations": [
          "aws.greengrass#SubscribeToIoTCore"
        ],
        "resources": [
          "$aws/things/+/tunnels/notify"
        ]
      }
    }
  }
}
```
Local log file

This component uses the following log file.

```
/greengrass/v2/logs/aws.greengrass.SecureTunneling.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.SecureTunneling.log
```

Licenses

This component includes the following third-party software/licensing:

- AWS IoT Device Client/Apache License 2.0
- AWS IoT Device SDK for Java/Apache License 2.0
- gson/Apache License 2.0
- log4j/Apache License 2.0
- slf4j/Apache License 2.0

See also

- AWS IoT secure tunneling in the AWS IoT Developer Guide.

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0.4</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>1.0.3</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>1.0.2</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>1.0.1</td>
<td>Version updated for Greengrass nucleus version 2.1.0 release.</td>
</tr>
<tr>
<td>1.0.0</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>
Shadow manager

The shadow manager component (`aws.greengrass.ShadowManager`) enables the local shadow service on your core device. The local shadow service allows components to use interprocess communication to interact with local shadows (p. 462). The shadow manager component manages the storage of local shadow documents, and also handles synchronization of local shadow states with the AWS IoT Device Shadow service.

For more information about how AWS IoT Greengrass devices can interact with shadows, see Interact with device shadows (p. 515).

Topics

- Versions (p. 294)
- Type (p. 294)
- Requirements (p. 294)
- Dependencies (p. 295)
- Configuration (p. 295)
- Local log file (p. 297)
- Changelog (p. 298)

Versions

This component has the following versions:

- 2.0.x

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus (p. 136) runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

- The Greengrass core device's AWS IoT policy configured to allow the following AWS IoT Core shadow policy actions that are required to sync shadows to the AWS IoT Device Shadow service.
  - `iot:GetThingShadow`
  - `iot:UpdateThingShadow`
  - `iot:DeleteThingShadow`

  For more information about these AWS IoT Core policies, see AWS IoT Core policy actions in the AWS IoT Developer Guide.

  For more information about the minimal AWS IoT policy, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices (p. 614)
Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 298) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.2.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.1 - 2.0.2

The following table lists the dependencies for versions 2.0.1 and 2.0.2 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.2.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

2.0.0

The following table lists the dependencies for version 2.0.0 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.2.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

synchronize

(Optional) The synchronization settings that determine how shadows are synced with the AWS Cloud.

**Note**
You must create a configuration update with this property if you want to enable syncing shadows with the AWS Cloud.

This object contains the following information.

coreThing

The core device shadows to sync. This object contains the following information.
classic

By default, the shadow manager syncs the local state of the classic shadow for your core device with the AWS Cloud. If you don't want to sync the classic device shadow, set this to false.

Default: true

namedShadows

The list of named core device shadows that you want to sync.

shadowDocuments

The list of additional device shadows to sync. Each object in this list contains the following information.

thingName

The thing name of the device for which to sync shadows.

classic

If you don't want to sync the classic device shadow for the thingName device, set this to false.

Default: true

namedShadows

The list of named device shadows that you want to sync.

rateLimits

The settings that determine the rate limits for shadow service requests.

This object contains the following information.

maxOutboundSyncUpdatesPerSecond

The maximum number of sync requests per second that the device transmits.

Default: 100 requests/second

maxTotalLocalRequestsRate

The maximum number of local IPC requests per second that are sent to the core device.

Default: 200 requests/second

maxLocalRequestsPerSecondPerThing

The maximum number of local IPC requests per second that are sent for each connected IoT thing.

Default: 20 requests/second for each thing

Note

These rate limits parameters define the maximum number of requests per second for the local shadow service. The maximum number of requests per second for the AWS IoT Device Shadow service depends on your AWS Region. For more information, see the limits for the AWS IoT Device Shadow Service API in the Amazon Web Services General Reference.

shadowDocumentSizeLimitBytes

The maximum allowed size of each JSON state document for local shadows.
Local log file

This component uses the same log file as the Greengrass nucleus (p. 136) component.

`/greengrass/v2/logs/greengrass.log`

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```
### Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.3</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
</tbody>
</table>
| 2.0.2   | Bug fixes and improvements  
|         | • Fixed an issue that caused shadow manager to not recognize the `delta` property when syncing shadow states from AWS IoT Core.  
|         | • Fixed an issue that sometimes caused sync requests for a shadow to be merged incorrectly. |
| 2.0.1   | Version updated for Greengrass nucleus version 2.3.0 release. |
| 2.0.0   | Initial version. |

### Amazon SNS

The Amazon SNS component (aws.greengrass.SNS) publishes messages to an Amazon Simple Notification Service (Amazon SNS) topic. You can use this component to send events from Greengrass core devices to web servers, email addresses, and other message subscribers. For more information, see What is Amazon SNS? in the Amazon Simple Notification Service Developer Guide.

To publish to an Amazon SNS topic with this component, publish a message to the topic where this component subscribes. By default, this component subscribes to the `sns/message` local publish/subscribe (p. 410) topic. You can specify other topics, including AWS IoT Core MQTT topics, when you deploy this component.

In your custom component, you might want to implement filtering or formatting logic to process messages from other sources before you publish them to this component. This enables you to centralize your message processing logic on a single component.

**Note**  
This component provides similar functionality to the Amazon SNS connector in AWS IoT Greengrass V1. For more information, see Amazon SNS connector in the AWS IoT Greengrass V1 Developer Guide.

**Topics**  
- Versions (p. 299)  
- Type (p. 299)  
- Requirements (p. 299)  
- Dependencies (p. 300)  
- Configuration (p. 302)  
- Input data (p. 304)  
- Output data (p. 305)  
- Local log file (p. 305)  
- Licenses (p. 306)  
- Changelog (p. 306)
### Versions

This component has the following versions:

- 2.0.x

### Type

This component is a Lambda component (\texttt{aws.greengrass.lambda}). The Greengrass nucleus (p. 136) runs this component's Lambda function using the Lambda launcher component (p. 188).

For more information, see Component types (p. 323).

### Requirements

This component has the following requirements:

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Requirements to run Lambda functions (p. 43).
- Python version 3.7 installed on the core device and added to the PATH environment variable.
- An Amazon SNS topic. For more information, see Creating an Amazon SNS topic in the Amazon Simple Notification Service Developer Guide.
- The Greengrass device role (p. 627) must allow the \texttt{sns:Publish} action, as shown in the following example IAM policy.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Action": ["sns:Publish"],
         "Effect": "Allow",
         "Resource": ["arn:aws:sns:region:account-id:topic-name"]
      }
   ]
}
```

You can dynamically override the default topic in the input message payload for this component. If your application uses this feature, the IAM policy must include all target topics as resources. You can grant granular or conditional access to resources (for example, by using a wildcard * naming scheme).

- To receive output data from this component, you must merge the following configuration update for the legacy subscription router component (p. 194) when you deploy this component. The legacy subscription router component (\texttt{aws.greengrass.LegacySubscriptionRouter}) is a dependency of this component. This configuration specifies the topic where this component publishes responses.

```json
{  
   "subscriptions": {  
      "aws-greengrass-sns": {  
         "id": "aws-greengrass-sns",
```
Legacy subscription router v2.0.x

```json
{
"subscriptions": {
  "aws-greengrass-sns": {
    "id": "aws-greengrass-sns",
    "subject": "sns/message/status",
    "target": "cloud"
  }
}
}
```

- Replace `region` with the AWS Region that you use.
- Replace `version` with the version of the Lambda function that this component runs. To find the Lambda function version, you must view the recipe for the version of this component that you want to deploy. Open this component's details page in the AWS IoT Greengrass console, and look for the Lambda function key-value pair. This key-value pair contains the name and version of the Lambda function.

**Important**

You must update the Lambda function version on the legacy subscription router every time you deploy this component. This ensures that you use the correct Lambda function version for the component version that you deploy.

For more information, see Create deployments (p. 384).

## Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall (p. 640).

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sns.region.amazonaws.com</code></td>
<td>443</td>
<td>Yes</td>
<td>Publish messages to Amazon SNS.</td>
</tr>
</tbody>
</table>

## Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 306) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.
2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.0.6

The following table lists the dependencies for version 2.0.6 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.0.5

The following table lists the dependencies for version 2.0.5 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>^2.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.0.4

The following table lists the dependencies for version 2.0.4 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>^2.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>
The following table lists the dependencies for version 2.0.3 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.3 &lt;2.1.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda launcher (p. 188)</td>
<td>&gt;=1.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Lambda runtimes (p. 193)</td>
<td>&gt;=1.0.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>&gt;=1.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

**Note**
This component's default configuration includes Lambda function parameters. We recommend that you edit only the following parameters to configure this component on your devices.

**lambdaParams**

An object that contains the parameters for this component's Lambda function. This object contains the following information:

**EnvironmentVariables**

An object that contains the Lambda function's parameters. This object contains the following information:

**DEFAULT_SNS_ARN**

The ARN of the default Amazon SNS topic where this component publishes messages. You can override the destination topic with the `sns_topic_arn` property in the input message payload.

**containerMode**

(Optional) The containerization mode for this component. Choose from the following options:

- NoContainer – The component doesn't run in an isolated runtime environment.
- GreengrassContainer – The component runs in an isolated runtime environment inside the AWS IoT Greengrass container.

Default: GreengrassContainer
containerParams

(Optional) An object that contains the container parameters for this component. The component uses these parameters if you specify `GreengrassContainer` for `containerMode`.

This object contains the following information:

memorySize

(Optional) The amount of memory (in kilobytes) to allocate to the component. Defaults to 512 MB (525,312 KB).

pubsubTopics

(Optional) An object that contains the topics where the component subscribes to receive messages. You can specify each topic and whether the component subscribes to MQTT topics from AWS IoT Core or local publish/subscribe topics.

This object contains the following information:

0 – This is an array index as a string.

An object that contains the following information:

type

(Optional) The type of publish/subscribe messaging that this component uses to subscribe to messages. Choose from the following options:

- Pubsub – Subscribe to local publish/subscribe messages. If you choose this option, the topic can't contain MQTT wildcards. For more information about how to send messages from custom component when you specify this option, see Publish/subscribe local messages (p. 410).
- IotCore – Subscribe to AWS IoT Core MQTT messages. If you choose this option, the topic can contain MQTT wildcards. For more information about how to send messages from custom components when you specify this option, see Publish/subscribe AWS IoT Core MQTT messages (p. 433).

Default: Pubsub

topic

(Optional) The topic to which the component subscribes to receive messages. If you specify `IotCore` for `type`, you can use MQTT wildcards (+ and #) in this topic.

Example Example: Configuration merge update (container mode)

```
{
    "lambdaExecutionParameters": {
        "EnvironmentVariables": {
            "DEFAULT_SNS_ARN": "arn:aws:sns:us-west-2:123456789012:mytopic"
        }
    },
    "containerMode": "GreengrassContainer"
}
```

Example Example: Configuration merge update (no container mode)

```
{
    "lambdaExecutionParameters": {
        "EnvironmentVariables": {
            "DEFAULT_SNS_ARN": "arn:aws:sns:us-west-2:123456789012:mytopic"
        }
    }
}
```
Input data

This component accepts messages on the following topic and publishes the message as is to the target Amazon SNS topic. By default, this component subscribes to local publish/subscribe messages. For more information about how to publish messages to this component from your custom components, see Publish/subcribe local messages (p. 410).

**Default topic (local publish/subscribe):** sns/message

The message accepts the following properties. Input messages must be in JSON format.

**request**

The information about the message to send to the Amazon SNS topic.

Type: object that contains the following information:

**message**

The content of the message as a string.

To send a JSON object, serialize it as a string, and specify `json` for the `message_structure` property.

Type: string

**subject**

(Optional) The subject of the message.

Type: string

The subject can be ASCII text and up to 100 characters. It must begin with a letter, number, or punctuation mark. It can't include line breaks or control characters.

**sns_topic_arn**

(Optional) The ARN of the Amazon SNS topic where this component publishes the message. Specify this property to override the default Amazon SNS topic.

Type: string

**message_structure**

(Optional) The structure of the message. Specify `json` to send a JSON message that you serialize as a string in the `content` property.

Type: string

Valid values: `json`

**id**

An arbitrary ID for the request. Use this property to map an input request to an output response. When you specify this property, the component sets the `id` property in the response object to this value.

Type: string
Note
The message size can be a maximum of 256 KB.

Example Example input: String message

```json
{
    "request": {
        "subject": "Message subject",
        "message": "Message data",
        "sns_topic_arn": "arn:aws:sns:region:account-id:topic2-name"
    },
    "id": "request123"
}
```

Example Example input: JSON message

```json
{
    "request": {
        "subject": "Message subject",
        "message": "{ "default": "Message data" }",
        "message_structure": "json"
    },
    "id": "request123"
}
```

Output data

This component publishes responses as output data on the following MQTT topic by default. You must specify this topic as the subject in the configuration for the legacy subscription router component (p. 194). For more information about how to subscribe to messages on this topic in your custom components, see Publish/subscribe AWS IoT Core MQTT messages (p. 433).

Default topic (AWS IoT Core MQTT): `sns/message/status`

Example Example output: Success

```json
{
    "response": {
        "sns_message_id": "f80a81bc-f44c-56f2-a0f0-d5af6a727c8a",
        "status": "success"
    },
    "id": "request123"
}
```

Example Example output: Failure

```json
{
    "response": {
        "error": "InvalidInputException",
        "error_message": "SNS Topic Arn is invalid",
        "status": "fail"
    },
    "id": "request123"
}
```

Local log file

This component uses the following log file.
Licenses

This component includes the following third-party software/licensing:

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF License
- docutils/BSD License, GNU General Public License (GPL), Python Software Foundation License, Public Domain
- jmespath/MIT License
- s3transfer/Apache License 2.0
- urllib3/MIT License

This component is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.7</td>
<td>Version updated for Greengrass nucleus version 2.4.0 release.</td>
</tr>
<tr>
<td>2.0.6</td>
<td>Version updated for Greengrass nucleus version 2.3.0 release.</td>
</tr>
<tr>
<td>2.0.5</td>
<td>Version updated for Greengrass nucleus version 2.2.0 release.</td>
</tr>
<tr>
<td>2.0.4</td>
<td>Version updated for Greengrass nucleus version 2.1.0 release.</td>
</tr>
<tr>
<td>2.0.3</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

Stream manager

The stream manager component (aws.greengrass.StreamManager) enables you to process data streams to transfer to the AWS Cloud from Greengrass core devices.

For more information about how to configure and use stream manager in custom components, see Manage data streams on the AWS IoT Greengrass Core (p. 519).

Topics

- Versions (p. 307)
Versions

This component has the following versions:

- 2.0.x

**Note**

If you use stream manager to export data to the cloud, you can't upgrade version 2.0.7 of the stream manager component to a version between v2.0.8 and v2.0.11. If you are deploying stream manager for the first time, we strongly recommend that you deploy the latest version of the stream manager component.

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

- The Authorize core devices to interact with AWS services (p. 627) must allow access to the AWS Cloud destinations that you use with stream manager. For more information, see:
  - the section called "AWS IoT Analytics channels" (p. 546)
  - the section called "Amazon Kinesis data streams" (p. 547)
  - the section called "AWS IoT SiteWise asset properties" (p. 548)
  - the section called "Amazon S3 objects" (p. 550)

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall (p. 640).

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iotanalytics.region.amazonaws.com</code> 443</td>
<td></td>
<td>No</td>
<td>Required if you publish data to AWS IoT Analytics.</td>
</tr>
</tbody>
</table>
## Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 311) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

### 2.0.12 and 2.0.11

The following table lists the dependencies for version 2.0.11 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.5.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>&gt;=0.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

### 2.0.10

The following table lists the dependencies for version 2.0.10 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.4.0</td>
<td>Soft</td>
</tr>
</tbody>
</table>
The following table lists the dependencies for version 2.0.9 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.3.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>&gt;=0.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

The following table lists the dependencies for version 2.0.8 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.0 &lt;2.2.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>&gt;=0.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

The following table lists the dependencies for version 2.0.7 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.0.3 &lt;2.1.0</td>
<td>Soft</td>
</tr>
<tr>
<td>Token exchange service (p. 311)</td>
<td>&gt;=0.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

**Configuration**

This component provides the following configuration parameters that you can customize when you deploy the component.

**STREAM_MANAGER_STORE_ROOT_DIR**

(Optional) The absolute path of the local directory used to store streams. This value must start with a forward slash (for example, /data).

**STREAM_MANAGER_SERVER_PORT**

(Optional) The local port number to use to communicate with stream manager.
You can specify 0 to use a random available port.

Default: 8088

STREAM_MANAGER_AUTHENTICATE_CLIENT

(Optional) You can make it mandatory for clients to authenticate before they can interact with stream manager. The AWS IoT Greengrass Core SDK controls interaction between clients and stream manager. This parameter determines which clients can call the AWS IoT Greengrass Core SDK to work with streams. For more information, see stream manager client authentication (p. 521).

If you specify true, the AWS IoT Greengrass Core SDK allows only Greengrass components as clients.

If you specify false, the AWS IoT Greengrass Core SDK allows all processes on the core device to be clients.

Default: true

STREAM_MANAGER_EXPORTER_MAX_BANDWIDTH

(Optional) The average maximum bandwidth (in kilobits per second) that stream manager can use to export data.

Default: No limit

STREAM_MANAGER_THREAD_POOL_SIZE

(Optional) The maximum number of active threads that stream manager can use to export data.

The optimal size depends on your hardware, stream volume, and planned number of export streams. If your export speed is slow, you can adjust this setting to find the optimal size for your hardware and business case. The CPU and memory of your core device hardware are limiting factors. To start, you might try setting this value equal to the number of processor cores on the device.

Be careful not to set a size that's higher than your hardware can support. Each stream consumes hardware resources, so try to limit the number of export streams on constrained devices.

Default: 5 threads

STREAM_MANAGER_EXPORTER_S3_DESTINATION_MULTIPART_UPLOAD_MIN_PART_SIZE_BYTES

(Optional) The minimum size (in bytes) of a part in a multipart upload to Amazon S3. Stream manager uses this setting and the size of the input file to determine how to batch data in a multipart PUT request.

Note
Stream manager uses the streams sizeThresholdForMultipartUploadBytes property to determine whether to export to Amazon S3 as a single or multipart upload. AWS IoT Greengrass components can set this threshold when they create a stream that exports to Amazon S3.

Default: 5242880 (5 MB). This is also the minimum value.

JVM_ARGS

(Optional) The custom Java Virtual Machine arguments to pass to stream manager at startup. Separate multiple arguments by spaces.

Use this parameter only when you must override the default settings used by the JVM. For example, you might need to increase the default heap size if you plan to export a large number of streams.

Example Example: Configuration merge update

The following example configuration specifies to use a non-default port.
Local log file

This component uses the following log file.

```
{  
    "STREAM_MANAGER_SERVER_PORT": "18088"
}
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.StreamManager.log
```

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2.0.12  | Bug fixes and improvements  
          Fixed an issue that prevented upgrading stream manager v2.0.7 to a version between v2.0.8 and v2.0.11. If you use stream manager to export data to the cloud, you can now upgrade to v2.0.12. |
| 2.0.11  | Version updated for Greengrass nucleus version 2.4.0 release. |
| 2.0.10  | Version updated for Greengrass nucleus version 2.3.0 release. |
| 2.0.9   | Version updated for Greengrass nucleus version 2.2.0 release. |
| 2.0.8   | Version updated for Greengrass nucleus version 2.1.0 release. |
| 2.0.7   | Initial version. |

Token exchange service

The token exchange service component (`aws.greengrass.TokenExchangeService`) provides AWS credentials that you can use to interact with AWS services in your custom components.

The token exchange service runs an Amazon Elastic Container Service (Amazon ECS) container instance as a local server. This local server connects to the AWS IoT credentials provider using the AWS IoT role alias that you configure in the Greengrass core nucleus component (p. 136). The component provides an environment variable, `AWS_CONTAINER_CREDENTIALS_FULL_URI`, that defines the URI to this local server. When a component creates an AWS SDK client, the client recognizes this URI environment variable to connect to the token exchange service and retrieve AWS credentials. This allows Greengrass to...
core devices to call AWS service operations. For more information about how to use this component in custom components, see Interact with AWS services (p. 329).

**Important**
Support to acquire AWS credentials in this way was added to the AWS SDKs on July 13th, 2016. Your component must use an AWS SDK version that was created on or after that date. For more information, see Using a supported AWS SDK in the Amazon Elastic Container Service Developer Guide.

**Topics**
- Versions (p. 312)
- Type (p. 312)
- Dependencies (p. 312)
- Configuration (p. 312)
- Local log file (p. 312)
- Changelog (p. 313)

**Versions**
This component has the following versions:
- 2.0.x

**Type**
This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

**Dependencies**
This component doesn't have any dependencies.

**Configuration**
This component doesn't have any configuration parameters.

**Local log file**
This component uses the same log file as the Greengrass nucleus (p. 136) component.

`/greengrass/v2/logs/greengrass.log`

**To view this component's logs**
- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```
Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.3</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

IoT SiteWise OPC-UA collector

The IoT SiteWise OPC-UA collector component (aws.iot.SiteWiseEdgeCollectorOpcua) enables AWS IoT SiteWise gateways to collect data from local OPC-UA servers.

With this component, AWS IoT SiteWise gateways can connect to multiple OPC-UA servers. For more information about AWS IoT SiteWise gateways, see Using AWS IoT SiteWise at the edge in the AWS IoT SiteWise User Guide.

Topics
- Versions (p. 313)
- Type (p. 313)
- Requirements (p. 313)
- Dependencies (p. 314)
- Configuration (p. 314)
- Output data (p. 314)
- Local log file (p. 314)
- Licenses (p. 314)
- Changelog (p. 314)
- See also (p. 315)

Versions

This component has the following versions:
- 2.0.x

Type

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component’s lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:
- The Greengrass core device must allow outbound network connectivity to OPC-UA servers.
Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 314) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

The following table lists the dependencies for version 2.0.x of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.3.0 &lt;3.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Stream manager (p. 306)</td>
<td>&gt;2.0.10&lt;3.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Secret manager (p. 285)</td>
<td>&gt;=2.0.8 &lt;3.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component doesn't have any configuration parameters.

Output data

This component writes BatchPutAssetPropertyValue messages to AWS IoT Greengrass stream manager. For more information, see BatchPutAssetPropertyValue in the AWS IoT SiteWise API Reference.

Local log file

This component uses the following log file.

```
/greengrass/v2/logs/aws.iot.SiteWiseEdgeCollectorOpcua.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace /greengrass/v2 with the path to the AWS IoT Greengrass root folder.

  ```
  sudo tail -f /greengrass/v2/logs/aws.iot.SiteWiseEdgeCollectorOpcua.log
  ```

Licenses

This component is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the component.
### See also

- What is AWS IoT SiteWise? in the *AWS IoT SiteWise User Guide*.

---

### IoT SiteWise publisher

The IoT SiteWise publisher component (`aws.iot.SiteWiseEdgePublisher`) enables AWS IoT SiteWise gateways to export data from the edge to the AWS Cloud.

For more information about AWS IoT SiteWise gateways, see Using AWS IoT SiteWise at the edge in the *AWS IoT SiteWise User Guide*.

#### Topics

- Versions (p. 315)
- Type (p. 315)
- Requirements (p. 315)
- Dependencies (p. 316)
- Configuration (p. 316)
- Input data (p. 317)
- Local log file (p. 317)
- Licenses (p. 317)
- Changelog (p. 317)
- See also (p. 317)

---

### Versions

This component has the following versions:

- 2.0.x

---

### Type

This component is a generic component (`aws.greengrass.generic`). The *Greengrass nucleus* (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

---

### Requirements

This component has the following requirements:

- The Greengrass core device must connect to the Internet.
• The Greengrass core device must be authorized to perform the iotsitewise:BatchPutAssetPropertyValue action. For more information, see Authorize core devices to interact with AWS services.

Example permissions policy

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": "iotsitewise:BatchPutAssetPropertyValue",
         "Resource": "*"
      }
   ]
}
```

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall (p. 640).

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data.iotsitewise.region.amazonaws.dd55</td>
<td>443</td>
<td>Yes</td>
<td>Publish data to AWS IoT SiteWise.</td>
</tr>
</tbody>
</table>

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions (p. 317) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

The following table lists the dependencies for version 2.0.1 of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass nucleus (p. 136)</td>
<td>&gt;=2.3.0&lt;3.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Stream manager (p. 306)</td>
<td>&gt;=2.0.10&lt;3.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.
Input data

This component reads BatchPutAssetPropertyValue messages from AWS IoT Greengrass stream manager. For more information, see BatchPutAssetPropertyValue in the AWS IoT SiteWise API Reference.

Local log file

This component uses the following log file.

```
/greengrass/v2/logs/aws.iot.SiteWiseEdgePublisher.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/aws.iot.SiteWiseEdgePublisher.log
```

Licenses

This component is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.1</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

See also

- What is AWS IoT SiteWise? in the AWS IoT SiteWise User Guide.

IoT SiteWise processor

The IoT SiteWise processor component (`aws.iot.SiteWiseEdgeProcessor`) enables AWS IoT SiteWise gateways to process data at the edge.

With this component, AWS IoT SiteWise gateways can use asset models and assets to process data on gateway devices. For more information about AWS IoT SiteWise gateways, see Using AWS IoT SiteWise at the edge in the AWS IoT SiteWise User Guide.

Topics

- Versions (p. 318)
- Type (p. 318)
• Requirements (p. 318)
• Dependencies (p. 319)
• Configuration (p. 320)
• Local log file (p. 320)
• Licenses (p. 320)
• Changelog (p. 321)
• See also (p. 321)

Versions

This component has the following versions:

• 2.0.x

Type

This component is a generic component (aws.greengrass.generic). The Greengrass nucleus (p. 136) runs the component's lifecycle scripts.

For more information, see Component types (p. 323).

Requirements

This component has the following requirements:

• The Greengrass core device must run on one of the following platforms:
  • OS: Ubuntu 20.04 or 18.04
    Architecture: x86_64 (AMD64)
  • OS: Red Hat Enterprise Linux (RHEL) 8
    Architecture: x86_64 (AMD64)
  • OS: Amazon Linux 2
    Architecture: x86_64 (AMD64)
• The Greengrass core device must allow inbound traffic on port 443.
• The Greengrass core device must allow outbound traffic on port 443 and 8883.
• The following ports are reserved for use by AWS IoT SiteWise: 80, 443, 3001, 8000, 8081, 8082, 8084, 8085, 8445, 8086, 9000, 9500, and 11080. Using a reserved port for traffic can result in a terminated connection.
• The Greengrass device role must have permissions that allow you to use AWS IoT SiteWise gateways on your AWS IoT Greengrass V2 devices. For more information, see Requirements in the AWS IoT SiteWise User Guide.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall (p. 640).
### Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>model.iotsitewise.region.amazonaws.com:443</code></td>
<td></td>
<td>Yes</td>
<td>Get information about your AWS IoT SiteWise assets and asset models.</td>
</tr>
<tr>
<td><code>edge.iotsitewise.region.amazonaws.com:443</code></td>
<td></td>
<td>Yes</td>
<td>Get information about the core device's AWS IoT SiteWise gateway configuration.</td>
</tr>
<tr>
<td><code>ecr.region.amazonaws.com</code></td>
<td>443</td>
<td>Yes</td>
<td>Download AWS IoT SiteWise Edge gateway Docker images from Amazon Elastic Container Registry.</td>
</tr>
<tr>
<td><code>iot.region.amazonaws.com</code></td>
<td>443</td>
<td>Yes</td>
<td>Get device endpoints for your AWS account.</td>
</tr>
<tr>
<td><code>sts.region.amazonaws.com</code></td>
<td>443</td>
<td>Yes</td>
<td>Get the ID of your AWS account.</td>
</tr>
<tr>
<td><code>monitor.iotsitewise.region.amazonaws.com:443</code></td>
<td></td>
<td>No</td>
<td>Required if you access AWS IoT SiteWise Monitor portals on the core device.</td>
</tr>
</tbody>
</table>

---

319
versions (p. 321) of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the Dependencies list.

The following table lists the dependencies for versions 2.0.x of this component.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Compatible versions</th>
<th>Dependency type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token exchange service (p. 311)</td>
<td>&gt;=2.0.3 &lt;3.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Stream manager (p. 306)</td>
<td>&gt;=2.0.10 &lt;3.0.0</td>
<td>Hard</td>
</tr>
<tr>
<td>Greengrass CLI (p. 166)</td>
<td>&gt;=2.3.0 &lt;3.0.0</td>
<td>Hard</td>
</tr>
</tbody>
</table>

For more information about component dependencies, see the component recipe reference (p. 343).

Configuration

This component doesn't have any configuration parameters.

Local log file

This component uses the following log file.

```
/greengrass/v2/logs/aws.iot.SiteWiseEdgeProcessor.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace /greengrass/v2 with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/aws.iot.SiteWiseEdgeProcessor.log
```

Licenses

This component includes the following third-party software/licensing:

- Apache-2.0
- MIT
- BSD-2-Clause
- BSD-3-Clause
- CDDL-1.0
- CDDL-1.1
- ISC
- Zlib
- GPL-3.0-with-GCC-exception
- Public Domain
- Python-2.0
- Unicode-DFS-2015
• BSD-1-Clause
• OpenSSL
• EPL-1.0
• EPL-2.0
• GPL-2.0-with-classpath-exception
• MPL-2.0
• CCO-1.0
• JSON

This component is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the component.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0.2</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>

See also

• What is AWS IoT SiteWise? in the AWS IoT SiteWise User Guide.
Develop AWS IoT Greengrass components

AWS IoT Greengrass components are software modules that you deploy to Greengrass core devices. Components can represent applications, runtime installers, libraries, or any code that you would run on a device. You can define components that depend on other components. For example, you might define a component that installs Python, and then define that component as a dependency of your components that run Python applications. When you deploy your components to your fleets of devices, Greengrass deploys only the software modules that your devices require.

You can develop and test components on your Greengrass core device. This lets you create and iterate your AWS IoT Greengrass software without interaction with the AWS Cloud. When you finish a version of your component, you can upload it to AWS IoT Greengrass in the cloud, so you and your team can deploy the component to other devices in your fleet. For more information about how to deploy components, see Deploy AWS IoT Greengrass components to devices (p. 383).

Every component is composed of a recipe and artifacts.

- **Recipes**

  Every component contains a recipe file, which defines its metadata. The recipe also specifies the component's configuration parameters, component dependencies, lifecycle, and platform compatibility. The component lifecycle defines the commands that install, run, and shut down the component. For more information, see AWS IoT Greengrass component recipe reference (p. 341).

  You can define recipes in JSON or YAML format.

- **Artifacts**

  Components can have any number of artifacts, which are component binaries. Artifacts can include scripts, compiled code, static resources, and any other files that a component consumes. Components can also consume artifacts from component dependencies.

AWS IoT Greengrass provides pre-built components that you can use in your applications and deploy to your devices. For example, you can use the stream manager component to upload data to various AWS services, or you can use the CloudWatch metrics component to publish custom metrics to Amazon CloudWatch. For more information, see AWS-provided components (p. 132).

The AWS IoT Greengrass Core software runs components as the system user and group, such as ggc_user and ggc_group, that you configure on the core device. This means that components have the permissions of that system user. If you use a system user without a home directory, then components can't use run commands or code that use a home directory. This means that you can't use the pip install some-library --user command to install Python packages for example. If you followed the getting started tutorial (p. 24) to set up your core device, then your system user doesn't have a home directory. For more information about how to configure the user and group that run components, see Configure the user and group that run components (p. 122).

**Note**

AWS IoT Greengrass uses semantic versions for components. Semantic versions follow a major.minor.patch number system. For example, version 1.0.0 represents the first major release for a component. For more information, see the semantic version specification.
Component lifecycle

The *component lifecycle* defines the stages that the AWS IoT Greengrass Core software uses to install and run components. Each stage defines a script and other information that specifies how the component behaves. For example, when you install a component, the AWS IoT Greengrass Core software runs the `Install` lifecycle script for that component. Components on core devices have the following lifecycle states:

- **NEW** – The component's recipe and artifacts are loaded on the core device, but the component isn’t installed. After a component enters this state, it runs its `bootstrap` (p. 346) and `install scripts` (p. 347).
- **INSTALLED** – The component is installed on the core device. The component enters this state after it runs its `install script` (p. 347).
- **STARTING** – The component is starting on the core device. The component enters this state when it runs its `startup script` (p. 348). If the startup succeeds, the component enters the `RUNNING` state.
- **RUNNING** – The component is running on the core device. The component enters this state when it runs its `run script` (p. 347) or when it has active background processes from its startup script.
- **FINISHED** – The component ran successfully and is no longer running.
- **STOPPING** – The component is stopping. The component enters this state when it runs its `shutdown script` (p. 349).
- **ERRORED** – The component encountered an error. When the component enters this state, it runs its `recover script` (p. 350). Then, the component restarts to try to return to normal use. If the component enters the `ERRORED` state three times without a successful run, the component becomes `BROKEN`.
- **BROKEN** – The component encountered errors multiple times and can’t recover. You must deploy the component again to fix it.

Component types

The *component type* specifies how the AWS IoT Greengrass Core software runs the component. Components can have the following types:

- **Nucleus** (*aws.greengrass.nucleus*)

  The Greengrass nucleus is the component that provides the minimum functionality of the AWS IoT Greengrass Core software. For more information, see *Greengrass nucleus* (p. 136).

- **Plugin** (*aws.greengrass.plugin*)
Create local components

The Greengrass nucleus runs a plugin component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change the version of a plugin component on a core device. To install and run plugin components, you must configure the Greengrass nucleus to run as a system service. For more information, see Configure AWS IoT Greengrass as a system service (p. 119).

Several components that are provided by AWS are plugin components, which enables them to interface directly with the Greengrass nucleus. Plugin components use the same log file as the Greengrass nucleus. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

- **Generic** (aws.greengrass.generic)
  The Greengrass nucleus runs a generic component's lifecycle scripts, if the component defines a lifecycle.
  
  This type is the default type for custom components.

- **Lambda** (aws.greengrass.lambda)
  The Greengrass nucleus runs a Lambda function component using the Lambda launcher component (p. 188).
  
  When you create a component from a Lambda function, the component has this type. For more information, see Run AWS Lambda functions (p. 361).

  **Note**
  We don't recommend that you specify the component type in a recipe. AWS IoT Greengrass sets the type for you when you create a component.

Create local AWS IoT Greengrass components

With AWS IoT Greengrass, you can develop and test components on your Greengrass core device. To develop a new component or a new version of a component, create a recipe and artifacts for your new component version.

**To develop a component on a Greengrass core device.**

1. Create a folder for your component with subfolders for recipes and artifacts. Run the following commands on your Greengrass core device to create these folders and change to the component folder. Replace `~/greengrassv2` with the path to the folder to use for local development.

   ```bash
   mkdir -p ~/greengrassv2/{recipes,artifacts}
   cd ~/greengrassv2
   ```

2. Create a recipe that defines your component's metadata, parameters, dependencies, lifecycle, and platform capability. Run the following command to create the recipe file and open it in the nano text editor. Include the component version in the recipe file name so that you can identify which recipe reflects which component version. You can choose YAML or JSON format for your recipe.

   **JSON**
   
   ```bash
   nano recipes/com.example.HelloWorld-1.0.0.json
   ```

   **YAML**
   
   ```bash
   nano recipes/com.example.HelloWorld-1.0.0.yaml
   ```
Note
AWS IoT Greengrass uses semantic versions for components. Semantic versions follow a
major.minor.patch number system. For example, version 1.0.0 represents the first major
release for a component. For more information, see the semantic version specification.

3. Define the recipe for your component. For more information, see AWS IoT Greengrass component
recipe reference (p. 341).

Your recipe might look similar to the following Hello World example recipe.

**JSON**

```json
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.HelloWorld",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "My first AWS IoT Greengrass component.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "world"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "Run": "python3 -u {artifacts:path}/hello_world.py '{configuration:/Message}'"
      }
    }
  ]
}
```

**YAML**

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: My first AWS IoT Greengrass component.
ComponentPublisher: Amazon
ComponentConfiguration: 
  DefaultConfiguration: 
    Message: world
Manifests: 
- Platform: 
  os: linux
Lifecycle: 
  Run: |
    python3 -u {artifacts:path}/hello_world.py '{configuration:/Message}'
```

This recipe runs a Hello World Python script, which might look similar to the following example
script.

```python
import sys
import datetime
```
Test AWS IoT Greengrass components with local deployments

When you are ready to test your component, you can create a local deployment to install it to your Greengrass core device.

To test a component on a Greengrass core device

1. The core device logs events such as component updates. You can view this log file to discover and troubleshoot errors with your component, such as an invalid recipe. This log file also displays messages that your component prints to standard out (stdout). We recommend that you open an additional terminal session on your core device to observe new log messages in real time. Open a new terminal session, such as through SSH, and run the following command to view the logs. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

You can also view the log file for your component.

```
sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
```

2. In your original terminal session, run the following command to update the core device with your component. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder, and replace `~/greengrassv2` with the path to your local development folder.

```
sudo /greengrass/v2/bin/greengrass-cli deployment create \n  --recipeDir ~/greengrassv2/recipes \\n```
Upload components to deploy

```
--artifactDir ~/greengrassv2/artifacts \
--merge "com.example.HelloWorld=1.0.0"
```

**Note**
You can also use the greengrass-cli deployment create command to set the value of your component's configuration parameters. For more information, see create (p. 600).

3. Test your component as it runs on the Greengrass core device. When you finish this version of your component, you can upload it to the AWS IoT Greengrass service. Then, you can deploy the component to other core devices. For more information, see Upload components to deploy to your core devices (p. 327).

**Upload components to deploy to your core devices**

After you finish a version of a component, you can upload your component to AWS IoT Greengrass in the AWS Cloud. Then you can deploy components to other core devices.

When you upload a component, you do the following:

1. Upload component artifacts to an S3 bucket.
2. Add each artifact's Amazon S3 URI to the component recipe.
3. Create a component version in AWS IoT Greengrass from the component recipe.

**Note**
Each component version that you upload must be unique. You can't edit a component version after you upload it.

**Upload a component (AWS CLI)**

Use the following procedure to upload a component with the AWS CLI. You can follow these steps on your development computer or on your AWS IoT Greengrass Core device.

**To upload a component (AWS CLI)**

1. If the component uses a version that exists in the AWS IoT Greengrass service, then you must change the version of the component. Open the recipe in a text editor, increment the version, and save the file. Choose a new version that reflects the changes that you made to the component.

   **Note**
   AWS IoT Greengrass uses semantic versions for components. Semantic versions follow a major.minor.patch number system. For example, version 1.0.0 represents the first major release for a component. For more information, see the semantic version specification.

2. If your component has artifacts, do the following:
   a. Upload the component's artifacts to an S3 bucket in your AWS account.

      **Tip**
      We recommend that you include the component name and version in the path to the artifact in the S3 bucket. This naming scheme can help you maintain the artifacts that previous versions of the component use, so you can continue to support previous component versions.

      Run the following command to upload an artifact file to an S3 bucket. Replace `DOC-EXAMPLE-BUCKET` with the name of the bucket, and replace `artifacts/com.example.HelloWorld/1.0.0/artifact.py` with the path to the artifact file.
aws s3 cp artifacts/com.example.HelloWorld/1.0.0/artifact.py s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/artifact.py

**Important**

Core device roles don't allow access to S3 buckets by default. If this is your first time using this S3 bucket, you must add permissions to the role to allow core devices to retrieve component artifacts from this S3 bucket. For more information, see Allow access to S3 buckets for component artifacts (p. 628).

b. Add a list named **Artifacts** to the component recipe if it isn't present. The **Artifacts** list appears in each manifest, which defines the component's requirements on each platform that it supports (or the component's default requirements for all platforms).

c. Add each artifact to the list of artifacts, or update the URI of existing artifacts. The Amazon S3 URI is composed of the bucket name and the path to the artifact object in the bucket. Your artifacts' Amazon S3 URIs should look similar to the following example.

```
s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.HelloWorld/1.0.0/artifact.py
```

After you complete these steps, your recipe should have an **Artifacts** list that looks like the following.

**JSON**

```
...
"Manifests": [  
{  
"Lifecycle": {  
...  
}  
"Artifacts": [  
{  
"URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/MyGreengrassComponent/1.0.0/artifact.py",  
"Unarchive": "NONE"  
}  
}  
]  
}
```

**Note**

You can add the "Unarchive": "ZIP" option for a ZIP artifact to configure the AWS IoT Greengrass Core software to unzip the artifact when the component deploys.

**YAML**

```
...
Manifests:  
- Lifecycle:  
  ...
- Artifacts:  
  - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/MyGreengrassComponent/1.0.0/artifact.py
  Unarchive: NONE
```
**Note**
You can use the **Unarchive: ZIP** option to configure the AWS IoT Greengrass Core software to unzip a ZIP artifact when the component deploys. For more information about how to use ZIP artifacts in a component, see the `artifacts:decompressedPath` recipe variable (p. 355).

For more information about recipes, see AWS IoT Greengrass component recipe reference (p. 341).

3. Use the AWS IoT Greengrass console to create a component from the recipe file.

Run the following command to create the component from a recipe file. This command creates the component and publishes it as a private AWS IoT Greengrass component in your AWS account. Replace `path/to/recipeFile` with the path to the recipe file.

```bash
aws greengrassv2 create-component-version  \
   --inline-recipe fileb://path/to/recipeFile
```

Copy the **arn** from the response to check the state of the component in the next step.

**Note**
AWS IoT Greengrass computes the digest of each artifact when you create the component. This means that you can't modify the artifact files in your S3 bucket after you create a component. If you do, deployments that include this component will fail, because the file digest doesn't match. If you modify an artifact file, you must create a new version of the component.

4. Each component in the AWS IoT Greengrass service has a state. Run the following command to confirm the state of the component version that you upload in this procedure. Replace `com.example.HelloWorld` and `1.0.0` with the component version to query. Replace the **arn** with the ARN from the previous step.

```bash
aws greengrassv2 describe-component  \
```

The operation returns a response that contains the component's metadata. The metadata contains a `status` object that contains the component state and any errors, if applicable.

When the component state is **DEPLOYABLE**, you can deploy the component to devices. For more information, see Deploy AWS IoT Greengrass components to devices (p. 383).

### Interact with AWS services

Greengrass core devices use X.509 certificates to connect to AWS IoT Core using TLS mutual authentication protocols. These certificates let devices interact with AWS IoT without AWS credentials, which typically comprise an access key ID and a secret access key. Other AWS services require AWS credentials instead of X.509 certificates to call API operations at service endpoints. AWS IoT Core has a credentials provider that enables devices to use their X.509 certificate to authenticate AWS requests. The AWS IoT credentials provider authenticates devices using an X.509 certificate and issues AWS credentials in the form a temporary, limited-privilege security token. Devices can use this token to sign and authenticate any AWS request. This eliminates the need to store AWS credentials on Greengrass core devices. For more information, see Authorizing direct calls to AWS services in the AWS IoT Core Developer Guide.

To fetch credentials from AWS IoT, Greengrass, core devices use an AWS IoT role alias that points to an IAM role. This IAM role is called the **token exchange role**. You create the role alias and token exchange role
when you install the AWS IoT Greengrass Core software. To specify the role alias that a core device uses, configure the `iotRoleAlias` parameter of the Greengrass nucleus (p. 136).

The AWS IoT credentials provider assumes the token exchange role on your behalf to provide AWS credentials to core devices. You can attach appropriate IAM policies to this role to allow your core devices access to your AWS resources, such as components artifacts in S3 buckets. For more information about how to configure the token exchange role, see Authorize core devices to interact with AWS services (p. 627).

Greengrass core devices store AWS credentials in memory, and the credentials expire after an hour by default. If the AWS IoT Greengrass Core software restarts, it must fetch credentials again. You can use the `updateRoleAlias` operation to configure the duration that credentials are valid.

AWS IoT Greengrass provides a public component, the token exchange service component, that you can define as a dependency in your custom component to interact with AWS services. The token exchange service provides your component with an environment variable, `AWS_CONTAINER_CREDENTIALS_FULL_URI`, that defines the URI to a local server that provides AWS credentials. When you create an AWS SDK client, the client checks for this environment variable and connects to the local server to retrieve AWS credentials and uses them to sign API requests. This lets you use AWS SDKs and other tools to call AWS services in your components. For more information, see Token exchange service (p. 311).

Important
Support to acquire AWS credentials in this way was added to the AWS SDKs on July 13th, 2016. Your component must use an AWS SDK version that was created on or after that date. For more information, see Using a supported AWS SDK in the Amazon Elastic Container Service Developer Guide.

To acquire AWS credentials in your custom component, define `aws.greengrass.TokenExchangeService` as a dependency in the component recipe. The following example recipe defines a component that installs `boto3` and runs a Python script that uses AWS credentials from the token exchange service to list Amazon S3 buckets.

Note
To run this example component, your device must have the `s3:ListAllMyBuckets` permission. For more information, see Authorize core devices to interact with AWS services (p. 627).

JSON

```json
{
    "RecipeFormatVersion": "2020-01-25",
    "ComponentName": "com.example.ListS3Buckets",
    "ComponentVersion": "1.0.0",
    "ComponentDescription": "A component that uses the token exchange service to list S3 buckets.",
    "ComponentPublisher": "Amazon",
    "ComponentDependencies": {
        "aws.greengrass.TokenExchangeService": {
            "VersionRequirement": "^2.0.0",
            "DependencyType": "HARD"
        }
    },
    "Manifests": [
        {
            "Lifecycle": {
                "Install": "pip3 install --user boto3",
                "Run": "python3 -u {artifacts:path}/list_s3_buckets.py"
            }
        }
    ]
}
```
Run a Docker container

You can configure AWS IoT Greengrass components to run a Docker container from images stored in the following locations:

- Public and private image repositories in Amazon Elastic Container Registry (Amazon ECR)
- Public Docker Hub repository
- Public Docker Trusted Registry
- S3 bucket

In your custom component, include the Docker image URI as an artifact to retrieve the image and run it on the core device. For Amazon ECR and Docker Hub images, you can use the Docker application.
manager (p. 162) component to download the images and manage credentials for private Amazon ECR repositories.

**Topics**

- Requirements (p. 332)
- Run a Docker container from a public image in Amazon ECR or Docker Hub (p. 333)
- Run a Docker container from a private image in Amazon ECR (p. 335)
- Run a Docker container from an image in Amazon S3 (p. 337)
- Use interprocess communication in Docker container components (p. 339)

**Requirements**

To run a Docker container in a component, you need the following:

- A Greengrass core device. If you don't have one, see Getting started with AWS IoT Greengrass V2 (p. 24).
- Docker Engine 1.9.1 or later installed on your Greengrass core device. Version 20.10 is the latest version that is verified to work with the connector. You must install Docker directly on the core device before you deploy custom components that run Docker containers.
- Root user permissions or Docker configured for you to run it as a non-root user. Adding a user to the docker group enables you to call [docker](https://docs.docker.com/) commands without sudo. To add ggc_user, or the non-root user that you use to run AWS IoT Greengrass, to the docker group that you configure, run `sudo usermod -aG docker user-name`.
- Files accessed by the Docker container component mounted as a volume in the Docker container.

In addition to these requirements, you must also meet the following requirements if they apply to your environment:

- To use [Docker Compose](https://docs.docker.com/compose/) to create and start your Docker containers, install Docker Compose on your Greengrass core device, and upload your Docker Compose file to an S3 bucket. You must store your Compose file in an S3 bucket in the same AWS account and AWS Region as the component. For an example that uses the `docker-compose up` command in a custom component, see Run a Docker container from a public image in Amazon ECR or Docker Hub (p. 333).
- If you run AWS IoT Greengrass behind a network proxy, configure the Docker daemon to use a proxy server.
- If your Docker images are stored in Amazon ECR or Docker Hub, include the Docker component manager (p. 162) component as a dependency in your Docker container component. You must start the Docker daemon on the core device before you deploy your component.

Also, include the image URIs as component artifacts. Image URIs must be in the format `docker:registry/image[:tag|@digest]` as shown in the following examples:

- Private Amazon ECR image: `docker:account-id.dkr.ecr.region.amazonaws.com/repository/image[:tag|@digest]`
- Public Amazon ECR image: `docker:public.ecr.aws/repository/image[:tag|@digest]`
- Public Docker Hub image: `docker:name[:tag|@digest]`

If your Docker images are stored in an Amazon ECR private repository, then you must include the token exchange service component as a dependency in the Docker container component. Also, the Greengrass device role (p. 627) must allow the `ecr:GetAuthorizationToken`, `ecr:BatchGetImage`, and `ecr:GetDownloadUrlForLayer` actions, as shown in the following example IAM policy.

```
{  
```
Run a Docker container from a public image in Amazon ECR or Docker Hub

This section describes how you can create a custom component that uses Docker Compose to run a Docker container from Docker images that are stored Amazon ECR and Docker Hub.

To run a Docker container using Docker Compose

1. Create and upload a Docker Compose file to an Amazon S3 bucket. Make sure that the Greengrass device role (p. 627) allows the s3:GetObject permission to enable the device to access the Compose file. The example Compose file shown in the following example includes the Amazon CloudWatch Agent image from Amazon ECR and the MySQL image from Docker Hub.
Run a Docker container from a public image in Amazon ECR or Docker Hub

2. Create a custom component (p. 324) on your AWS IoT Greengrass core device. The example recipe shown in the following example has the following properties:

- The Docker application manager component as a dependency. This component enables AWS IoT Greengrass to download images from public Amazon ECR and Docker Hub repositories.
- A component artifact that specifies a Docker image in a public Amazon ECR repository.
- A component artifact that specifies a Docker image in a public Docker Hub repository.
- A component artifact that specifies the Docker Compose file that includes containers for the Docker images that you want to run.
- A lifecycle run script that uses `docker-compose up` to create and start a container from the specified images.

JSON

```json
{
    "RecipeFormatVersion": "2020-01-25",
    "ComponentName": "com.example.MyDockerComposeComponent",
    "ComponentVersion": "1.0.0",
    "ComponentDescription": "A component that uses Docker Compose to run images from public Amazon ECR and Docker Hub."
    "ComponentPublisher": "Amazon",
    "ComponentDependencies": {
        "aws.greengrass.DockerApplicationManager": {
            "VersionRequirement": "~2.0.0"
        }
    },
    "Manifests": [
        {
            "Platform": {
                "os": "all"
            },
            "Lifecycle": {
                "Run": "docker-compose -f {artifacts:path}/docker-compose.yaml up"
            },
            "Artifacts": [
                {
                    "URI": "docker:public.ecr.aws/cloudwatch-agent/cloudwatch-agent:latest"
                },
                {
                    "URI": "docker:mysql:8.0"
                },
                {
                    "URI": "s3://DOC-EXAMPLE-BUCKET/folder/docker-compose.yaml"
                }
            ]
        }
    ]
}
```

YAML

```yaml
- ---
```
Run a Docker container from a private image in Amazon ECR

This section describes how you can create a custom component that runs a Docker container from a Docker image that is stored in a private repository in Amazon ECR.

To run a Docker container

1. Create a custom component (p. 324) on your AWS IoT Greengrass core device. Use the following example recipe, which has the following properties:
   - The Docker application manager component as a dependency. This component enables AWS IoT Greengrass to manage credentials to download images from private repositories.
   - The token exchange service component as a dependency. This component enables AWS IoT Greengrass to retrieve AWS credentials to interact with Amazon ECR.
   - A component artifact that specifies a Docker image in a private Amazon ECR repository.
   - A lifecycle run script that uses docker run to create and start a container from the image.

   ```
   RecipeFormatVersion: '2020-01-25'
   ComponentName: com.example.MyDockerComposeComponent
   ComponentVersion: '1.0.0'
   ComponentDescription: 'A component that uses Docker Compose to run images from public Amazon ECR and Docker Hub.'
   ComponentPublisher: Amazon
   ComponentDependencies:
     aws.greengrass.DockerApplicationManager:
       VersionRequirement: ~2.0.0
   Manifests:
     - Platform:
         os: all
       Lifecycle:
         Run: docker-compose -f {artifacts:path}/docker-compose.yaml up
       Artifacts:
         - URI: "docker:mysql:8.0"
         - URI: "s3://DOC-EXAMPLE-BUCKET/folder/docker-compose.yaml"
   ```

2. Test the component (p. 326) to verify that it works as expected.

3. When the component is ready, upload the component to AWS IoT Greengrass to deploy to other core devices. For more information, see Upload components to deploy to your core devices (p. 327).

4. When the component is ready, upload the component to AWS IoT Greengrass to deploy to other core devices. For more information, see Upload components to deploy to your core devices (p. 327).

Note

To use interprocess communication (IPC) in a Docker container component, you must set AWS IoT Greengrass Core environment variables in the Docker container. For more information, see Use interprocess communication in Docker container components (p. 339).

Important

You must install and start the Docker daemon before you deploy the component.

After you deploy the component locally, you can run the `docker container ls` command to verify that your container runs.

```
docker container ls
```

4. When the component is ready, upload the component to AWS IoT Greengrass to deploy to other core devices. For more information, see Upload components to deploy to your core devices (p. 327).
Run a Docker container from a private image in Amazon ECR

**JSON**

```json
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.MyPrivateDockerComponent",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that runs a Docker container from a private Amazon ECR image.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.DockerApplicationManager": {
      "VersionRequirement": "~2.0.0"
    },
    "aws.greengrass.TokenExchangeService": {
      "VersionRequirement": "~2.0.0"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "all"
      },
      "Lifecycle": {
        "Run": "docker run account-id.dkr.ecr.region.amazonaws.com/repository[:tag] @digest"
      },
      "Artifacts": [
        {
          "URI": "docker:account-id.dkr.ecr.region.amazonaws.com/repository[:tag] @digest"
        }
      ]
    }
  ]
}
```

**YAML**

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.MyPrivateDockerComponent
ComponentVersion: '1.0.0'
ComponentDescription: 'A component that runs a Docker container from a private Amazon ECR image.'
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.DockerApplicationManager:
    VersionRequirement: ~2.0.0
  aws.greengrass.TokenExchangeService:
    VersionRequirement: ~2.0.0
Manifests:
  - Platform:
      os: all
    Lifecycle:
      Run: docker run account-id.dkr.ecr.region.amazonaws.com/repository[:tag] @digest
    Artifacts:
      - URI: "docker:account-id.dkr.ecr.region.amazonaws.com/repository[:tag] @digest"
```
Note
To use interprocess communication (IPC) in a Docker container component, you must set AWS IoT Greengrass Core environment variables in the Docker container. For more information, see Use interprocess communication in Docker container components (p. 339).

2. Test the component (p. 326) to verify that it works as expected.

Important
You must install and start the Docker daemon before you deploy the component.

After you deploy the component locally, you can run the docker container ls command to verify that your container runs.

```
docker container ls
```

3. Upload the component to AWS IoT Greengrass to deploy to other core devices. For more information, see Upload components to deploy to your core devices (p. 327).

Run a Docker container from an image in Amazon S3

This section describes how you can run a Docker container in a component from a Docker image that is stored in Amazon S3.

To run a Docker container in a component from an image in Amazon S3

1. Run the docker save command to create a backup of a Docker container. You provide this backup as a component artifact to run the container on AWS IoT Greengrass. Replace `hello-world` with the name of the image, and replace `hello-world.tar` with the name of the archive file to create.

```
docker save hello-world > artifacts/com.example.MyDockerComponent/1.0.0/hello-world.tar
```

2. Create a custom component (p. 324) on your AWS IoT Greengrass core device. Use the following example recipe, which has the following properties:

- A lifecycle install script that uses `docker load` to load a Docker image from an archive.
- A lifecycle run script that uses `docker run` to create and start a container from the image. The `--rm` option cleans up the container when it exits.

```
{  
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.MyS3DockerComponent",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that runs a Docker container from an image in an S3 bucket.",
  "ComponentPublisher": "Amazon",
  "Manifests": [  
    {  
      "Platform": {  
        "os": "linux"
      },
      "Lifecycle": {  
        "Install": {  
          "Script": "docker load -i {artifacts:path}/hello-world.tar"
        }
      }
    }
  ]
}
```

337
Run a Docker container from an image in Amazon S3

```json
{
  "Run": {
    "Script": "docker run --rm hello-world"
  }
}
```

YAML

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.MyS3DockerComponent
ComponentVersion: '1.0.0'
ComponentDescription: 'A component that runs a Docker container from an image in an S3 bucket.'
ComponentPublisher: Amazon
Manifests:
  - Platform:
    os: linux
    Lifecycle:
      Install:
        Script: docker load -i {artifacts:path}/hello-world.tar
      Run:
        Script: docker run --rm hello-world
```

**Note**

To use interprocess communication (IPC) in a Docker container component, you must set AWS IoT Greengrass Core environment variables in the Docker container. For more information, see Use interprocess communication in Docker container components (p. 339).

You can also configure AWS IoT Greengrass to install Docker Engine when the component installs. For example, the following install script installs Docker Engine before it loads the Docker image. This install script works on Debian-based Linux distributions, such as Ubuntu. If you configure the component to install Docker Engine with this command, you may need to add `sudo` to the `docker` commands to run them.

```
sudo apt-get install docker-ce docker-ce-cli containerd.io && sudo docker load -i {artifacts:path}/hello-world.tar
```

3. **Test the component** (p. 326) to verify that it works as expected.

After you deploy the component locally, you can run the `docker container ls` command to verify that your container runs.

```
docker container ls
```

4. When the component is ready, upload the Docker image archive to an S3 bucket, and add its URI to the component recipe. Then, you can upload the component to AWS IoT Greengrass to deploy to other core devices. For more information, see Upload components to deploy to your core devices (p. 327).

When you’re done, the component recipe should look like the following example.

**JSON**

```json
{

```
Use interprocess communication in Docker container components

Interprocess communication (p. 396) enables you to develop components that can communicate with AWS IoT Greengrass Core and other components. To use interprocess communication in your Docker container components, you must set the following environment variables that AWS IoT Greengrass Core provides to components.

- AWS_REGION
Use interprocess communication in Docker container components

- `SVCUID`
- `AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT`
- `AWS_CONTAINER_AUTHORIZATION_TOKEN`
- `AWS_CONTAINER_CREDENTIALS_FULL_URI`

You can use the `-e`, `--env`, or `--env-file` parameter to set environment variables in the Docker container that you run.

When you start the Docker container, you must also mount as a volume any files that the Docker container component will need to access. To use interprocess communication, mount the root folder for your AWS IoT Greengrass Core software. You can use the `-v`, `--volume`, or `--mount` parameter to mount a volume in the Docker container that you run.

The following example component recipe has the following properties:

- The Docker application manager component as a dependency. This component enables AWS IoT Greengrass to manage credentials to download images from private repositories.
- The token exchange service component as a dependency. This component enables AWS IoT Greengrass to retrieve AWS credentials to interact with Amazon ECR.
- A component artifact that specifies a Docker image in a private Amazon ECR repository.
- A lifecycle run script that uses `docker run` to create and start a container from the image.
- The `--rm` option cleans up the container when it exits.

```
JSON
{
    "RecipeFormatVersion": "2020-01-25",
    "ComponentName": "com.example.MyIPCDockerComponent",
    "ComponentVersion": "1.0.0",
    "ComponentDescription": "A component that runs a Docker container and uses interprocess communication.",
    "ComponentPublisher": "Amazon",
    "ComponentDependencies": {
        "aws.greengrass.DockerApplicationManager": {
            "VersionRequirement": "~2.0.0"
        },
        "aws.greengrass.TokenExchangeService": {
            "VersionRequirement": "~2.0.0"
        }
    },
    "ComponentConfiguration": {
        "DefaultConfiguration": {
            "accessControl": {
                "aws.greengrass.ipc.pubsub": {
                    "com.example.MyDockerComponent:pubsub:1": {"policyDescription": "Allows access to publish and subscribe to all topics.", "operations": ["*"], "resources": ["*"]}
                }
            }
        }
    },
    "Manifests": [
        {
            "Platform": {
                "os": "all"
            },
            "Lifecycle": {
```
"Run": "docker run --rm -v /greengrass/v2:/greengrass/v2 -e AWS_REGION= #AWS_REGION -e SVCUID=$SVCUID -e AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT= #AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT -e AWS_CONTAINER_AUTHORIZATION_TOKEN=$AWS_CONTAINER_AUTHORIZATION_TOKEN -e AWS_CONTAINER_CREDENTIALS_FULL_URI=$AWS_CONTAINER_CREDENTIALS_FULL_URI account-id.dkr.ecr.region.amazonaws.com/repository[:tag:@digest]"
},
  "Artifacts": [
    {
      "URI": "docker:account-id.dkr.ecr.region.amazonaws.com/repository[:tag]@digest"
    }
  ]
}
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
Each recipe contains a list of *manifests*. Each manifest specifies a set of platform requirements and the lifecycle and artifacts to use for core devices whose platform meets those requirements. The core device uses the first manifest with platform requirements that the device meets. Specify a manifest without any platform requirements to match any core device.

You can also specify a global lifecycle that isn't in a manifest. In the global lifecycle, you can use *selection keys* that identify sub-sections of the lifecycle. Then, you can specify these selection keys within a manifest to use those sections of the global lifecycle in addition to the manifest's lifecycle. The core device uses the manifest's selection keys only if the manifest doesn't define a lifecycle. You can use the *all* selection in a manifest to match sections of the global lifecycle without selection keys.

After the AWS IoT Greengrass Core software selects a manifest that matches the core device, it does the following to identify the lifecycle steps to use:

- If the selected manifest defines a lifecycle, the core device uses that lifecycle.
- If the selected manifest doesn't define a lifecycle, the core device uses the global lifecycle. The core device does the following to identify which sections of the global lifecycle to use:
  - If the manifest defines selection keys, the core device uses the sections of the global lifecycle that have the manifest's selection keys.
  - If the manifest doesn't define selection keys, the core device uses the sections of the global lifecycle that don't have selection keys. This behavior is equivalent to a manifest that defines the *all* selection.

**Important**

A core device must match least one manifest's platform requirements to install the component. If no manifest matches the core device, then the AWS IoT Greengrass Core software doesn't install the component and the deployment fails.

You can define recipes in **JSON** or **YAML** format. The recipe examples section includes recipes in each format.

**Topics**

- Recipe format (p. 342)
- Recipe variables (p. 354)
- Recipe examples (p. 355)

### Recipe format

When you define a recipe for a component, you specify the following information in the recipe document. The same structure applies to recipes in YAML and JSON formats.

**RecipeFormatVersion**

The template version for the recipe. Choose the following option:

- 2020-01-25

**ComponentName**

The name of the component that this recipe defines. The component name must be unique in your AWS account in each Region.

**Tips**

- Use inverse domain name format to avoid name collision within your company. For example, if your company owns example.com and you work on a solar energy project, you can name your Hello World component com.example.solar.HelloWorld. This helps avoid component name collisions within your company.
• Avoid the `aws.greengrass` prefix in your component names. AWS IoT Greengrass uses this prefix for the public components (p. 132) that it provides. If you choose the same name as a public component, your component replaces that component. Then, AWS IoT Greengrass provides your component instead of the public component when it deploys components with a dependency on that public component. This feature enables you to override the behavior of public components, but it can also break other components if you don't intend to override a public component.

**ComponentVersion**

The version of the component.

**Note**

AWS IoT Greengrass uses semantic versions for components. Semantic versions follow a `major.minor.patch` number system. For example, version `1.0.0` represents the first major release for a component. For more information, see the semantic version specification.

**ComponentDescription**

(Optional) The description of the component.

**ComponentPublisher**

The publisher or author of the component.

**ComponentConfiguration**

(Optional) An object that defines the configuration or parameters for the component. You define the default configuration, and then when you deploy the component, you can specify the configuration object to provide to the component. Component configuration supports nested parameters and structures. This object contains the following information:

**DefaultConfiguration**

An object that defines the default configuration for the component. You define the structure of this object.

**Note**

AWS IoT Greengrass uses JSON for configuration values. JSON specifies a number type but doesn't differentiate between integers and floats. As a result, configuration values might convert to floats in AWS IoT Greengrass. To ensure that your component uses the correct data type, we recommend that you define numeric configuration values as strings. Then, have your component parse them as integers or floats. This ensures that your configuration values have the same type in the configuration and on your core device.

**ComponentDependencies**

(Optional) A dictionary of objects that each define a component dependency for the component. The key for each object identifies the name of the component dependency. AWS IoT Greengrass installs component dependencies when the component installs. AWS IoT Greengrass waits for dependencies to start before it starts the component. Each object contains the following information:

**VersionRequirement**

The npm-style semantic version constraint that defines the compatible component versions for this dependency. You can specify a version or a range of versions. For more information, see the npm semantic version calculator.

**DependencyType**

(Optional) The type of this dependency. Choose from the following options.

- **SOFT** – The component doesn't restart if the dependency changes state.
- **HARD** – The component restarts if the dependency changes state.
ComponentType

(Optional) The type of component.

Note
We don't recommend that you specify the component type in a recipe. AWS IoT Greengrass sets the type for you when you create a component.

The type can be one of the following types:

• `aws.greengrass.generic` – The component runs commands or provides artifacts.
• `aws.greengrass.lambda` – The component runs a Lambda function using the Lambda launcher component (p. 188). The ComponentSource parameter specifies the ARN of the Lambda function that this component runs.

We don't recommend that you use this option, because it's set by AWS IoT Greengrass when you create a component from a Lambda function. For more information, see Run AWS Lambda functions (p. 361).

• `aws.greengrass.plugin` – The component runs in the same Java Virtual Machine (JVM) as the Greengrass nucleus. If you deploy or restart a plugin component, the Greengrass nucleus restarts.

Plugin components use the same log file as the Greengrass nucleus. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

We don't recommend that you use this option in component recipes, because it's intended for AWS-provided components written in Java that directly interface with the Greengrass nucleus. For more information about which public components are plugins, see AWS-provided components (p. 132).

• `aws.greengrass.nucleus` – The nucleus component. For more information, see Greengrass nucleus (p. 136).

We don't recommend that you use this option in component recipes. It is intended for the Greengrass nucleus component, which provides the minimum functionality of the AWS IoT Greengrass Core software.

Defaults to `aws.greengrass.generic` when you create a component from a recipe, or `aws.greengrass.lambda` when you create a component from a Lambda function.

For more information, see Component types (p. 323).

ComponentSource

(Optional) The ARN of the Lambda function that a component runs.

We don't recommend that you specify the component source in a recipe. AWS IoT Greengrass sets this parameter for you when you create a component from a Lambda function. For more information, see Run AWS Lambda functions (p. 361).

Manifests

A list of objects that each define the component's lifecycle, parameters, and requirements for a platform. If a core device matches multiple manifests' platform requirements, AWS IoT Greengrass uses the first manifest that the core device matches. To ensure that core devices use the correct manifest, define the manifests with stricter platform requirements first. A manifest that applies to all platforms must be the last manifest in the list.

Important
A core device must match at least one manifest's platform requirements to install the component. If no manifest matches the core device, then the AWS IoT Greengrass Core software doesn't install the component and the deployment fails.
Each object contains the following information:

**Name**

(Optional) A friendly name for the platform that this manifest defines.

If you omit this parameter, AWS IoT Greengrass creates a name from the platform `os` and `architecture`.

**Platform**

(Optional) An object that defines the platform to which this manifest applies. Omit this parameter to define a manifest that applies to all platforms.

This object specifies key-value pairs about the platform on which a core device runs. When you deploy this component, the AWS IoT Greengrass Core software compares these key-value pairs with the platform attributes on the core device. The AWS IoT Greengrass Core software always defines `os` and `architecture`, and it might define additional attributes.

You can specify custom platform attributes for a core device when you deploy the Greengrass nucleus component. For more information, see the platform overrides parameter (p. 143) of the Greengrass nucleus component (p. 136).

For each key-value pair, you can specify one of the following values:

- An exact value, such as `linux`. Exact values must start with a letter or a number.
- `*`, which matches any value. This also matches when a value isn’t present.
- A Java-style regular expression, such as `/windows|linux/`. The regular expression must start and end with a slash character `/`. For example, the regular expression `/.+/` matches any non-blank value.

This object contains the following information:

**os**

(Optional) The name of the operating system for the platform that this manifest supports. Common platforms include the following values:

- `linux`
- `darwin` (macOS)

**architecture**

(Optional) The processor architecture for the platform that this manifest supports. Common architectures include the following values:

- `amd64`
- `arm`
- `aarch64`
- `x86`

**key**

(Optional) A platform attribute that you define for this manifest. Replace `key` with the name of the platform attribute. The AWS IoT Greengrass Core software matches this platform attribute with the key-value pairs that you specify in the Greengrass nucleus component configuration. For more information, see the platform overrides parameter (p. 143) of the Greengrass nucleus component (p. 136).

**Tip**

Use inverse domain name format to avoid name collision within your company. For example, if your company owns `example.com` and you work on a radio project, you can name a custom platform attribute `com.example.radio.RadioModule`. This helps avoid platform attribute name collisions within your company.
For example, you might define a platform attribute, com.example.radio.RadioModule, to specify a different manifest based on which radio module is available on a core device. Each manifest can include different artifacts that apply to different hardware configurations, so that you deploy the minimal set of software to the core device.

**Lifecycle**

An object that defines how to install and run the component on the platform that this manifest defines. You can also define a global lifecycle (p. 352) that applies to all platforms. The core device uses the global lifecycle only if the manifest to use doesn't specify a lifecycle.

**Note**

You define this lifecycle within a manifest. The lifecycle steps that you specify here apply to only the platform that this manifest defines. You can also define a global lifecycle (p. 352) that applies to all platforms.

This object contains the following information:

**Setenv**

(Optional) A dictionary of environment variables to provide to all lifecycle scripts. You can override these environment variables with `Setenv` in each lifecycle script.

**Bootstrap**

(Optional) An object that defines the script to run when the AWS IoT Greengrass Core software deploys the component. This lifecycle step runs before the install lifecycle step (p. 347) in the following cases:

- The component deploys to the core device for the first time.
- The component version changes.
- The bootstrap script changes as the result of a component configuration update.

You can use this lifecycle step to restart the AWS IoT Greengrass Core software. This lets you develop a component that performs a restart after it installs operating system updates or runtime updates, for example.

After the AWS IoT Greengrass Core software completes the bootstrap step for all components that have a bootstrap step in a deployment, the software restarts.

**Important**

You must configure the AWS IoT Greengrass Core software as a system service to restart the AWS IoT Greengrass Core software. If you don't configure the AWS IoT Greengrass Core software as a system service, the software won't restart. For more information, see Configure AWS IoT Greengrass as a system service (p. 119).

This object contains the following information:

**Script**

The script to run. The exit code of this script defines the restart instruction. Use the following exit codes:

- 0 – Don't restart the AWS IoT Greengrass Core software or the core device. The AWS IoT Greengrass Core software still restarts after all components bootstrap.
- 100 – Request to restart the AWS IoT Greengrass Core software.

Exit codes 100 to 199 are reserved for special behavior. Other exit codes represent script errors.

**RequiresPrivilege**

(Optional) You can run the script with root privileges. If you set this option to `true`, then the AWS IoT Greengrass Core software runs this lifecycle script as root instead of as the system user that you configure to run this component. Defaults to `false`.
Recipe format

**Timeout**

(Optional) The maximum amount of time in seconds that the script can run before the AWS IoT Greengrass Core software terminates the process.

Default: 120 seconds

**Setenv**

(Optional) The dictionary of environment variables to provide to the script. These environment variables override the variables that you provide in Lifecycle.Setenv.

**Install**

(Optional) An object that defines the script to run when the component installs. The AWS IoT Greengrass Core software also runs this lifecycle step when each time the software launches.

If the Install script exits with a success code, the component enters the INSTALLED state.

This object contains the following information:

**Script**

The script to run.

**RequiresPrivilege**

(Optional) You can run the script with root privileges. If you set this option to true, then the AWS IoT Greengrass Core software runs this lifecycle script as root instead of as the system user that you configure to run this component. Defaults to false.

**Skipif**

(Optional) The check to determine whether or not to run the script. You can define to check if an executable is on the path or if a file exists. If the output is true, then the AWS IoT Greengrass Core software skips the step. Choose from the following checks:

- **onpath** **runnable** – Check if a runnable is on the system path. For example, use `onpath python3` to skip this lifecycle step if Python 3 is available.
- **exists** **file** – Check if a file exists. For example, use `exists /tmp/my-configuration.db` to skip this lifecycle step if `/tmp/my-configuration.db` is present.

**Timeout**

(Optional) The maximum amount of time in seconds that the script can run before the AWS IoT Greengrass Core software terminates the process.

Default: 120 seconds

**Setenv**

(Optional) The dictionary of environment variables to provide to the script. These environment variables override the variables that you provide in Lifecycle.Setenv.

**Run**

(Optional) An object that defines the script to run when the component starts.

The component enters the RUNNING state when this lifecycle step runs. If the Run script exits with a success code, the component enters the FINISHED state.

Components that depend on this component start when this lifecycle step runs. To run a background process, such as a service that dependent components use, use the Startup lifecycle step instead.
Recipe format

**Note**
You can define only one Startup or Run lifecycle.

This object contains the following information:

**Script**

The script to run.

**RequiresPrivilege**

(Optional) You can run the script with root privileges. If you set this option to `true`, then the AWS IoT Greengrass Core software runs this lifecycle script as root instead of as the system user that you configure to run this component. Defaults to `false`.

**Skipif**

(Optional) The check to determine whether or not to run the script. You can define to check if an executable is on the path or if a file exists. If the output is true, then the AWS IoT Greengrass Core software skips the step. Choose from the following checks:

- `onpath` `runnable` — Check if a runnable is on the system path. For example, use `onpath python3` to skip this lifecycle step if Python 3 is available.
- `exists` `file` — Check if a file exists. For example, use `exists /tmp/my-configuration.db` to skip this lifecycle step if `/tmp/my-configuration.db` is present.

**Timeout**

(Optional) The maximum amount of time in seconds that the script can run before the AWS IoT Greengrass Core software terminates the process.

This lifecycle step doesn't timeout by default. If you omit this timeout, the Run script runs until it exits.

**Setenv**

(Optional) The dictionary of environment variables to provide to the script. These environment variables override the variables that you provide in Lifecycle.Setenv.

**Startup**

(Optional) An object that defines the background process to run when the component starts.

Use Startup to run a command that must exit successfully before dependent components can start. For example, you might define a Startup step that starts the MySQL process with `/etc/init.d/mysqld start`.

The component enters the STARTING state when this lifecycle step runs. If the Startup script exits with a success code, the component enters the RUNNING state. Then, dependent components can start.

**Note**
You can define only one Startup or Run lifecycle.

This object contains the following information:

**Script**

The script to run.

**RequiresPrivilege**

(Optional) You can run the script with root privileges. If you set this option to `true`, then the AWS IoT Greengrass Core software runs this lifecycle script as root instead of as the system user that you configure to run this component. Defaults to `false`.
Recipe format

Skipif

(Optional) The check to determine whether or not to run the script. You can define to check if an executable is on the path or if a file exists. If the output is true, then the AWS IoT Greengrass Core software skips the step. Choose from the following checks:

- onpath runnable – Check if a runnable is on the system path. For example, use `onpath python3` to skip this lifecycle step if Python 3 is available.
- exists file – Check if a file exists. For example, use `exists /tmp/my-configuration.db` to skip this lifecycle step if `/tmp/my-configuration.db` is present.

Timeout

(Optional) The maximum amount of time in seconds that the script can run before the AWS IoT Greengrass Core software terminates the process.

Default: 120 seconds

Setenv

(Optional) The dictionary of environment variables to provide to the script. These environment variables override the variables that you provide in Lifecycle.Setenv.

Shutdown

(Optional) An object that defines the script to run when the component shuts down.

If you start a background process in Startup, use the Shutdown step to stop that process when the component shuts down. For example, you might define a Shutdown step that stops the MySQL process with `/etc/init.d/mysqld stop`.

The component enters the STOPPING state when this lifecycle step runs.

This object contains the following information:

Script

The script to run.

RequiresPrivilege

(Optional) You can run the script with root privileges. If you set this option to true, then the AWS IoT Greengrass Core software runs this lifecycle script as root instead of as the system user that you configure to run this component. Defaults to false.

Skipif

(Optional) The check to determine whether or not to run the script. You can define to check if an executable is on the path or if a file exists. If the output is true, then the AWS IoT Greengrass Core software skips the step. Choose from the following checks:

- onpath runnable – Check if a runnable is on the system path. For example, use `onpath python3` to skip this lifecycle step if Python 3 is available.
- exists file – Check if a file exists. For example, use `exists /tmp/my-configuration.db` to skip this lifecycle step if `/tmp/my-configuration.db` is present.

Timeout

(Optional) The maximum amount of time in seconds that the script can run before the AWS IoT Greengrass Core software terminates the process.

Default: 15 seconds.
Setenv

(Optional) The dictionary of environment variables to provide to the script. These environment variables override the variables that you provide in Lifecycle.Setenv.

Recover

(Optional) An object that defines the script to run when the component encounters an error.

This step runs when a component enters the **ERRORED** state. If the component becomes **ERRORED** three times without successfully recovering, the component changes to the **BROKEN** state. To fix a **BROKEN** component, you must deploy it again.

This object contains the following information:

**Script**

The script to run.

**RequiresPrivilege**

(Optional) You can run the script with root privileges. If you set this option to `true`, then the AWS IoT Greengrass Core software runs this lifecycle script as root instead of as the system user that you configure to run this component. Defaults to `false`.

**Skipif**

(Optional) The check to determine whether or not to run the script. You can define to check if an executable is on the path or if a file exists. If the output is true, then the AWS IoT Greengrass Core software skips the step. Choose from the following checks:

- **onpath runnable** – Check if a runnable is on the system path. For example, use `onpath python3` to skip this lifecycle step if Python 3 is available.
- **exists file** – Check if a file exists. For example, use `exists /tmp/my-configuration.db` to skip this lifecycle step if `/tmp/my-configuration.db` is present.

**Timeout**

(Optional) The maximum amount of time in seconds that the script can run before the AWS IoT Greengrass Core software terminates the process.

Default: 60 seconds.

Setenv

(Optional) The dictionary of environment variables to provide to the script. These environment variables override the variables that you provide in Lifecycle.Setenv.

Selections

(Optional) A list of selection keys that specify sections of the global lifecycle (p. 352) to run for this manifest. In the global lifecycle, you can define lifecycle steps with selection keys at any level to select sub-sections of the lifecycle. Then, the core device uses those sections that match the selection keys in this manifest. For more information, see the global lifecycle examples (p. 352).

**Important**

The core device uses the selections from the global lifecycle only if this manifest doesn't define a lifecycle.

You can specify the all selection key to run sections of the global lifecycle that don't have selection keys.
Artifacts

(Optional) A list of objects that each define a binary artifact for the component on the platform that this manifest defines. For example, you can define code or images as artifacts.

When the component deploys, the AWS IoT Greengrass Core software downloads the artifact to a folder on the core device. You can also define artifacts as archive files that the software extracts after it downloads them.

You can use recipe variables (p. 354) to get the paths to the folders where the artifacts install on the core device.

- Normal files – Use the artifacts:path recipe variable (p. 354) to get the path to the folder that contains the artifacts. For example, specify `{artifacts:path}/my_script.py` in a recipe to get the path to an artifact that has the URI `s3://DOC-EXAMPLE-BUCKET/path/to/my_script.py`.
- Extracted archives – Use the artifacts:decompressedPath recipe variable (p. 355) to get the path to the folder that contains the extracted archive artifacts. The AWS IoT Greengrass Core software extracts each archive to a folder with the same name as the archive. For example, specify `{artifacts:decompressedPath}/my_archive/my_script.py` in a recipe to get the path to `my_script.py` in the archive artifact that has the URI `s3://DOC-EXAMPLE-BUCKET/path/to/my_archive.zip`.

**Note**
When you develop a component with an archive artifact on a local core device, you might not have a URI for that artifact. To test your component with a Unarchive option that extracts the artifact, specify a URI where the file name matches the name of your archive artifact file. You can specify the URI where you expect to upload the archive artifact, or you can specify a new placeholder URI. For example, to extract the `my_archive.zip` artifact during a local deployment, you can specify `s3://DOC-EXAMPLE-BUCKET/my_archive.zip`.

Each object contains the following information:

**URI**

The URI of an artifact in an S3 bucket. The AWS IoT Greengrass Core software fetches the artifact from this URI when the component installs, unless the artifact already exists on the device. Each artifact must have a unique file name within each manifest.

**Unarchive**

(Optional) The type of archive to unpack. Choose from the following options:

- NONE – The file isn't an archive to unpack. The AWS IoT Greengrass Core software installs the artifact to a folder on the core device. You can use the artifacts:path recipe variable (p. 354) to get the path to this folder.
- ZIP – The file is a ZIP archive. The AWS IoT Greengrass Core software extracts the archive to a folder with the same name as the archive. You can use the artifacts:decompressedPath recipe variable (p. 355) to get the path to the folder that contains this folder.

Defaults to NONE.

**Permission**

(Optional) An object that defines the access permissions to set for this artifact file. You can set the read permission and the execute permission.

**Note**
You can't set the write permission, because the AWS IoT Greengrass Core software doesn't allow components to edit artifact files in the artifacts folder. To edit an
artifact file in a component, copy it to another location or publish and deploy a new artifact file.

If you define an artifact as an archive to unpack, then the AWS IoT Greengrass Core software sets these access permissions on the files that it unpacks from the archive. The AWS IoT Greengrass Core software sets the folder's access permissions to ALL for Read and Execute. This allows components to view the unpacked files in the folder. To set permissions on individual files from the archive, you can set the permissions in the install lifecycle script (p. 347).

This object contains the following information:

Read

(Optional) The read permission to set for this artifact file. To allow other components to access this artifact, such as components that depend on this component, specify ALL. Choose from the following options:

• NONE – The file isn’t readable.
• OWNER – The file is readable by the system user that you configure to run this component.
• ALL – The file is readable by all users.

Defaults to OWNER.

Execute

(Optional) The run permission to set for this artifact file. The Execute permission implies the Read permission. For example, if you specify ALL for Execute, then all users can read and run this artifact file.

Choose from the following options:

• NONE – The file isn’t runnable.
• OWNER – The file is runnable by the system user that you configure to run the component.
• ALL – The file is runnable by all users.

Defaults to NONE.

Digest

(Read-only) The cryptographic digest hash of the artifact. When you create a component, AWS IoT Greengrass uses a hash algorithm to calculate a hash of the artifact file. Then, when you deploy the component, the Greengrass nucleus calculates the hash of the downloaded artifact and compares the hash with this digest to verify the artifact before installation. If the hash doesn’t match the digest, the deployment fails.

If you set this parameter, AWS IoT Greengrass replaces the value that you set when you create the component.

Algorithm

(Read-only) The hash algorithm that AWS IoT Greengrass uses to calculate the digest hash of the artifact.

If you set this parameter, AWS IoT Greengrass replaces the value that you set when you create the component.

Lifecycle

An object that defines how to install and run the component. The core device uses the global lifecycle only if the manifest (p. 344) to use doesn’t specify a lifecycle.
**Note**
You define this lifecycle outside a manifest. You can also define a [manifest lifecycle](p. 346) that applies to the platforms that match that manifest.

In the global lifecycle, you can specify lifecycles that run for certain selection keys (p. 350) that you specify in each manifest. Selection keys are strings that identify sections of the global lifecycle to run for each manifest.

The `all` selection key is the default on any section without a selection key. This means that you can specify the `all` selection key in a manifest to run the sections of the global lifecycle without selection keys. You don't need to specify the `all` selection key in the global lifecycle.

If a manifest doesn't define a lifecycle or selection keys, the core device defaults to use the `all` selection. This means that in this case, the core device uses the sections of the global lifecycle that don't use selection keys.

This object contains the same information as the [manifest lifecycle](p. 346), but you can specify selection keys at any level to select sub-sections of the lifecycle.

**Tip**
We recommend that you use only lowercase letters for each selection key to avoid conflicts between selection keys and lifecycle keys. Lifecycle keys start with a capital letter.

### Example Example global lifecycle with top-level selection keys

```json
Lifecycle:
  key1:
    Install:
      Skipif: onpath executable | exists file
      Script: command1
  key2:
    Install:
      Script: command2
  all:
    Install:
      Script: command3
```

### Example Example global lifecycle with bottom-level selection keys

```json
Lifecycle:
  Install:
    Script:
      key1: command1
      key2: command2
      all: command3
```

### Example Example global lifecycle with multiple levels of selection keys

```json
Lifecycle:
  key1:
    Install:
      Skipif: onpath executable | exists file
      Script: command1
  key2:
    Install:
      Script: command2
  all:
    Install:
      Script:
        key3: command3
        key4: command4
```
Recipe variables

Recipe variables expose information from the current component and nucleus for you to use in your recipes. You can use recipe variables within lifecycle definitions in component recipes. For example, use a recipe variable to pass component configuration parameters to an application that you run in a lifecycle script.

Recipe variables use {recipe_variable} syntax. The curly braces indicate a recipe variable.

AWS IoT Greengrass supports the following recipe variables:

**component_dependency_name**:configuration:json_pointer

The value of a configuration parameter for the component that this recipe defines or for a component on which this component depends.

You can use this variable to provide a parameter to a script that you run in the component lifecycle.

This recipe variable has the following inputs:

- **component_dependency_name** – (Optional) The name of the component dependency to query. Omit this segment to query the component that this recipe defines. You can specify only direct dependencies.

- **json_pointer** – The JSON pointer to the configuration value to evaluate. JSON pointers start with a forward slash /. To identify a value in a nested component configuration, use forward slashes (/) to separate the keys for each level in the configuration. You can use a number as a key to specify an index in a list. For more information, see the JSON pointer specification.

AWS IoT Greengrass Core uses JSON pointers for recipes in YAML format.

The JSON pointer can reference the following node types:

- A value node. AWS IoT Greengrass Core replaces the recipe variable with the string representation of the value. Null values convert to null as a string.

- An object node. AWS IoT Greengrass Core replaces the recipe variable with the serialized JSON string representation of that object.

- No node. AWS IoT Greengrass Core doesn't replace the recipe variable.

For example, the {configuration:/Message} recipe variable retrieves the value of the Message key in the component configuration. The {com.example.MyComponentDependency:configuration:/server/port} recipe variable retrieves the value of port in the server configuration object of a component dependency.

**component_dependency_name**:artifacts:path

The root path of the artifacts for the component that this recipe defines or for a component on which this component depends.

When a component installs, AWS IoT Greengrass copies the component's artifacts to the folder that this variable exposes. You can use this variable to identify the location of a script to run in the component lifecycle, for example.

The folder at this path is read-only. To modify artifact files, copy the files to another location, such as the current working directory ($PWD or .). Then, modify the files there.

To read or run an artifact from a component dependency, that artifact's Read or Execute permission must be ALL. For more information, see the artifact permissions (p. 351) that you define in the component recipe.
This recipe variable has the following inputs:
- `component_dependency_name` – (Optional) The name of the component dependency to query. Omit this segment to query the component that this recipe defines. You can specify only direct dependencies.

```component_dependency_name:artifacts:decompressedPath```

The root path of the decompressed archive artifacts for the component that this recipe defines or for a component on which this component depends.

When a component installs, AWS IoT Greengrass unpacks the component’s archive artifacts to the folder that this variable exposes. You can use this variable to identify the location of a script to run in the component lifecycle, for example.

Each artifact unzips to a folder within the decompressed path, where the folder has the same name as the artifact minus its extension. For example, a ZIP artifact named `models.zip` unpacks to the `{artifacts:decompressedPath}/models` folder.

The folder at this path is read-only. To modify artifact files, copy the files to another location, such as the current working directory (`$PWD` or `.`). Then, modify the files there.

To read or run an artifact from a component dependency, that artifact’s `Read` or `Execute` permission must be `ALL`. For more information, see the artifact permissions (p. 351) that you define in the component recipe.

This recipe variable has the following inputs:
- `component_dependency_name` – (Optional) The name of the component dependency to query. Omit this segment to query the component that this recipe defines. You can specify only direct dependencies.

```component_dependency_name:work:path```

This feature is available for v2.0.4 and later of the Greengrass nucleus component (p. 136).

The work path for the component that this recipe defines or for a component on which this component depends. The value of this recipe variable is equivalent to the output of the `$PWD` environment variable and the `pwd` command when run from the context of the component.

You can use this recipe variable to share files between a component and a dependency.

The folder at this path is readable and writable by the component that this recipe defines and by other components that run as the same user and group.

This recipe variable has the following inputs:
- `component_dependency_name` – (Optional) The name of the component dependency to query. Omit this segment to query the component that this recipe defines. You can specify only direct dependencies.

```kernel:rootPath```

The AWS IoT Greengrass Core root path.

```iot:thingName```

This feature is available for v2.3.0 and later of the Greengrass nucleus component (p. 136).

The name of the core device’s AWS IoT thing.

### Recipe examples

You can reference the following recipe examples to help you create recipes for your components.
Hello World component recipe

The following recipe describes a Hello World component that runs a Python script. This component supports Linux and accepts a `Message` parameter that AWS IoT Greengrass passes as an argument to the Python script. This is the recipe for the Hello World component in the Getting started tutorial (p. 24).

**JSON**

```json
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.HelloWorld",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "My first AWS IoT Greengrass component.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "world"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "Run": "python3 -u {artifacts:path}/hello_world.py '{configuration:/Message}'"
      }
    }
  ]
}
```

**YAML**

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: My first AWS IoT Greengrass component.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    Message: world
Manifests:
- Platform:
  os: linux
Lifecycle:
  Run: |
    python3 -u {artifacts:path}/hello_world.py '{configuration:/Message}'
```

Python runtime component example

The following recipe describes a component that installs Python. This component supports 64-bit Linux devices.
Component recipe that specifies several fields

The following component recipe uses several recipe fields.

JSON

```json
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.FooService",
  "ComponentDescription": "Complete recipe for AWS IoT Greengrass components",
  "ComponentPublisher": "Amazon",
  "ComponentVersion": "1.0.0",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "TestParam": "TestValue"
    }
  },
  "ComponentDependencies": {
    "BarService": {
      "VersionRequirement": "^1.1.0",
      "DependencyType": "SOFT"
    },
    "BazService": {
      "VersionRequirement": "^2.0.0"
    }
  }
}
```
YAML

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.FooService
ComponentDescription: Complete recipe for AWS IoT Greengrass components
ComponentPublisher: Amazon
ComponentVersion: '1.0.0'
ComponentConfiguration:
  DefaultConfiguration:
    TestParam: TestValue
ComponentDependencies:
  BarService:
    VersionRequirement: ^1.1.0
    DependencyType: SOFT
  BazService: VersionRequirement: ^2.0.0
Manifests:
  - Platform:
    os: linux
    architecture: amd64
  Lifecycle:
    Install:
      Skipif: onpath git
      Script: sudo apt-get install git
Artifacts:
```
Component environment variable reference

The AWS IoT Greengrass Core software sets environment variables when it runs lifecycle scripts for components. You can get these environment variables in your components to get the thing name, AWS Region, and Greengrass nucleus version. The software also sets environment variables that your component requires to use the interprocess communication SDK (p. 396) and to interact with AWS services (p. 329).

You can also set custom environment variables for your component's lifecycle scripts. For more information, see Setenv (p. 346).

The AWS IoT Greengrass Core software sets the following environment variables:

**AWS_IOT_THING_NAME**

The name of the AWS IoT thing that represents this Greengrass core device.

**AWS_REGION**

The AWS Region where this Greengrass core device operates.

The AWS SDKs use this environment variable to identify the default Region to use. This variable is equivalent to `AWS_DEFAULT_REGION`.

**AWS_DEFAULT_REGION**

The AWS Region where this Greengrass core device operates.

The AWS CLI uses this environment variable to identify the default Region to use. This variable is equivalent to `AWS_REGION`.

**GGC_VERSION**

The version of the Greengrass nucleus component (p. 136) that runs on this Greengrass core device.

**AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT**

The path to the IPC socket that components use to communicate with the AWS IoT Greengrass Core software. For more information, see Use the AWS IoT Device SDK for interprocess communication (IPC) (p. 396).

**SVCUID**

The secret token that components use to connect to the IPC socket and communicate with the AWS IoT Greengrass Core software. For more information, see Use the AWS IoT Device SDK for interprocess communication (IPC) (p. 396).

**AWS_CONTAINER_AUTHORIZATION_TOKEN**

The secret token that components use to retrieve credentials from the token exchange service component (p. 311).
AWS CONTAINER_CREDENTIALS_FULL_URI

The URI that components request to retrieve credentials from the token exchange service component (p. 311).
Run AWS Lambda functions

You can import AWS Lambda functions as components that run on AWS IoT Greengrass core devices. You might want to do this in the following cases:

• You have application code in Lambda functions that you want to deploy to core devices.
• You have AWS IoT Greengrass V1 applications that you want to run on AWS IoT Greengrass V2 core devices. For more information, see Run AWS IoT Greengrass V1 applications on AWS IoT Greengrass V2 (p. 7).

Lambda functions include dependencies on the following components. You don’t need to define these components as dependencies when you import the function. When you deploy the Lambda function component, the deployment includes these Lambda component dependencies.

• The Lambda launcher component (p. 188) (aws.greengrass.LambdaLauncher) handles processes and environment configuration.
• The Lambda manager component (p. 190) (aws.greengrass.LambdaManager) handles interprocess communication and scaling.
• The Lambda runtimes component (p. 193) (aws.greengrass.LambdaRuntimes) provides artifacts for each supported Lambda runtime.

Topics
• Requirements (p. 361)
• Configure Lambda function lifecycle (p. 362)
• Configure Lambda function containerization (p. 362)
• Import a Lambda function as a component (console) (p. 364)
• Import a Lambda function as a component (AWS CLI) (p. 368)

Requirements

Your core devices and Lambda functions must meet the following requirements for you to run the functions on the AWS IoT Greengrass Core software:

• Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Requirements to run Lambda functions (p. 43).
• You must install the programming languages that the Lambda function uses on your core devices.
  Tip
  You can create a component that installs the programming language, and then specify that component as a dependency of your Lambda function component.
• Your Lambda function must use one of the following runtimes:
  • Python 3.8 – python3.8
  • Python 3.7 – python3.7
  • Python 2.7 – python2.7 *
  • Java 8 – java8

361
Configure Lambda function lifecycle

The Greengrass Lambda function lifecycle determines when a function starts and how it creates and uses containers. The lifecycle also determines how the AWS IoT Greengrass Core software retains variables and preprocessing logic that are outside of the function handler.

AWS IoT Greengrass supports on-demand (default) and long-lived lifecycles:

• **On-demand** functions start when they are invoked and stop when there are no tasks left to run. Each invocation of the function creates a separate container, also called a sandbox, to process invocations, unless an existing container is available for reuse. Any of the containers might process data that you send to the function.

  Multiple invocations of an on-demand function can run simultaneously.

  Variables and preprocessing logic that you define outside of the function handler are not retained when new containers are created.

• **Long-lived** (or pinned) functions start when the AWS IoT Greengrass Core software starts and run in a single container. The same container processes all data that you send to the function.

  Multiple invocations are queued until the AWS IoT Greengrass Core software runs earlier invocations.

  Variables and preprocessing logic that you define outside of the function handler are retained for every invocation of the handler.

  Use long-lived Lambda functions when you need to start doing work without any initial input. For example, a long-lived function can load and start processing a machine learning model to be ready when the function receives device data.

  **Note**

  Long-lived functions have timeouts that are associated with each invocation of their handler. If you want to invoke code that runs indefinitely, you must start it outside of the handler. Make sure that there’s no blocking code outside of the handler that might prevent the function from initializing.

  These functions run unless the AWS IoT Greengrass Core software stops, such as during a deployment or reboot. These functions won't run if the function encounters an uncaught exception, exceeds its memory limits, or enters an error state, such as a handler timeout.

  For more information about container reuse, see Understanding Container Reuse in AWS Lambda in the AWS Compute Blog.

Configure Lambda function containerization

By default, Lambda functions run inside of an AWS IoT Greengrass container. Greengrass containers provide isolation between your functions and the host. This isolation increases security for both the host and the functions in the container.
We recommend that you run Lambda functions in a Greengrass container, unless your use case requires them to run without containerization. By running your Lambda functions in a Greengrass container, you have more control over how you restrict access to resources.

You might run a Lambda function without containerization in the following cases:

- You want to run AWS IoT Greengrass on a device that doesn't support container mode. An example would be if you wanted to use a special Linux distribution, or have an earlier kernel version that is out of date.
- You want to run your Lambda function in another container environment with its own OverlayFS, but encounter OverlayFS conflicts when you run in a Greengrass container.
- You need access to local resources with paths that can't be determined at deployment time, or whose paths can change after deployment. An example of this resource would be a pluggable device.
- You have an earlier application that was written as a process, and you encounter issues when you run it in a Greengrass container.

### Containerization differences

<table>
<thead>
<tr>
<th>Containerization</th>
<th>Notes</th>
</tr>
</thead>
</table>
| Greengrass container | • All AWS IoT Greengrass features are available when you run a Lambda function in a Greengrass container.  
  • Lambda functions that run in a Greengrass container don't have access to the deployed code of other Lambda functions, even if they run with the same system group. In other words, your Lambda functions run with increased isolation from one another.  
  • Because the AWS IoT Greengrass Core software runs all child processes in the same container as the Lambda function, the child processes stop when the Lambda function stops. |
| No container       | • The following features aren't available to non-containerized Lambda functions:  
  • Lambda function memory limits.  
  • Local device and volume resources. You must access these resources using their file paths on the core device instead of as Lambda function resources.  
  • If your non-containerized Lambda function accesses a machine learning resource, you must identify a resource owner and set access permissions on the resource, not on the Lambda function.  
  • Non-containerized Lambda functions have read-only access to the deployed code of other Lambda functions that run with the same system group. |

If you change the containerization for a Lambda function when you deploy it, the function might not work as expected. If the Lambda function uses local resources that are no longer available with the new containerization setting, deployment fails.
Import a Lambda function as a component (console)

When you use the AWS IoT Greengrass console to create a Lambda function component, you import an existing AWS Lambda function and then configure it to create a component that runs on your Greengrass device.

Before you begin, review the requirements to run Lambda functions on Greengrass devices.

Tasks

- Step 1: Choose a Lambda function to import (p. 364)
- Step 2: Configure Lambda function parameters (p. 365)
- Step 3: (Optional) Specify supported platforms for the Lambda function (p. 366)
- Step 4: (Optional) Specify component dependencies for the Lambda function (p. 366)
- Step 5: (Optional) Run the Lambda function in a container (p. 367)
- Step 6: Create the Lambda function component (p. 368)

Step 1: Choose a Lambda function to import

1. In the AWS IoT Greengrass console navigation menu, choose Components.
2. On the Components page, choose Create component.
3. On the Create component page, under Component information, choose Import Lambda function.
4. In Lambda function, search for and choose the Lambda function that you want to import.

   AWS IoT Greengrass creates the component with the name of the Lambda function.
5. In Lambda function version, choose the version to import. You can’t choose Lambda aliases like $LATEST.
AWS IoT Greengrass creates the component with the version of the Lambda function as a valid semantic version. For example, if your function version is 3, the component version becomes 3.0.0.

Step 2: Configure Lambda function parameters

On the Create component page, under Lambda function configuration, configure the following parameters to use to run the Lambda function.

1. (Optional) Add the list of event sources to which the Lambda function subscribes for work messages. The Lambda function is called when it receives a message from an event source. You can subscribe this function to local publish/subscribe messages and AWS IoT Core MQTT messages.

   Under Event sources, do the following to add an event source:
   a. For each event source that you add, specify the following options:
      • Topic – The name of the topic to subscribe for messages.
      • Type – The type of event source.
   b. To add another event source, choose Add event source and repeat the previous step. To remove an event source, choose Remove next to the event source that you want to remove.

2. For Timeout (seconds), enter the maximum amount of time in seconds that a non-pinned Lambda function can run before it times out. The default is 3 seconds.

3. For Pinned, choose whether the Lambda function component is pinned. The default is True.
   • A pinned (or long-lived) Lambda function starts when AWS IoT Greengrass starts and keeps running in its own container.
   • A non-pinned (or on-demand) Lambda function starts only when it receives a work item and exits after it remains idle for a specified maximum idle time. If the function has multiple work items, the AWS IoT Greengrass Core software creates multiple instances of the function.

4. (Optional) Under Additional parameters, set the following Lambda function parameters.
   • Status timeout (seconds) – The interval in seconds at which the Lambda function component sends status updates to the Lambda manager component. This parameter applies only to pinned functions. The default is 60 seconds.
   • Maximum queue size – The maximum size of the message queue for the Lambda function component. The AWS IoT Greengrass Core software stores messages in a FIFO (first-in, first-out) queue until it can run the Lambda function to consume each message. The default is 1,000 messages.
   • Maximum number of instances – The maximum number of instances that a non-pinned Lambda function can run at the same time. The default is 100 instances.
   • Maximum idle time (seconds) – The maximum amount of time in seconds that a non-pinned Lambda function can idle before the AWS IoT Greengrass Core software stops its process. The default is 60 seconds.
   • Encoding type – The type of payload that the Lambda function supports. Choose from the following options:
     • JSON
     • Binary
     The default is JSON.

5. (Optional) Specify the list of command line arguments to pass to the Lambda function when it runs.
   b. For each argument that you add, enter the argument that you want to pass to the function.
Step 3: (Optional) Specify supported platforms for the Lambda function

All core devices have attributes for operating system and architecture. When you deploy the Lambda function component, the AWS IoT Greengrass Core software compares the platform values that you specify with the platform attributes on the core device to determine whether the Lambda function is supported on that device.

**Note**
You can also specify custom platform attributes when you deploy the Greengrass nucleus component to a core device. For more information, see the platform overrides parameter (p. 143) of the Greengrass nucleus component (p. 136).

Under **Lambda function configuration, Additional parameters, Platforms**, do the following to specify the platforms that this Lambda function supports.

1. For each platform, specify the following options:
   - **Operating system** – The name of the operating system for the platform. Currently, the only supported value is `linux`.
   - **Architecture** – The processor architecture for the platform. Supported values are:
     - `amd64`
     - `arm`
     - `aarch64`
     - `x86`

2. To add another platform, choose **Add platform** and repeat the previous step. To remove a supported platform, choose **Remove** next to the platform that you want to remove.

Step 4: (Optional) Specify component dependencies for the Lambda function

Component dependencies identify additional AWS-provided components or custom components that your function uses. When you deploy the Lambda function component, the deployment includes these dependencies for your function to run.

**Important**
To import a Lambda function that you created to run on AWS IoT Greengrass V1, you must define individual component dependencies for the features that your function uses, such as subscriptions, secrets, and stream manager. Define these components as **hard**
Step 5: (Optional) Run the Lambda function in a container

By default, Lambda functions run in an isolated runtime environment inside the AWS IoT Greengrass Core software. You can also choose to run the Lambda function as a process without any isolation (that is, in No container mode).

Under Linux process configuration, for Isolation mode, choose from the following options to select the containerization for your Lambda function:

- **Greengrass container** – The Lambda function runs in a container. This is the default selection.
- **No container** – The Lambda function runs as a process without any isolation.

If you run the Lambda function in a container, complete the following steps to configure the process configuration for the Lambda function.

1. Configure the amount of memory and the system resources, such as volumes and devices, to make available to the container.

Under Container parameters, do the following.

a. For Memory size, enter the memory size that you want to allocate to the container. You can specify the memory size in MB or kB.

b. For Read-only sys folder, choose whether or not the container can read information from the device's /sys folder. The default is False.

2. (Optional) Configure the local volumes that the containerized Lambda function can access. When you define a volume, the AWS IoT Greengrass Core software mounts the source files to the destination inside the container.

a. Under Volumes, choose Add volume.

b. For each volume that you add, specify the following options:
Step 6: Create the Lambda function component

After you configure settings for your Lambda function component, choose Create to finish creating the new component.

To run the Lambda function on your core device, you can then deploy the new component to your core devices. For more information, see Deploy AWS IoT Greengrass components to devices (p. 383).

Import a Lambda function as a component (AWS CLI)

Use the CreateComponentVersion operation to create components from Lambda functions. When you call this operation, specify lambdaFunction to import a Lambda function.

Tasks

- Step 1: Define the Lambda function configuration (p. 368)
- Step 2: Create the Lambda function component (p. 381)

Step 1: Define the Lambda function configuration

1. Create a file called lambda-function-component.json, and then copy the following JSON object into the file. Replace the lambdaArn with the ARN of the Lambda function to import.

```json
{
```
Step 1: Define the Lambda function configuration

```json
"lambdaFunction": {
  "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1"
}
```

**Important**

You must specify an ARN that includes the version of the function to import. You can’t use version aliases like \$LATEST.

2. (Optional) Specify the name (componentName) of the component. If you omit this parameter, AWS IoT Greengrass creates the component with the name of the Lambda function.

```json
{
  "lambdaFunction": {
    "componentName": "com.example.HelloWorldLambda"
  }
}
```

3. (Optional) Specify the version (componentVersion) for the component. If you omit this parameter, AWS IoT Greengrass creates the component with the version of the Lambda function as a valid semantic version. For example, if your function version is 3, the component version becomes 3.0.0.

   **Note**

   Each component version that you upload must be unique. You can’t edit a component version after you upload it.

   AWS IoT Greengrass uses semantic versions for components. Semantic versions follow a major.minor.patch number system. For example, version 1.0.0 represents the first major release for a component. For more information, see the [semantic version specification](#).

```json
{
  "lambdaFunction": {
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0"
  }
}
```

4. (Optional) Specify the platforms that this Lambda function supports. Each platform contains a map of attributes that identify a platform. All core devices have attributes for operating system (os) and architecture (architecture). The AWS IoT Greengrass Core software may add other platform attributes. You can also specify custom platform attributes when you deploy the Greengrass nucleus component (p. 136) to a core device. Do the following:

   a. Add a list of platforms (componentPlatforms) to the Lambda function in lambda-function-component.json.

```json
{
  "lambdaFunction": {
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
    ]
  }
}
```

   b. Add each supported platform to the list. Each platform has a friendly name to identify it and a map of attributes. The following example specifies that this function supports x86 devices that run Linux.

```json
{
  "lambdaFunction": {
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "friendlyName": "x86 Linux",
        "attributes": {
          "os": "Linux",
          "architecture": "x86"  
        }
      }
    ]
  }
}
```
Step 1: Define the Lambda function configuration

```
{
  "name": "Linux x86",
  "attributes": {
    "os": "linux",
    "architecture": "x86"
  }
}
```

Your `lambda-function-component.json` might contain a document similar to the following example.

```
{
  "lambdaFunction": {
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {}
  }
}
```

5. (Optional) Specify the component dependencies for your Lambda function. When you deploy the Lambda function component, the deployment includes these dependencies for your function to run.

   **Important**
   To import a Lambda function that you created to run on AWS IoT Greengrass V1, you must define individual component dependencies for the features that your function uses, such as subscriptions, secrets, and stream manager. Define these components as hard dependencies (p. 341) so that your Lambda function component restarts if the dependency changes state. For more information, see Run V1 Lambda functions (p. 7).

   Do the following:

   a. Add a map of component dependencies (`componentDependencies`) to the Lambda function in `lambda-function-component.json`.

```
{
  "lambdaFunction": {
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
    }
  }
}
```
b. Add each component dependency to the map. Specify the component name as the key and specify an object with the following parameters:

- **versionRequirement** – The npm-style semantic version constraint that identifies the compatible versions of the component dependency. You can specify a single version or a range of versions. For more information about semantic version constraints, see the npm semver calculator.

- **dependencyType** – (Optional) The type of the dependency. Choose from the following:
  - **SOFT** – The Lambda function component doesn't restart if the dependency changes state.
  - **HARD** – The Lambda function component restarts if the dependency changes state.

The default is **HARD**.

The following example specifies that this Lambda function depends on any version in the first major version of the stream manager component (p. 306). The Lambda function component restarts when stream manager restarts or updates.

```json
{
  "aws.greengrass.StreamManager": {
    "versionRequirement": "^1.0.0",
    "dependencyType": "HARD"
  }
}
```

Your `lambda-function-component.json` might contain a document similar to the following example.

```json
{
  "lambdaFunction": {
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    }
  }
}
```

6. (Optional) Configure the Lambda function parameters to use to run the function. You can configure options such environment variables, message event sources, timeouts, and container settings. Do the following:

a. Add the Lambda parameters object (componentLambdaParameters) to the Lambda function in `lambda-function-component.json`. 
Step 1: Define the Lambda function configuration

```
{
  "lambdaFunction": {
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": []
    }
  }
}
```

b. (Optional) Specify the event sources to which the Lambda function subscribes for work messages. The Lambda function is called when it receives a message from an event source. You can subscribe this function to local publish/subscribe messages and AWS IoT Core MQTT messages. Do the following:

i. Add the list of event sources (eventSources) to the Lambda function parameters.

```
{
  "lambdaFunction": {
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        "eventSource1",
        "eventSource2"
      ]
    }
  }
}
```

ii. Add each event source to the list. Each event source has the following parameters:
Step 1: Define the Lambda function configuration

- **topic** – The topic to subscribe for messages. This topic supports MQTT topic wildcards (+ and #) when you subscribe to AWS IoT Core MQTT topics.

- **type** – The type of event source. Choose from the following:
  - **PUB_SUB** – Subscribe to local publish/subscribe messages.
  - **IOT_CORE** – Subscribe to AWS IoT Core MQTT messages.

The following example subscribes to AWS IoT Core MQTT on topics that match the `hello/world/+` topic filter.

```json

{
  "topic": "hello/world/+",
  "type": "IOT_CORE"
}
```

Your `lambda-function-component.json` might look similar to the following example.

```json

{
  "lambdaFunction": {
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": { "aws.greengrass.StreamManager": { "versionRequirement": "^1.0.0",
        "dependencyType": "HARD" }
    },
    "componentLambdaParameters": { "eventSources": [ {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
      ]
    }
  }
}
```

c. (Optional) Specify any of the following parameters in the Lambda function parameters object:

- **environmentVariables** – The map of environment variables that are available to the Lambda function when it runs.
- **execArgs** – The list of arguments to pass to the Lambda function when it runs.
- **inputPayloadEncodingType** – The type of payload that the Lambda function supports. Choose from the following options:
  - `json`
  - `binary`
Step 1: Define the Lambda function configuration

**Default: json**

- **pinned** – Whether or not the Lambda function is pinned. The default is `true`.
  - A pinned (or long-lived) Lambda function starts when AWS IoT Greengrass starts and keeps running in its own container.
  - A non-pinned (or on-demand) Lambda function starts only when it receives a work item and exits after it remains idle for a specified maximum idle time. If the function has multiple work items, the AWS IoT Greengrass Core software creates multiple instances of the function.

Use `maxIdleTimeInSeconds` to set the maximum idle time for your function.

- **timeoutInSeconds** – The maximum amount of time in seconds that the Lambda function can run before it times out. The default is 3 seconds.
- **statusTimeoutInSeconds** – The interval in seconds at which the Lambda function component sends status updates to the Lambda manager component. This parameter applies only to pinned functions. The default is 60 seconds.
- **maxIdleTimeInSeconds** – The maximum amount of time in seconds that a non-pinned Lambda function can idle before the AWS IoT Greengrass Core software stops its process. The default is 60 seconds.
- **maxInstancesCount** – The maximum number of instances that a non-pinned Lambda function can run at the same time. The default is 100 instances.
- **maxQueueSize** – The maximum size of the message queue for the Lambda function component. The AWS IoT Greengrass Core software stores messages in a FIFO (first-in-first-out) queue until it can run the Lambda function to consume each message. The default is 1,000 messages.

Your `lambda-function-component.json` might contain a document similar to the following example.

```json
{
    "lambdaFunction": {
        "componentName": "com.example.HelloWorldLambda",
        "componentVersion": "1.0.0",
        "componentPlatforms": [
            {
                "name": "Linux x86",
                "attributes": {
                    "os": "linux",
                    "architecture": "x86"
                }
            }
        ],
        "componentDependencies": {
            "aws.greengrass.StreamManager": {
                "versionRequirement": "^1.0.0",
                "dependencyType": "HARD"
            }
        },
        "componentLambdaParameters": {
            "eventSources": [
                {
                    "topic": "hello/world/+",
                    "type": "IOT_CORE"
                }
            ],
            "environmentVariables": {
                "LIMIT": "300"
            }
        }
    }
}
```
d. (Optional) Configure the container settings for the Lambda function. By default, Lambda functions run in an isolated runtime environment inside the AWS IoT Greengrass Core software. You can also choose to run the Lambda function as a process without any isolation. If you run the Lambda function in a container, you configure the memory size of the container and what system resources are available to the Lambda function. Do the following:

i. Add the Linux process parameters object (LinuxProcessParams) to the Lambda parameters object in lambda-function-component.json.

```json
{
    "lambdaFunction": {
        "componentName": "com.example.HelloWorldLambda",
        "componentVersion": "1.0.0",
        "componentPlatforms": [
            {
                "name": "Linux x86",
                "attributes": {
                    "os": "linux",
                    "architecture": "x86"
                }
            }
        ],
        "componentDependencies": {
            "aws.greengrass.StreamManager": {
                "versionRequirement": "^1.0.0",
                "dependencyType": "HARD"
            }
        },
        "componentLambdaParameters": {
            "eventSources": [
                {
                    "topic": "hello/world/+",
                    "type": "IOT_CORE"
                }
            ],
            "environmentVariables": {
                "LIMIT": "300"
            },
            "execArgs": [
                "-d"
            ],
            "inputPayloadEncodingType": "json",
            "pinned": true,
            "timeoutInSeconds": 120,
            "statusTimeoutInSeconds": 30,
            "maxIdleTimeoutInSeconds": 30,
            "maxInstancesCount": 50,
            "maxQueueSize": 500
        }
    }
}
```
Step 1: Define the Lambda function configuration

ii. (Optional) Specify whether or not the Lambda function runs in a container. Add the isolationMode parameter to the process parameters object, and choose from the following options:

- **GreengrassContainer** – The Lambda function runs in a container.
- **NoContainer** – The Lambda function runs as a process without any isolation.

The default is GreengrassContainer.

iii. (Optional) If you run the Lambda function in a container, you can configure the amount of memory and the system resources, such as volumes and devices, to make available to the container. Do the following:

A. Add the container parameters object (containerParams) to the Linux process parameters object in lambda-function-component.json.

```json
{
    "lambdaFunction": {
        "componentName": "com.example.HelloWorldLambda",
        "componentVersion": "1.0.0",
        "componentPlatforms": [
            {
                "name": "Linux x86",
                "attributes": {
                    "os": "linux",
                    "architecture": "x86"
                }
            }
        ],
        "componentDependencies": {
            "aws.greengrass.StreamManager": {
                "versionRequirement": "^1.0.0",
                "dependencyType": "HARD"
            }
        },
        "componentLambdaParameters": {
            "eventSources": [
                {
                    "topic": "hello/world/+",
                    "type": "IOT_CORE"
                }
            ],
            "environmentVariables": {
                "LIMIT": "300"
            },
            "execArgs": ["-d"],
            "inputPayloadEncodingType": "json",
            "pinned": true,
            "timeoutInSeconds": 120,
            "statusTimeoutInSeconds": 30,
            "maxIdleTimeInSeconds": 30,
            "maxInstancesCount": 50,
            "maxQueueSize": 500,
        }
    }
}
```
Step 1: Define the Lambda function configuration

B. (Optional) Add the `memorySizeInKB` parameter to specify the memory size of the container. The default is 16,384 KB (16 MB).

C. (Optional) Add the `mountROSysfs` parameter to specify whether or not the container can read information from the device's /sys folder. The default is false.

D. (Optional) Configure the local volumes that the containerized Lambda function can access. When you define a volume, the AWS IoT Greengrass Core software mounts the source files to the destination inside the container. Do the following:

I. Add the list of volumes (volumes) to the container parameters.
Step 1: Define the Lambda function configuration

```json
"mountROSysfs": true,
"volumes": [
]
}
}
}
}
```

II. Add each volume to the list. Each volume has the following parameters:

- **sourcePath** – The path to the source folder on the core device.
- **destinationPath** – The path to the destination folder in the container.
- **permission** – (Optional) The permission to access the source folder from the container. Choose from the following options:
  - **ro** – The Lambda function has read-only access to the source folder.
  - **rw** – The Lambda function has read-write access to the source folder.

The default is **ro**.

- **addGroupOwner** – (Optional) Whether or not to add the system group that runs the Lambda function component as an owner of the source folder. The default is **false**.

Your `lambda-function-component.json` might contain a document similar to the following example.

```json
{
   "lambdaFunction": {
      "componentName": "com.example.HelloWorldLambda",
      "componentVersion": "1.0.0",
      "componentPlatforms": [
         {
            "name": "Linux x86",
            "attributes": {
               "os": "linux",
               "architecture": "x86"
            }
         }
      ],
      "componentDependencies": {
         "aws.greengrass.StreamManager": {
            "versionRequirement": "^1.0.0",
            "dependencyType": "HARD"
         }
      },
      "componentLambdaParameters": {
         "eventSources": [
            {
               "topic": "hello/world/+",
               "type": "IOT_CORE"
            }
         ],
         "environmentVariables": {
            "LIMIT": "300"
         },
         "execArgs": [
            "-d"
         ]
      }
   }
}
```
Step 1: Define the Lambda function configuration

```
"inputPayloadEncodingType": "json",
"pinned": true,
"timeoutInSeconds": 120,
"statusTimeoutInSeconds": 30,
"maxIdleTimeInSeconds": 30,
"maxInstancesCount": 50,
"maxQueueSize": 500,
"linuxProcessParams": {
    "containerParams": {
        "memorySizeInKB": 32768,
        "mountROSysfs": true,
        "volumes": [
            {
                "sourcePath": "/var/data/src",
                "destinationPath": "/var/data/dest",
                "permission": "rw",
                "addGroupOwner": true
            }
        ]
    }
}
```

E. (Optional) Configure the local system devices that the containerized Lambda function can access. Do the following:

I. Add the list of system devices (devices) to the container parameters.

```
{
    "lambdaFunction": {
        "componentName": "com.example.HelloWorldLambda",
        "componentVersion": "1.0.0",
        "componentPlatforms": [
            {
                "name": "Linux x86",
                "attributes": {
                    "os": "linux",
                    "architecture": "x86"
                }
            }
        ],
        "componentDependencies": {
            "aws.greengrass.StreamManager": {
                "versionRequirement": "^1.0.0",
                "dependencyType": "HARD"
            }
        },
        "componentLambdaParameters": {
            "eventSources": [
                {
                    "topic": "hello/world/+",
                    "type": "IOT_CORE"
                }
            ],
            "environmentVariables": {
                "LIMIT": "300"
            },
            "execArgs": [
                "-d"
            ],
            "inputPayloadEncodingType": "json",
```
Step 1: Define the Lambda function configuration

```
"pinned": true,
"timeoutInSeconds": 120,
"statusTimeoutInSeconds": 30,
"maxIdleTimeInSeconds": 30,
"maxInstancesCount": 50,
"maxQueueSize": 500,
"linuxProcessParams": {
  "containerParams": {
    "memorySizeInKB": 32768,
    "mountROSysfs": true,
    "volumes": [
      {
        "sourcePath": "/var/data/src",
        "destinationPath": "/var/data/dest",
        "permission": "rw",
        "addGroupOwner": true
      }
    ],
    "devices": []
  }
}
```

II. Add each system device to the list. Each system device has the following parameters:

- **path** – The path to the system device on the core device.
- **permission** – (Optional) The permission to access the system device from the container. Choose from the following options:
  - **ro** – The Lambda function has read-only access to the system device.
  - **rw** – The Lambda function has read-write access to the system device.

The default is **ro**.

- **addGroupOwner** – (Optional) Whether or not to add the system group that runs the Lambda function component as an owner of the system device. The default is **false**.

Your `lambda-function-component.json` might contain a document similar to the following example.

```
{
  "lambdaFunction": {
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
      }
    }
  }
}
```
Step 2: Create the Lambda function component

1. Run the following command to create the Lambda function component from `lambda-function-component.json`.

   ```
   aws greengrassv2 create-component-version --cli-input-json file://lambda-function-component.json
   ```

   The response looks similar to the following example if the request succeeds.

7. (Optional) Add tags (tags) for the component. For more information, see Tag your AWS IoT Greengrass Version 2 resources (p. 759).

---

**Step 2: Create the Lambda function component**

1. Run the following command to create the Lambda function component from `lambda-function-component.json`.

   ```
   aws greengrassv2 create-component-version --cli-input-json file://lambda-function-component.json
   ```

   The response looks similar to the following example if the request succeeds.
Step 2: Create the Lambda function component

Copy the arn from the output to check the state of the component in the next step.

2. When you create a component, its state is REQUESTED. Then, AWS IoT Greengrass validates that the component is deployable. You can run the following command to query the component status and verify that your component is deployable. Replace the arn with the ARN from the previous step.

```bash
aws greengrassv2 describe-component \
```

If the component validates, the response indicates that the component state is DEPLOYABLE.

```json
{
  "componentName": "com.example.HelloWorldLambda",
  "componentVersion": "1.0.0",
  "creationTimestamp": "2020-12-15T20:56:34.376000-08:00",
  "publisher": "AWS Lambda",
  "status": {
    "componentState": "DEPLOYABLE",
    "message": "NONE",
    "errors": {}
  },
  "platforms": [
    {
      "name": "Linux x86",
      "attributes": {
        "architecture": "x86",
        "os": "linux"
      }
    }
  ]
}
```

After the component is DEPLOYABLE, you can deploy the Lambda function to your core devices. For more information, see Deploy AWS IoT Greengrass components to devices (p. 383).
Deploy AWS IoT Greengrass components to devices

You can deploy components to individual devices or groups of devices with AWS IoT Greengrass. With *deployments*, you define the AWS IoT Greengrass components and configurations to deploy to core devices. AWS IoT Greengrass deploys to *target devices*, which are AWS IoT things or thing groups that represent Greengrass core devices. AWS IoT Greengrass uses AWS IoT Core jobs to deploy to your core devices, so you can configure how the job rolls out to your devices.

Each core device runs the combination of the software from the deployments that target the device. However, deployments to the same target overwrite previous deployments to that target. When you create a deployment, you define the components and configurations to apply to the core devices’ existing software. When you revise a deployment for a target, you replace the components from the previous revision with the components in the new revision. For example, you deploy components A and B to a thing group X. You then deploy components B and C to a different thing group Y. As a result, the components A, B, and C all run on any core device that is a member of both groups. Then, you create another deployment to thing group X that specifies only component B. As a result, the core devices in that group no longer run component A.

**Note**

When you remove a core device from a thing group, AWS IoT Greengrass doesn't delete that thing group's components from the device. Because of this, we recommend that you don't remove core devices from thing groups where you deploy components.

To remove a component from a device, where the device is no longer a member of the thing group that deploys that component, use the `deployment create` command of the Greengrass CLI. Specify the component to remove with the `--remove` argument, and specify the thing group with the `--groupId` argument.

Deployments are continuous. When you create a deployment, AWS IoT Greengrass rolls out the deployment to target devices that are online. If a target device isn't online, then it receives the deployment the next time it connects to AWS IoT Greengrass. When you add a core device to a target thing group, AWS IoT Greengrass sends the device the latest deployment for that thing group.

Each target thing or thing group can have one deployment at a time. This means that when you create a deployment for a target, AWS IoT Greengrass no longer deploys the previous revision of that target's deployment.

Deployments provide several options that let you control which devices receive an update and how the update deploys. When you create a deployment, you can configure the following options:

- **AWS IoT Greengrass components**

  Define the components to install and run on the target devices. AWS IoT Greengrass components are software modules that you deploy and run on Greengrass core devices. Devices receive components only if the component supports the device's platform. This lets you deploy to groups of devices even if the target devices run on multiple platforms. If a component doesn't support the device's platform, the component doesn't deploy to the device.

  You can deploy custom components and AWS-provided components to your devices. When you deploy a component, AWS IoT Greengrass identifies any component dependencies and deploys them too. For more information, see [Develop AWS IoT Greengrass components](#) and [AWS-provided components](#).
You define the version and configuration update to deploy for each component. The configuration update specifies how to modify the component's existing configuration on the core device, or the component's default configuration if the component doesn't exist on the core device. You can specify which configuration values to reset to default values and the new configuration values to merge onto the core device. For more information, see Update component configurations (p. 390).

**Important**
When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all component dependencies for that component. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly. To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment (p. 384). For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA) (p. 129).

- **Deployment policies**
  Define when it's safe to deploy a configuration and what to do if the deployment fails. You can specify whether or not to wait for components to report that they can update. You can also specify whether or not to roll back devices to their previous configuration if they apply a deployment that fails.

- **Stop configuration**
  Define when and how to stop a deployment. The deployment stops and fails if the criteria that you define are met. For example, you can configure a deployment to stop if a percentage of devices fail to apply that deployment after a minimum number of devices receive it.

- **Rollout configuration**
  Define the rate at which a deployment rolls out to the target devices. You can configure an exponential rate increase with minimum and maximum rate bounds.

- **Timeout configuration**
  Define the maximum amount of time each device has to apply a deployment. If a device exceeds the duration that you specify, then the device fails to apply the deployment.

**Important**
Custom components can define artifacts in S3 buckets. When the AWS IoT Greengrass Core software deploys a component, it downloads the component's artifacts from the AWS Cloud. Core device roles don't allow access to S3 buckets by default. To deploy custom components that define artifacts in an S3 bucket, the core device role must grant permissions to download artifacts from that bucket. For more information, see Allow access to S3 buckets for component artifacts (p. 628).

**Topics**
- Create deployments (p. 384)
- Revise deployments (p. 393)
- Cancel deployments (p. 394)
- Check deployment status (p. 395)

## Create deployments

You can create a deployment that targets a thing or thing group.
When you create a deployment, you configure the software components to deploy and how the deployment job rolls out to target devices. You can define the deployment in the JSON file that you provide to the AWS CLI.

The deployment target determines the devices on which you want to run your components. To deploy to one core device, specify a thing. To deploy to multiple core devices, specify a thing group that includes those devices. For more information about how to configure thing groups, see Static thing groups and Dynamic thing groups in the AWS IoT Developer Guide.

Follow the steps in this section to create a deployment to a target for the first time. For more information about how to update the software components on a target that has a deployment, see Revise deployments (p. 393).

To create a deployment (AWS CLI)

1. Create a file called `deployment.json`, and then copy the following JSON object into the file. Replace `targetArn` with the ARN of the AWS IoT thing or thing group to target for the deployment. Thing and thing group ARNs have the following format:

   • Thing: arn:aws:iot:region:account-id:thing/thingName
   • Thing group: arn:aws:iot:region:account-id:thinggroup/thingGroupName

   ```json
   {
     "targetArn": "targetArn",
     "components": {
       ...
     }
   }
   ```

2. Add each component to deploy the target devices. To do so, add key-value pairs to the `components` object, where the key is the component name, and the value is an object that contains the details for that component. Specify the following details for each component that you add:

   • `version` – The component version to deploy.
   • `configurationUpdate` – The configuration update (p. 390) to deploy. The update is a patch operation that modifies the component's existing configuration on each target device, or the component's default configuration if it doesn't exist on the target device. You can specify the following configuration updates:
     • `reset` – (Optional) A list of JSON pointers that define the configuration values to reset to their default values on the target device. The AWS IoT Greengrass Core software applies reset updates before it applies merge updates. For more information, see Reset updates (p. 390).
     • `merge` – (Optional) A JSON document that defines the configuration values to merge onto the target device. You must serialize the JSON document as a string. For more information, see Merge updates (p. 391).
   • `runWith` – (Optional) The system process options that the AWS IoT Greengrass Core software uses to run this component's processes on the core device. If you omit a parameter in the `runWith` object, the AWS IoT Greengrass Core software uses the default values that you configure on the Greengrass nucleus component (p. 136).

You can specify any of the following options:

• `posixUser` – The POSIX system user and (optional) group to use to run this component's processes. You can set the system user and group for generic and Lambda components. Specify the user and group separated by a colon (:) in the following format: `user:group`. If you don't specify a group, the AWS IoT Greengrass Core software uses the primary group for the user. For more information, see Configure the user and group that run components (p. 122).
• systemResourceLimits – The system resource limits to apply to this component's processes. You can apply system resource limits to generic and non-containerized Lambda components. For more information, see Configure system resource limits for components (p. 124).

You can specify any of the following options:

• cpus – The maximum amount of CPU time that this component's processes can use on the core device. A core device's total CPU time is equivalent to the device's number of CPU cores. For example, on a core device with 4 CPU cores, you can set this value to 2 to limit this component's processes to 50 percent usage of each CPU core. On a device with 1 CPU core, you can set this value to 0.25 to limit this component's processes to 25 percent usage of the CPU. If you set this value to a number greater than the number of CPU cores, the AWS IoT Greengrass Core software doesn't limit the component's CPU usage.

• memory – The maximum amount of RAM (in kilobytes) that this component's processes can use on the core device.

This feature is available for v2.4.0 and later of the Greengrass nucleus component (p. 136).

Example Example basic configuration update

The following example components object specifies to deploy a component, com.example.PythonRuntime, that expects a configuration parameter named pythonVersion.

```json
{
  "targetArn": "targetArn",
  "components": {
    "com.example.PythonRuntime": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "merge": "{"\"pythonVersion\":\"3.7\"}"
      }
    }
  }
}
```

Example Example configuration update with reset and merge updates

Consider an example industrial dashboard component, com.example.IndustrialDashboard, that has the following default configuration.

```json
{
  "name": null,
  "mode": "REQUEST",
  "network": {
    "useHttps": true,
    "port": {
      "http": 80,
      "https": 443
    },
  },
  "tags": []
}
```

The following configuration update specifies the following instructions:

1. Reset the HTTPS setting to its default value (true).
2. Reset the list of industrial tags to an empty list.
3. Merge a list of industrial tags that identify temperature and pressure data streams for two boilers.
The following example `components` object specifies to deploy this industrial dashboard component and configuration update.

```
{
  "targetArn": "targetArn",
  "components": {
    "com.example.IndustrialDashboard": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "reset": [
          "/network/useHttps",
          "/tags"
        ],
        "merge": "{\"tags\":[[\"boiler/1/temperature\",\"/boiler/1/pressure\",\"/boiler/2/temperature\",\"/boiler/2/pressure\"]]}"
      }
    }
  }
}
```

3. (Optional) Define deployment policies for the deployment. You can configure when core devices can safely apply a deployment or what to do if a core device fails to apply the deployment. To do so, add a `deploymentPolicies` object to `deployment.json`, and then do any of the following:

1. (Optional) Specify the component update policy (`componentUpdatePolicy`). This policy defines whether or not the deployment lets components defer an update until they are ready to update. For example, components may need to clean up resources or finish critical actions before they can restart to apply an update. This policy also defines the amount of time that components have to respond to an update notification.

This policy is an object with the following parameters:

- **action** – (Optional) Whether or not to notify components and wait for them to report when they're ready to update. Choose from the following options:
  - **NOTIFY_COMPONENTS** – The deployment notifies each component before it stops and updates that component. Components can use the `SubscribeToComponentUpdates (p. 448)` IPC operation to receive these notifications.
  - **SKIP_NOTIFY_COMPONENTS** – The deployment doesn't notify components or wait for them to be safe to update.

Defaults to **NOTIFY_COMPONENTS**.
Create deployments

• timeoutInSeconds The amount of time in seconds that each component has to respond to an update notification with the DeferComponentUpdate (p. 449) IPC operation. If the component doesn’t respond within this amount of time, then the deployment proceeds on the core device.

Defaults to 60 seconds.

2. (Optional) Specify the configuration validation policy (configurationValidationPolicy). This policy defines how long each component has to validate a configuration update from a deployment. Components can use the SubscribeToValidateConfigurationUpdates (p. 454) IPC operation to subscribe to notifications for their own configuration updates. Then, components can use the SendConfigurationValidityReport (p. 454) IPC operation to tell the AWS IoT Greengrass Core software if the configuration update is valid. If the configuration update isn’t valid, the deployment fails.

This policy is an object with the following parameter:
• timeoutInSeconds (Optional) The amount of time in seconds that each component has to validate a configuration update. If the component doesn’t respond within this amount of time, then the deployment proceeds on the core device.

Defaults to 30 seconds.

3. (Optional) Specify the failure handling policy (failureHandlingPolicy). This policy is a string that defines whether or not to roll back devices if the deployment fails. Choose from the following options:
• ROLLBACK – If the deployment fails on a core device, then the AWS IoT Greengrass Core software rolls back that core device to its previous configuration.
• DO NOTHING – If the deployment fails on a core device, then the AWS IoT Greengrass Core software keeps the new configuration. This can result in broken components if the new configuration isn’t valid.

Defaults to ROLLBACK.

Your deployment in deployment.json may look similar to the following example:

```json
{
    "targetArn": "targetArn",
    "components": {
        "com.example.IndustrialDashboard": {
            "componentVersion": "1.0.0",
            "configurationUpdate": {
                "reset": [
                    "/network/useHttps",
                    "/tags"
                ],
                "merge": "{"tags": ["/boiler/1/temperature","/boiler/1/pressure","/boiler/2/temperature","/boiler/2/pressure"]}"
            }
        },
    },
    "deploymentPolicies": {
        "componentUpdatePolicy": {
            "action": "NOTIFY_COMPONENTS",
            "timeoutInSeconds": 30
        },
        "configurationValidationPolicy": {
            "timeoutInSeconds": 60
        },
        "failureHandlingPolicy": "ROLLBACK"
    }
}
```
4. (Optional) Define how the deployment stops, rolls out, or times out. AWS IoT Greengrass uses AWS IoT Core jobs to send deployments to core devices, so these options are identical to the configuration options for AWS IoT Core jobs. For more information, see Job rollout and abort configuration in the AWS IoT Developer Guide.

To define the job options, add an `iotJobConfigurations` object to `deployment.json`. Then, define the options to configure.

Your deployment in `deployment.json` may look similar to the following example:

```json
{
  "targetArn": "targetArn",
  "components": {
    "com.example.IndustrialDashboard": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "reset": [
          "/network/useHttps",
          "/tags"
        ],
        "merge": "{"tags":="{"/boiler/1/temperature","/boiler/1/pressure","/boiler/2/temperature","/boiler/2/pressure"}"
      }
    }
  },
  "deploymentPolicies": {
    "componentUpdatePolicy": {
      "action": "NOTIFY_COMPONENTS",
      "timeoutInSeconds": 30
    },
    "configurationValidationPolicy": {
      "timeoutInSeconds": 60
    },
    "failureHandlingPolicy": "ROLLBACK"
  },
  "iotJobConfigurations": {
    "abortConfig": {
      "criteriaList": [
        {
          "action": "CANCEL",
          "failureType": "ALL",
          "minNumberOfExecutedThings": 100,
          "thresholdPercentage": 5
        }
      ]
    },
    "jobExecutionsRolloutConfig": {
      "exponentialRate": {
        "baseRatePerMinute": 5,
        "incrementFactor": 2,
        "rateIncreaseCriteria": {
          "numberOfNotifiedThings": 10,
          "numberOfSucceededThings": 5
        }
      },
      "maximumPerMinute": 50
    },
    "timeoutConfig": {
      "inProgressTimeoutInMinutes": 5
    }
  }
}
```
5. (Optional) Add tags (tags) for the deployment. For more information, see Tag your AWS IoT Greengrass Version 2 resources (p. 759).

6. Run the following command to create the deployment from deployment.json.

```bash
aws greengrassv2 create-deployment --cli-input-json file://deployment.json
```

The response includes a deploymentId that identifies this deployment. You can use the deployment ID to check the status of the deployment. For more information, see Check deployment status (p. 395).

Update component configurations

Component configurations are JSON objects that define the parameters for each component. Each component's recipe defines its default configuration, which you modify when you deploy components to core devices.

When you create a deployment, you can specify the configuration update to apply for each component. Configuration updates are patch operations, which means that the update modifies the component configuration that exists on the core device. If the core device doesn't have the component, then the configuration update modifies and applies the default configuration for that deployment.

The configuration update defines reset updates and merge updates. Reset updates define which configuration values to reset to their defaults or remove. Merge updates define the new configuration values to set for the component. When you deploy a configuration update, the AWS IoT Greengrass Core software runs the reset update before the merge update.

Components can validate the configuration updates that you deploy. The component subscribes to receive a notification when a deployment changes its configuration, and it can reject a configuration that it doesn't support. For more information, see Interact with component configuration (p. 451).

Topics

- Reset updates (p. 390)
- Merge updates (p. 391)
- Examples (p. 391)

Reset updates

Reset updates define which configuration values to reset to their default values on the core device. If a configuration value doesn't have a default value, then the reset update removes that value from the component's configuration. This can help you fix a component that breaks as the result of an invalid configuration.

Use a list of JSON pointers to define which configuration values to reset. JSON pointers start with a forward slash / . To identify a value in a nested component configuration, use forward slashes (/) to separate the keys for each level in the configuration. For more information, see the JSON pointer specification.

Note
You can reset only an entire list to its default values. You can't use reset updates to reset an individual element in a list.

To reset a component's entire configuration to its default values, specify a single empty string as the reset update.
"reset": ["""]

Merge updates

Merge updates define the configuration values to insert into the component configuration on the core. The merge update is a JSON object that the AWS IoT Greengrass Core software merges after it resets the values in the paths that you specify in the reset update. When you use the AWS CLI or AWS SDKs, you must serialize this JSON object as a string.

You can merge a key-value pair that doesn't exist in the component's default configuration. You can also merge a key-value pair that has a different type than the value with the same key. The new value replaces the old value. This means that you can change the configuration object's structure.

You can merge null values and empty strings, lists, and objects.

Note

You can't use merge updates for the purpose of inserting or appending an element to a list. You can replace an entire list, or you can define an object where each element has a unique key.

AWS IoT Greengrass uses JSON for configuration values. JSON specifies a number type but doesn't differentiate between integers and floats. As a result, configuration values might convert to floats in AWS IoT Greengrass. To ensure that your component uses the correct data type, we recommend that you define numeric configuration values as strings. Then, have your component parse them as integers or floats. This ensures that your configuration values have the same type in the configuration and on your core device.

Examples

The following example demonstrates configuration updates for a dashboard component that has the following default configuration. This example component displays information about industrial equipment.

```
{
   "name": null,
   "mode": "REQUEST",
   "network": {
      "useHttps": true,
      "port": {
         "http": 80,
         "https": 443
      }
   },
   "tags": []
}
```

Then, you apply the following configuration update, which specifies a merge update but not a reset update. This configuration tells the component to display the dashboard on HTTP port 8080 with data from two boilers.

```
{
   "merge": {
      "name": "Factory 2A",
      "network": {
         "useHttps": false,
         "port": {
            "http": 8080
         }
      },
      "tags": [
         "/boiler/1/temperature",
         "/boiler/2/temperature"
      ]
   }
}
```
Update component configurations

"/boiler/1/pressure",
"/boiler/2/temperature",
"/boiler/2/pressure"
]
}
}

After this update, the dashboard component has the following configuration.

{
"name": "Factory 2A",
"mode": "REQUEST",
"network": {
"useHttps": false,
"port": {
"http": 8080,
"https": 443
}
},
"tags": [
"/boiler/1/temperature",
"/boiler/1/pressure",
"/boiler/2/temperature",
"/boiler/2/pressure"
]
}

Then, apply the following configuration update to display the dashboard on the default HTTPS port with data from different boilers.

{
"reset": [
"/network/useHttps",
"/tags"
],
"merge": {
"tags": [
"/boiler/3/temperature",
"/boiler/3/pressure",
"/boiler/4/temperature",
"/boiler/4/pressure"
]
}
}

After this update, the dashboard component has the following configuration.

{
"name": "Factory 2A",
"mode": "REQUEST",
"network": {
"useHttps": true,
"port": {
"http": 8080,
"https": 443
}
},
"tags": [
"/boiler/3/temperature",
"/boiler/3/pressure",
"/boiler/4/temperature",
"/boiler/4/pressure",
"/boiler/5/temperature",
"/boiler/5/pressure"
]
Revise deployments

Each target thing or thing group can have one active deployment at a time. When you create a deployment for a target that already has a deployment, the software components in the new deployment replace those from the previous deployment. If the new deployment doesn't define a component that the previous deployment defines, the AWS IoT Greengrass Core software removes that component from the target core devices. You can revise an existing deployment so that you don't remove the components that run on core devices from a previous deployment to a target.

You use the CreateDeployment operation to revise deployments. This operation is the same operation that you use to create deployments (p. 384).

To revise a deployment (AWS CLI)

1. To revise a deployment, you create a deployment that defines the same components and configuration as a previous deployment. Run the following command to list the deployments for the target to revise. Replace \texttt{targetArn} with the ARN of the target AWS IoT thing or thing group.

   ```
   aws greengrassv2 list-deployments \
   --target-arn targetArn
   ```

   The response contains a list with the latest deployment for the target. Copy the \texttt{deploymentId} from the response to use in the next step.

   \textbf{Note}
   
   You can also revise a deployment other than the latest revision for the target. Specify the \texttt{--history-filter} \texttt{ALL} argument to list all deployments for the target. Then, copy the ID of the deployment to revise.

2. Run the following command to get the deployment's details, which include its metadata, components, and job configuration. Replace \texttt{deploymentId} with the ID from the previous step.

   ```
   aws greengrassv2 get-deployment \
   --deployment-id deploymentId
   ```

   The response contains the deployment's details.

3. Create a file called \texttt{deployment.json} and copy the previous command's response into the file.

4. Remove the following key-value pairs from the JSON object in \texttt{deployment.json}:

   - \texttt{deploymentId}
   - \texttt{revisionId}
   - \texttt{iotJobId}
   - \texttt{iotJobArn}
   - \texttt{creationTimestamp}
   - \texttt{isLatestForTarget}
   - \texttt{deploymentStatus}

   The CreateDeployment operation expects a payload with the following structure.

   ```
   |
   |
   |
   ```
5. In deployment.json, do any of the following:
   • Change the deployment's name (deploymentName).
   • Change the deployment's components (components).
   • Change the deployment's policies (deploymentPolicies).
   • Change the deployment's job configuration (iotJobConfiguration).
   • Change the deployment's tags (tags).

   For more information about how to define these deployment details, see Create deployments (p. 384).

6. Run the following command to create the deployment from deployment.json.

   ```bash
   aws greengrassv2 create-deployment
   --cli-input-json file://deployment.json
   ```

   The response includes a deploymentId that identifies this deployment. You can use the deployment ID to check the status of the deployment. For more information, see Check deployment status (p. 395).

---

**Cancel deployments**

You can cancel an active deployment to prevent its software components from installing on AWS IoT Greengrass core devices. If you cancel a deployment that targets a thing group, core devices that you add to the group won't receive that continuous deployment. If a core device already runs the deployment, you won't change the components on that device when you cancel the deployment. You must create a new deployment (p. 384) or revise the deployment (p. 393) to modify the components that run on the core devices that received the canceled deployment.

**To cancel a deployment (AWS CLI)**

1. Run the following command to find the ID of the latest deployment revision for a target. The latest revision is the only deployment that can be active for a target, because previous deployments cancel when you create a new revision. Replace `targetArn` with the ARN of the target AWS IoT thing or thing group.

   ```bash
   aws greengrassv2 list-deployments
   --target-arn targetArn
   ```

   The response contains a list with the latest deployment for the target. Copy the deploymentId from the response to use in the next step.

2. Run the following command to cancel the deployment. Replace `deploymentId` with the ID from the previous step.

   ```bash
   aws greengrassv2 cancel-deployment
   --deployment-id deploymentId
   ```

   If the operation succeeds, the deployment status changes to CANCELED.
Check deployment status

You can check the status of a deployment that you create in AWS IoT Greengrass. You can also check the status of the AWS IoT jobs that roll out the deployment to each core device. While a deployment is active, the AWS IoT job's status is IN_PROGRESS. After you create a new revision of a deployment, the status of the previous revision's AWS IoT job changes to CANCELLED.

Topics

- Check deployment status (p. 395)
- Check device deployment status (p. 395)

Check deployment status

You can check the status of a deployment that you identify by its target or its ID.

To check deployment status by target (AWS CLI)

- Run the following command to retrieve the status of the latest deployment for a target. Replace targetArn with the ARN of the AWS IoT thing or thing group that the deployment targets.

```
aws greengrassv2 list-deployments \
  --target-arn targetArn
```

The response contains a list with the latest deployment for the target. This deployment object includes the status of the deployment.

To check deployment status by ID (AWS CLI)

- Run the following command to retrieve the status of a deployment. Replace deploymentId with the ID of the deployment to query.

```
aws greengrassv2 get-deployment \
  --deployment-id deploymentId
```

The response contains the status of the deployment.

Check device deployment status

You can check the status of a deployment job that applies to an individual core device.

To check deployment job status (AWS CLI)

- Run the following command to retrieve the status of all deployment jobs for a core device. Replace coreDeviceName with the name of the core device to query.

```
aws greengrassv2 list-effective-deployments \
  --core-device-thing-name coreDeviceName
```

The response contains the list of deployment jobs for the core device. You can identify the job for a deployment by the jobs deploymentId or targetArn. Each deployment job contains the status of the job on the core device.
Use the AWS IoT Device SDK for interprocess communication (IPC)

Components running on your core device can use the AWS IoT Greengrass Core interprocess communication (IPC) library in the AWS IoT Device SDK to communicate with other AWS IoT Greengrass components and processes. To develop and run custom components that use IPC, you must use the AWS IoT Device SDK to connect to the AWS IoT Greengrass Core IPC service and perform IPC operations.

The IPC interface supports two types of operations:

- **Request/response**
  Components send a request to the IPC service and receive a response that contains the result of the request.

- **Subscription**
  Components send a subscription request to the IPC service and expect a stream of event messages in response. Components provide a subscription handler that handles event messages, errors, and stream closure. The AWS IoT Device SDK includes a handler interface with the correct response and event types for each IPC operation. For more information, see Subscribe to IPC event streams (p. 405).

**Topics**
- Supported SDKs for interprocess communication (p. 396)
- Connect to the AWS IoT Greengrass Core IPC service (p. 397)
- Authorize components to perform IPC operations (p. 403)
- Subscribe to IPC event streams (p. 405)
- Publish/subscribe local messages (p. 410)
- Publish/subscribe AWS IoT Core MQTT messages (p. 433)
- Interact with component lifecycle (p. 446)
- Interact with component configuration (p. 451)
- Retrieve secret values (p. 455)
- Interact with local shadows (p. 462)

**Supported SDKs for interprocess communication**

The AWS IoT Greengrass Core IPC libraries are included in the following AWS IoT Device SDK versions.

- **AWS IoT Device SDK for Java v2** (v1.2.10 or later)
  For more information about using the AWS IoT Device SDK for Java v2 to connect to the AWS IoT Greengrass Core IPC service, see Use AWS IoT Device SDK for Java v2 (p. 397).

- **AWS IoT Device SDK for Python v2** (v1.5.3 or later)
  For more information about using the AWS IoT Device SDK for Python v2 to connect to the AWS IoT Greengrass Core IPC service, see Use AWS IoT Device SDK for Python v2 (p. 399).

- **AWS IoT Device SDK for C++ v2** (v1.13.0 or later)
Connect to the AWS IoT Greengrass Core IPC service

To use interprocess communication in your custom component, you must create a connection to an IPC server socket that the AWS IoT Greengrass Core software runs. Complete the following tasks to download and use the AWS IoT Device SDK in the language of your choice.

Use AWS IoT Device SDK for Java v2

To use the AWS IoT Device SDK for Java v2

1. Download the AWS IoT Device SDK for Java v2 (v1.2.10 or later).
2. Do one of the following to run your custom code in your component:
   - Build your component as a JAR file that includes the AWS IoT Device SDK, and run this JAR file in your component recipe.
   - Define the AWS IoT Device SDK JAR as a component artifact, and add that artifact to the classpath when you run your application in your component recipe.
3. Create a connection to the AWS IoT Greengrass Core IPC service. The IPC client, GreengrassCoreIPCClient, requires an EventStreamRPCConnection. Download the following IPCUtils class that provides this connection for you.

```java
import software.amazon.awssdk.crt.io.ClientBootstrap;
import software.amazon.awssdk.crt.io.EventLoopGroup;
import software.amazon.awssdk.crt.io.SocketOptions;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnectionConfig;
import software.amazon.awssdk.eventstreamrpc.GreengrassConnectMessageSupplier;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public final class IPCUtils {
    // Port number is not used in domain sockets.
    // It is ignored but the field needs to be set when creating socket connection
    public static final int DEFAULT_PORT_NUMBER = 8033;
    private static EventStreamRPCConnection clientConnection = null;

    private IPCUtils() {
    }

    public static EventStreamRPCConnection getEventStreamRpcConnection()
        throws ExecutionException, InterruptedException {
        String ipcServerSocketPath = System.getenv("AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT");
        String authToken = System.getenv("SVCUID");
        SocketOptions socketOptions = IPCUtils.getSocketOptionsForIPC();
        if (clientConnection == null) {
            return null;
        }

        return clientConnection;
    }
}
```
clientConnection = connectToGGCOverEventStreamIPC(socketOptions, authToken, ipcServerSocketPath);
}
return clientConnection;

// removed dependency on kernel, as it is only being used to pull ipcServerSocketPath
private static EventStreamRPCConnection
connectToGGCOverEventStreamIPC(SocketOptions socketOptions, String
authToken, String ipcServerSocketPath)
throws ExecutionException, InterruptedException {
try (EventLoopGroup elGroup = new EventLoopGroup(1);
ClientBootstrap clientBootstrap = new ClientBootstrap(elGroup, null)) {
final EventStreamRPCConnectionConfig config =
new EventStreamRPCConnectionConfig(clientBootstrap, elGroup, socketOptions, null,
ipcServerSocketPath, DEFAULT_PORT_NUMBER,
GreengrassConnectMessageSupplier.connectMessageSupplier(authToken));
final CompletableFuture<Void> connected = new CompletableFuture<>();
final EventStreamRPCConnection connection = new EventStreamRPCConnection(config);
final boolean disconnected[] = {false};
final int disconnectedCode[] = {-1};
//this is a bit cumbersome but does not prevent a convenience wrapper from
exposing a sync
//connect() or a connect() that returns a CompletableFuture that errors
//this could be wrapped by utility methods to provide a more
connection.connect(new EventStreamRPCConnection.LifecycleHandler() {
//only called on successful connection.
// That is full on Connect -> ConnectAck(ConnectionAccepted=true)
@Override
public void onConnect() {
connected.complete(null);
}

@Override
public void onDisconnect(int errorCode) {
disconnected[0] = true;
disconnectedCode[0] = errorCode;
clientConnection = null;
}

//This on error is for any errors that is connection level, including
problems during connect()
@Override
public boolean onError(Throwable t) {
connected.completeExceptionally(t);
clientConnection = null;
return true;    //hints at handler to disconnect due to this error
}
});
connected.get();
return connection;
}

private static SocketOptions getSocketOptionsForIPC() {
SocketOptions socketOptions = new SocketOptions();
socketOptions.connectTimeOutMs = 3000;
socketOptions.domain = SocketOptions.SocketDomain.LOCAL;
Connect to the AWS IoT Greengrass Core IPC service

4. Use the following code to create the IPC client.

```java
try (EventStreamRPCConnection eventStreamRPCConnection = IPCUtils.getEventStreamRpcConnection()) {
    GreengrassCoreIPCClient ipcClient = new GreengrassCoreIPCClient(eventStreamRPCConnection);
} catch (Exception e) {
    LOGGER.log(Level.SEVERE, "Exception occurred when using IPC.", e);
    System.exit(1);
}
```

Use AWS IoT Device SDK for Python v2

To use the AWS IoT Device SDK for Python v2

1. Download the AWS IoT Device SDK for Python (v1.5.3 or later).
2. Add the SDK's installation steps to the install lifecycle in your component's recipe.
3. Create a connection to the AWS IoT Greengrass Core IPC service. Complete the following steps to create the IPC client and establish a connection.

SDK v1.5.4 or later

Use the following code to create the IPC client.

```python
import awsiot.greengrasscoreipc

click = awsiot.greengrasscoreipc.connect()
```

SDK v1.5.3

1. Download the following `IPCUtils` class that provides the IPC server connection for you.

```python
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

import os

from awscrt.io import (ClientBootstrap, DefaultHostResolver, EventLoopGroup, SocketDomain, SocketOptions,
) from awsiot.eventstreamrpc import Connection, LifecycleHandler, MessageAmendment

TIMEOUT = 10

class IPCUtils:
    def connect(self):
        elg = EventLoopGroup()
        resolver = DefaultHostResolver(elg)
```

399
Connect to the AWS IoT Greengrass Core IPC service

```python
bootstrap = ClientBootstrap(elg, resolver)
socket_options = SocketOptions()
socket_options.domain = SocketDomain.Local
amender = MessageAmendment.create_static_authtoken_amender(os.getenv("SVCUID"))
hostname = os.getenv("AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT")
connection = Connection(
    host_name=hostname,
    port=8033,
    bootstrap=bootstrap,
    socket_options=socket_options,
    connect_message_amender=amender,
)
self.lifecycle_handler = LifecycleHandler()
connect_future = connection.connect(self.lifecycle_handler)
connect_future.result(TIMEOUT)
return connection
```

2. Use the following code to create the IPC client.

```python
import awsiot.greengrasscoreipc.client as client
ipc_utils = IPCUtils()
connection = ipc_utils.connect()
ipc_client = client.GreengrassCoreIPCClient(connection)
```

Use AWS IoT Device SDK for C++ v2

**Important**
To build the AWS IoT Device SDK v2 for C++, a device must have the following tools:

- C++ 11 or later
- CMake 3.1 or later
- One of the following compilers:
  - GCC 4.8 or later
  - Clang 3.9 or later
  - MSVC 2015 or later

**To use the AWS IoT Device SDK for C++ v2**

1. Download the [AWS IoT Device SDK for C++ v2](https://github.com/awslabs/aws-iot-device-sdk-cpp) (v1.13.0 or later).
2. Follow the [installation instructions in the README](https://github.com/awslabs/aws-iot-device-sdk-cpp/blob/master/README.md) to build the AWS IoT Device SDK for C++ v2 from source.
3. In your C++ build tool, link the Greengrass IPC library, `AWS::GreengrassIpc-cpp`, that you built in the previous step. The following `CMakeLists.txt` example links the Greengrass IPC library to a project that you build with CMake.

```makefile
# CMakeLists.txt example
cmake_minimum_required(VERSION 3.1)
project (greengrassv2_pubsub_subscriber)

file(GLOB MAIN_SRC
    "*.h"
    "*.cpp"
)
add_executable(${PROJECT_NAME} ${MAIN_SRC})
```
set_target_properties(${PROJECT_NAME} PROPERTIES LINKER_LANGUAGE CXX CXX_STANDARD 11)
find_package(aws-crt-cpp PATHS ~/sdk-cpp-workspace/build)
find_package(EventstreamRpc-cpp PATHS ~/sdk-cpp-workspace/build)
find_package(GreengrassIpc-cpp PATHS ~/sdk-cpp-workspace/build)
target_link_libraries(${PROJECT_NAME} AWS::GreengrassIpc-cpp)

4. In your component code, create a connection to the AWS IoT Greengrass Core IPC service to create an IPC client (Aws::Greengrass::GreengrassCoreIpcClient). You must define an IPC connection lifecycle handler that handles IPC connection, disconnection, and error events. The following example creates an IPC client and an IPC connection lifecycle handler that prints when the IPC client connects, disconnects, and encounters errors.

```cpp
#include <iostream>
#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
    }

    void OnDisconnectCallback(RpcError error) override {
        std::cout << "OnDisconnectCallback: " << error.StatusToString() << std::endl;
        exit(-1);
    }

    bool OnErrorCallback(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
};

int main() {
    // Create the IPC client.
    ApiHandle apiHandle(g Allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " << connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    // Use the IPC client to create an operation request.
    // Activate the operation request.
    auto activate = operation.Activate(request, nullptr);
    activate.wait();

    // Wait for Greengrass Core to respond to the request.
    auto responseFuture = operation.GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) == std::future_status::timeout) {
```
std::cerr << "Operation timed out while waiting for response from Greengrass Core." << std::endl;
exit(-1);
}

// Check the result of the request.
auto response = responseFuture.get();
if (response) {
    std::cout << "Successfully published to topic: " << topic << std::endl;
} else {
    // An error occurred.
    std::cout << "Failed to publish to topic: " << topic << std::endl;
    auto errorType = response.GetResultType();
    if (errorType == OPERATION_ERROR) {
        auto *error = response.GetOperationError();
        std::cout << "Operation error: " << error->GetMessage().value() << std::endl;
    } else {
        std::cout << "RPC error: " << response.GetRpcError() << std::endl;
    }
exit(-1);
}
return 0;

5. To run your custom code in your component, build your code as a binary artifact, and run the binary artifact in your component recipe. Set the artifact's Execute permission to OWNER to enable the AWS IoT Greengrass Core software to run the binary artifact.

Your component recipe's Manifests section might look similar to the following example.

**JSON**

```json
{
...
"Manifests": [
{
    "Lifecycle": {
        "Run": "{artifacts:path}/greengrassv2_pubsub_subscriber"
    },
    "Artifacts": [
        {
            "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.PubSubSubscriberCpp/1.0.0/greengrassv2_pubsub_subscriber",
            "Permission": {
                "Execute": "OWNER"
            }
        }
    ]
}
]
```

**YAML**

```yaml
...
Manifests:
  - Lifecycle:
    Run: {artifacts:path}/greengrassv2_pubsub_subscriber
  Artifacts:
    - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.PubSubSubscriberCpp/1.0.0/greengrassv2_pubsub_subscriber
```
Authorize components to perform IPC operations

To allow your custom components to use some IPC operations, you must define *authorization policies* that allow the component to perform the operation on certain resources. Each authorization policy defines a list of operations and a list of resources that the policy allows. For example, the publish/subscribe messaging IPC service defines publish and subscribe operations for topic resources. You can use the * wildcard to allow access to all operations or all resources.

You define authorization policies in the component recipe with the `accessControl` configuration parameter. The `accessControl` object maps IPC service identifiers to lists of authorization policies. You can define multiple authorization policies for each IPC service to control access. Each authorization policy has a policy ID, which must be unique among all components.

**Tip**
To create unique policy IDs, you can combine the component name, IPC service name, and a counter. For example, a component named `com.example.HelloWorld` might define two publish/subscribe authorization policies with the following IDs:

- `com.example.HelloWorld:pubsub:1`
- `com.example.HelloWorld:pubsub:2`

Authorization policies use the following format. This object is the `accessControl` configuration parameter.

**JSON**

```json
{
  "IPC service identifier": {
    "policyId": {
      "policyDescription": "description",
      "operations": [
        "operation1",
        "operation2"
      ],
      "resources": [
        "resource1",
        "resource2"
      ]
    }
  }
}
```

**YAML**

```
IPC service identifier:
  policyId:
    policyDescription: description
    operations:
      - operation1
      - operation2
    resources:
      - resource1
      - resource2
```
Authorize components to perform IPC operations

Example component recipe with an authorization policy

The following example component recipe includes an accessControl object defines an authorization policy. This policy authorizes the com.example.HelloWorld component to publish to the test/topic topic.

**JSON**

```json
{
    "RecipeFormatVersion": "2020-01-25",
    "ComponentName": "com.example.HelloWorld",
    "ComponentVersion": "1.0.0",
    "ComponentDescription": "A component that publishes messages.",
    "ComponentPublisher": "Amazon",
    "ComponentConfiguration": {
        "DefaultConfiguration": {
            "accessControl": {
                "aws.greengrass.ipc.pubsub": {
                    "com.example.HelloWorld:pubsub:1": {
                        "policyDescription": "Allows access to publish to test/topic.",
                        "operations": [
                            "aws.greengrass#PublishToTopic"
                        ],
                        "resources": [
                            "test/topic"
                        ]
                    }
                }
            }
        },
        "Manifests": [
            {
                "Lifecycle": {
                    "Run": "java -Dlog.level=INFO -jar {artifacts:path}/HelloWorld.jar"
                }
            }
        ]
    }
}
```

**YAML**

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: A component that publishes messages.
ComponentPublisher: Amazon
ComponentConfiguration: 
    DefaultConfiguration: 
        accessControl: 
            aws.greengrass.ipc.pubsub:
                "com.example.HelloWorld:pubsub:1": 
                    policyDescription: Allows access to publish to test/topic.
                    operations: 
                        - "aws.greengrass#PublishToTopic"
                    resources: 
                        - "test/topic"
    Manifests: 
        - Lifecycle: 
            Run: |
                java -Dlog.level=INFO -jar {artifacts:path}/HelloWorld.jar
```
Subscribe to IPC event streams

You can use IPC operations to subscribe to streams of events on a Greengrass core device. To use a subscribe operation, define a *subscription handler* and create a request to the IPC service. Then, the IPC client runs the subscription handler's functions each time that the core device streams an event message to your component.

You can close a subscription to stop processing event messages. To do so, call `closeStream()` (Java), `close()` (Python), or `Close()` (C++) on the subscription operation object that you used to open the subscription.

The AWS IoT Greengrass Core IPC service supports the following subscribe operations:

- SubscribeToTopic (p. 414)
- SubscribeToIoTCore (p. 436)
- SubscribeToComponentUpdates (p. 448)
- SubscribeToConfigurationUpdate (p. 453)
- SubscribeToValidateConfigurationUpdates (p. 454)

Topics

- Define subscription handlers (p. 405)
- Best practices for subscription handlers (p. 406)
- Example subscription handlers (p. 407)

Define subscription handlers

To define a subscription handler, create a class with callback functions that handle event messages, errors, and stream closure.

Java

Implement the generic `software.amazon.awssdk.eventstreamrpc.StreamResponseHandler<StreamEventType>` interface. `StreamEventType` is the type of event message for the subscription operation. Define the following functions to handle event messages, errors, and stream closure.

```java
void onStreamEvent(StreamEventType event)
    The callback that the IPC client calls when it receives an event message, such as an MQTT message or a component update notification.

boolean onStreamError(Throwable error)
    The callback that the IPC client calls when a stream error occurs.
    Return true to close the subscription stream as a result of the error, or return false to keep the stream open.

void onStreamClosed()
    The callback that the IPC client calls when the stream closes.
```

Python

Extend the stream response handler class that corresponds to the subscription operation. The AWS IoT Device SDK includes a subscription handler class for each subscription operation.

```python
Extend the stream response handler class that corresponds to the subscription operation.
```
StreamEventType is the type of event message for the subscription operation. Define the following functions to handle event messages, errors, and stream closure.

```python
    def on_stream_event(self, event: StreamEventType) -> None
        The callback that the IPC client calls when it receives an event message, such as an MQTT message or a component update notification.

    def on_stream_error(self, error: Exception) -> bool
        The callback that the IPC client calls when a stream error occurs.
        Return true to close the subscription stream as a result of the error, or return false to keep the stream open.

    def on_stream_closed(self) -> None
        The callback that the IPC client calls when the stream closes.
```

C++

Implement a class that derives from the stream response handler class that corresponds to the subscription operation. The AWS IoT Device SDK includes a subscription handler base class for each subscription operation. StreamEventType is the type of event message for the subscription operation. Define the following functions to handle event messages, errors, and stream closure.

```cpp
    void OnStreamEvent(StreamEventType *event)
        The callback that the IPC client calls when it receives an event message, such as an MQTT message or a component update notification.

    bool OnStreamError(OperationError *error)
        The callback that the IPC client calls when a stream error occurs.
        Return true to close the subscription stream as a result of the error, or return false to keep the stream open.

    void OnStreamClosed()
        The callback that the IPC client calls when the stream closes.
```

**Best practices for subscription handlers**

The IPC client uses a single thread that communicates with the IPC server and calls your subscription handler. You must consider this synchronous behavior when you write subscription handler functions. Follow these guidelines when you write subscription handler functions.

- **Run blocking code asynchronously**

  The IPC client can't send new requests or process new event messages while the thread is blocked. You can run blocking code in a separate thread that you run from the handler function. Blocking code includes `sleep` calls, loops that continuously run, and synchronous I/O requests that take time to complete.

- **Send new IPC requests asynchronously**

  The IPC client can't send a new request from within subscription handler functions, because the request blocks the handler function if you wait for a response. You can send IPC requests in a separate thread that you run from the handler function.
Example subscription handlers

The following example demonstrates how to use the SubscribeToTopic (p. 414) operation and a subscription handler to subscribe to local publish/subscribe messages.

Java

**Example: Subscribe to local publish/subscribe messages**

```java
String topic = "my/topic";
SubscribeToTopicRequest subscribeToTopicRequest = new SubscribeToTopicRequest();
subscribeToTopicRequest.setTopic(topic);
StreamResponseHandler<SubscriptionResponseMessage> streamResponseHandler = new StreamResponseHandler<SubscriptionResponseMessage>() {
    @Override
    public void onStreamEvent(SubscriptionResponseMessage subscriptionResponseMessage) {
        try {
            String message = new String(subscriptionResponseMessage.getBinaryMessage().getMessage(), StandardCharsets.UTF_8);
            // Handle message.
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    @Override
    public boolean onStreamError(Throwable error) {
        // Handle error.
        return false; // Return true to close stream, false to keep stream open.
    }
    @Override
    public void onStreamClosed() {
        // Handle close.
    }
};
SubscribeToTopicResponseHandler operationResponseHandler = greengrassCoreIPCCClient.subscribeToTopic(subscribeToTopicRequest, Optional.of(streamResponseHandler));
operationResponseHandler.getResponse().get();
// Keep the main thread alive, or the process will exit.
try {
    while (true) {
        Thread.sleep(10000);
    }
} catch (InterruptedException e) {
    System.out.println("Subscribe interrupted.");
}
```
/ To stop subscribing, close the stream.
operationResponseHandler.closeStream();

Example Example: Subscribe to local publish/subscribe messages

Note
This example assumes that you are using version 1.5.4 or later of the AWS IoT Device SDK for Python v2. If you are using version 1.5.3 of the SDK, see Use AWS IoT Device SDK for Python v2 (p. 399) for information about connecting to the AWS IoT Greengrass Core IPC service.

```python
import time
import traceback
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import (
    SubscribeToTopicRequest,
    SubscriptionResponseMessage
)

TIMEOUT = 10

ipc_client = awsiot.greengrasscoreipc.connect()

class StreamHandler(client.SubscribeToTopicStreamHandler):
    def __init__(self):
        super().__init__()

    def on_stream_event(self, event: SubscriptionResponseMessage) -> None:
        try:
            message_string = str(event.binary_message.message, "utf-8")
            # Handle message.
        except:
            traceback.print_exc()

    def on_stream_error(self, error: Exception) -> bool:
        # Handle error.
        return True  # Return True to close stream, False to keep stream open.

    def on_stream_closed(self) -> None:
        # Handle close.
        pass

topic = "my/topic"

request = SubscribeToTopicRequest()
request.topic = topic
handler = StreamHandler()
operation = ipc_client.new_subscribe_to_topic(handler)
future = operation.activate(request)
future.result(TIMEOUT)

# Keep the main thread alive, or the process will exit.
while True:
    time.sleep(10)

# To stop subscribing, close the operation stream.
operation.close()
```
Example Example: Subscribe to local publish/subscribe messages

```cpp
#include <iostream>
#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class SubscribeResponseHandler : public SubscribeToTopicStreamHandler {
    void OnStreamEvent(SubscriptionResponseMessage *response) override {
        auto jsonMessage = response->GetJsonMessage();
        if (jsonMessage.has_value() && jsonMessage.value().GetMessage().has_value()) {
            auto messageString = jsonMessage.value().GetMessage().value().View().WriteReadable();
            // Handle JSON message.
        } else {
            auto binaryMessage = response->GetBinaryMessage();
            if (binaryMessage.has_value() &
                binaryMessage.value().GetMessage().has_value()) {
                auto messageBytes = binaryMessage.value().GetMessage().value();
                std::string messageString(messageBytes.begin(), messageBytes.end());
                // Handle binary message.
            }
        }
    }

    bool OnStreamError(OperationError *error) override {
        // Handle error.
        return false; // Return true to close stream, false to keep stream open.
    }

    void OnStreamClosed() override {
        // Handle close.
    }
};

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }

    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }

    bool OnErrorCallback(RpcError error) override {
        // Handle IPC service connection error.
        return true;
    }
};

int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
```
std::cerr << "Failed to establish IPC connection: " <<
  connectionStatus.StatusToString() << std::endl;
exit(-1);
}

String topic("my/topic");
int timeout = 10;

SubscribeToTopicRequest request;
request.SetTopic(topic);

SubscribeResponseHandler streamHandler;
SubscribeToTopicOperation operation = ipcClient.NewSubscribeToTopic(streamHandler);
auto activate = operation.Activate(request, nullptr);
activate.wait();

auto responseFuture = operation.GetResult();
if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
  std::future_status::timeout) {
  std::cerr << "Operation timed out while waiting for response from Greengrass
Core." << std::endl;
  exit(-1);
}

if (!response) {
  // Handle error.
  auto errorType = response.GetResultType();
  if (errorType == OPERATION_ERROR) {
    auto *error = response.GetOperationError();
    // Handle operation error.
  } else {
    // Handle RPC error.
  }
  exit(-1);
}

// Keep the main thread alive, or the process will exit.
while (true) {
  std::this_thread::sleep_for(std::chrono::seconds(10));
}

operation.Close();
return 0;

---

Publish/subscribe local messages

Publish/subscribe (pubsub) messaging enables you to send and receive messages to topics. Components can publish messages to topics to send messages to other components. Then, components that are subscribed to that topic can act on the messages that they receive.

**Note**

You can't use this publish/subscribe IPC service to publish or subscribe to AWS IoT Core MQTT. For more information about how to exchange messages with AWS IoT Core MQTT, see the Publish/subscribe AWS IoT Core MQTT messages (p. 433).

**Topics**

- [Authorization](p. 411)
- [PublishToTopic](p. 411)
- [SubscribeToTopic](p. 414)
Authorization

To use local publish/subscribe messaging in a custom component, you must define authorization policies that allows your component to send and receive messages to topics. For information about defining authorization policies, see Authorize components to perform IPC operations (p. 403).

Authorization policies for publish/subscribe messaging have the following properties.

**IPC service identifier**: `aws.greengrass.ipc.pubsub`

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aws.greengrass#PublishToTopic</code></td>
<td>Allows a component to publish messages to the topics that you specify.</td>
<td>A topic string, such as test/topic, or * to allow access to all topics. This topic string doesn't support MQTT topic wildcards (# and +).</td>
</tr>
<tr>
<td><code>aws.greengrass#SubscribeToTopic</code></td>
<td>Allows a component to subscribe to messages for the topics that you specify.</td>
<td>A topic string, such as test/topic, or * to allow access to all topics. This topic string doesn't support MQTT topic wildcards (# and +).</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Allows a component to publish and subscribe to messages for the topics that you specify.</td>
<td>A topic string, such as test/topic, or * to allow access to all topics. This topic string doesn't support MQTT topic wildcards (# and +).</td>
</tr>
</tbody>
</table>

**Example authorization policy**

The following example authorization policy allows a component to publish and subscribe to all topics.

```
{
  "accessControl": {
    "aws.greengrass.ipc.pubsub": {
      "com.example.MyLocalPubSubComponent:pubsub:1": {
        "policyDescription": "Allows access to publish/subscribe to all topics.",
        "operations": [
          "aws.greengrass#PublishToTopic",
          "aws.greengrass#SubscribeToTopic"
        ],
        "resources": [
          "*
        ]
      }
    }
  }
}
```

**PublishToTopic**

Publish a message to a topic.
Request

This operation's request has the following parameters:

**topic**

The topic to which to publish the message.

**publishMessage (Python: publish_message)**

The message to publish. This object, PublishMessage, contains the following information. You must specify one of `jsonMessage` and `binaryMessage`.

**jsonMessage (Python: json_message)**

(Optional) A JSON message. This object, JsonMessage, contains the following information:

**message**

The JSON message as an object.

**binaryMessage (Python: binary_message)**

(Optional) A binary message. This object, BinaryMessage, contains the following information:

**message**

The binary message as a blob.

This operation doesn't provide any information in its response.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java

**Example Example: Publish a binary message**

```java
String topic = "my/topic";
String message = "Hello, World!";

PublishToTopicRequest publishToTopicRequest = new PublishToTopicRequest();
publishMessage publishMessage = new PublishMessage();
BinaryMessage binaryMessage = new BinaryMessage();
binaryMessage.setMessage(message.getBytes(StandardCharsets.UTF_8));
publishMessage.setBinaryMessage(binaryMessage);
publishToTopicRequest.setPublishMessage(publishMessage);
publishToTopicRequest.setTopic(topic);
greengrassCoreIPCClient.publishToTopic(publishToTopicRequest, Optional.empty()).getResponse().get();
```

Python

**Example Example: Publish a binary message**

```python
import awsiot.greengrasscoreipc
def setup():
    client = awsiot.greengrasscoreipc.connect()
    topic = "my/topic"
    message = "Hello, World!"

    request = awsiot.greengrasscoreipc.model.PublishToTopicRequest()
    request.topic = topic
    request.publish_message = awsiot.greengrasscoreipc.model.PublishMessage()
    request.publish_message.binary_message = awsiot.greengrasscoreipc.model.BinaryMessage()
    request.publish_message.binary_message.message = message.encode()

    response = client.publish_to_topic(request)

    # Do something with the response
```

**Note**

This example assumes that you are using version 1.5.4 or later of the AWS IoT Device SDK for Python v2. If you are using version 1.5.3 of the SDK, see Use AWS IoT Device SDK for Python v2 (p. 399) for information about connecting to the AWS IoT Greengrass Core IPC service.
PublishToTopicRequest, PublishMessage, BinaryMessage

TIMEOUT = 10

ipc_client = awsiot.greengrasscoreipc.connect()

topic = "my/topic"
message = "Hello, World!"

request = PublishToTopicRequest()
request.topic = topic
publish_message = PublishMessage()
publish_message.binary_message = BinaryMessage()
publish_message.binary_message.message = bytes(message, "utf-8")
request.publish_message = publish_message
operation = ipc_client.new_publish_to_topic()
operation.activate(request)
future = operation.get_response()
future.result(TIMEOUT)

C++

Example Example: Publish a binary message

#include <iostream>
#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
public:
  void OnConnectCallback() override {
    // Handle connection to IPC service.
  }

  void OnDisconnectCallback(RpcError error) override {
    // Handle disconnection from IPC service.
  }

  bool OnErrorCallback(RpcError error) override {
    // Handle IPC service connection error.
    return true;
  }
};

int main() {
  ApiHandle apiHandle(g_allocator);
  Io::EventLoopGroup eventLoopGroup(1);
  Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
  Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
  IpcClientLifecycleHandler ipcLifecycleHandler;
  GreengrassCoreIpcClient ipcClient(bootstrap);
  auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
  if (!connectionStatus) {
    std::cerr << "Failed to establish IPC connection: " <<
              connectionStatus.StatusToString() << std::endl;
    exit(-1);
  }

  String topic("my/topic");
String message("Hello, World!");
int timeout = 10;

PublishToTopicRequest request;
Vector<uint8_t> messageData({message.begin(), message.end()});
BinaryMessage binaryMessage;
binaryMessage.SetMessage(messageData);
PublishMessage publishMessage;
publishMessage.SetBinaryMessage(binaryMessage);
request.SetTopic(topic);
request.SetPublishMessage(publishMessage);

PublishToTopicOperation operation = ipcClient.NewPublishToTopic();
auto activate = operation.Activate(request, nullptr);
activate.wait();

auto responseFuture = operation.GetResult();
if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
std::future_status::timeout) {
    std::cerr << "Operation timed out while waiting for response from Greengrass
Core." << std::endl;
    exit(-1);
}
if (!response) {
    // Handle error.
    auto errorType = response.GetResultType();
    if (errorType == OPERATION_ERROR) {
        auto *error = response.GetOperationError();
        // Handle operation error.
    } else {
        // Handle RPC error.
    }
}
return 0;

---

**SubscribeToTopic**

Subscribe to messages at a topic.

This operation is a subscription operation where you subscribe to a stream of event messages. To use this operation, define a stream response handler with functions that handle event messages, errors, and stream closure. For more information, see [Subscribe to IPC event streams](p. 405).

**Event message type:** SubscriptionResponseMessage

**Request**

This operation's request has the following parameters:

- **topic**
  The topic to which to subscribe.

  **Note**
  This topic doesn't support MQTT topic wildcards (# and +).

**Response**

This operation's response has the following information:
topicName (Python: topic_name)

The topic to which the message was published.

messages

The stream of messages. This object, SubscriptionResponseMessage, contains the following information. Each message contains jsonMessage or binaryMessage.

jsonMessage (Python: json_message)

(Optional) A JSON message. This object, JsonMessage, contains the following information:

message

The JSON message as an object.

binaryMessage (Python: binary_message)

(Optional) A binary message. This object, BinaryMessage, contains the following information:

message

The binary message as a blob.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java

Example Example: Subscribe to local publish/subscribe messages

```java
String topic = "my/topic";

SubscribeToTopicRequest subscribeToTopicRequest = new SubscribeToTopicRequest();
subscribeToTopicRequest.setTopic(topic);

StreamResponseHandler<SubscriptionResponseMessage> streamResponseHandler =
new StreamResponseHandler<SubscriptionResponseMessage>() {
    @Override
    public void onStreamEvent(SubscriptionResponseMessage subscriptionResponseMessage) {
        try {
            String message = new String(subscriptionResponseMessage.getBinaryMessage().getMessage(), StandardCharsets.UTF_8);
            // Handle message.
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    public boolean onStreamError(Throwable error) {
        // Handle error.
        return false; // Return true to close stream, false to keep stream open.
    }

    @Override
    public void onStreamClosed() {
        // Handle close.
    }
};
```
SubscribeToTopicResponseHandler operationResponseHandler = greengrassCoreIPCClient
.subscribeToTopic(subscribeToTopicRequest, Optional.of(streamResponseHandler));
operationResponseHandler.getResponse().get();

// Keep the main thread alive, or the process will exit.
try {
    while (true) {
        Thread.sleep(10000);
    }
} catch (InterruptedException e) {
    System.out.println("Subscribe interrupted.");
}

// To stop subscribing, close the stream.
operationResponseHandler.closeStream();

Python

Example Example: Subscribe to local publish/subscribe messages

Note
This example assumes that you are using version 1.5.4 or later of the AWS IoT Device SDK for Python v2. If you are using version 1.5.3 of the SDK, see Use AWS IoT Device SDK for Python v2 (p. 399) for information about connecting to the AWS IoT Greengrass Core IPC service.

import time
import traceback
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import (
    SubscribeToTopicRequest,
    SubscriptionResponseMessage
)

TIMEOUT = 10

ipc_client = awsiot.greengrasscoreipc.connect()

class StreamHandler(client.SubscribeToTopicStreamHandler):
    def __init__(self):
        super().__init__()

    def on_stream_event(self, event: SubscriptionResponseMessage) -> None:
        try:
            message_string = str(event.binary_message.message, "utf-8")
            # Handle message.
        except:
            traceback.print_exc()

    def on_stream_error(self, error: Exception) -> bool:
        # Handle error.
        return True  # Return True to close stream, False to keep stream open.

    def on_stream_closed(self) -> None:
        # Handle close.
        pass

topic = "my/topic"

request = SubscribeToTopicRequest()
request.topic = topic
handler = StreamHandler()
operation = ipc_client.new_subscribe_to_topic(handler)
future = operation.activate(request)
future.result(TIMEOUT)

# Keep the main thread alive, or the process will exit.
while True:
    time.sleep(10)

# To stop subscribing, close the operation stream.
operation.close()

### C++

**Example Example: Subscribe to local publish/subscribe messages**

```cpp
#include <iostream>
#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class SubscribeResponseHandler : public SubscribeToTopicStreamHandler {
    void OnStreamEvent(SubscriptionResponseMessage *response) override {
        auto jsonMessage = response->GetJsonMessage();
        if (jsonMessage.has_value() && jsonMessage.value().GetMessage().has_value()) {
            auto messageString = jsonMessage.value().GetMessage().value().View().WriteReadable();
            // Handle JSON message.
        } else {
            auto binaryMessage = response->GetBinaryMessage();
            if (binaryMessage.has_value() && binaryMessage.value().GetMessage().has_value()) {
                auto messageBytes = binaryMessage.value().GetMessage().value();
                std::string messageString(messageBytes.begin(), messageBytes.end());
                // Handle binary message.
            }
        }
    }

    bool OnStreamError(OperationError *error) override {
        // Handle error.
        return false; // Return true to close stream, false to keep stream open.
    }

    void OnStreamClosed() override {
        // Handle close.
    }
};

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }

    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }

    bool OnErrorCallback(RpcError error) override {
        // Handle IPC service connection error.
        return true;
    }
};
```
int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect().get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
                  connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    String topic("my/topic");
    int timeout = 10;

    SubscribeToTopicRequest request;
    request.SetTopic(topic);

    SubscribeResponseHandler streamHandler;
    SubscribeToTopicOperation operation = ipcClient.NewSubscribeToTopic(streamHandler);
    auto activate = operation.Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation.GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
        std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
                  Core." << std::endl;
        exit(-1);
    }

    if (!response) {
        // Handle error.
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            // Handle operation error.
        } else {
            // Handle RPC error.
        }
        exit(-1);
    }

    // Keep the main thread alive, or the process will exit.
    while (true) {
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    operation.Close();
    return 0;
}

Examples

Use the following examples to learn how to use the publish/subscribe IPC service in your components.

Example publish/subscribe publisher (Java)

The following example recipe allows the component to publish to all topics.
The following example Java application demonstrates how to use the publish/subscribe IPC service to publish messages to other components.

```java
/* Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved. */
* SPDX-License-Identifier: Apache-2.0 */
```
package com.example.ipc.pubsub;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientAsync;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientBuilder;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientConfiguration;
import software.amazon.awssdk.aws.greengrass.model.PublishToTopicRequest;
import software.amazon.awssdk.aws.greengrass.model.PublishToTopicResponse;
import software.amazon.awssdk.aws.greengrass.model.PublishMessage;
import software.amazon.awssdk.aws.greengrass.model.BinaryMessage;
import software.amazon.awssdk.aws.greengrass.model.MessageType;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;
import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class PubSubPublisher {
    public static void main(String[] args) {
        String message = "Hello from the pub/sub publisher (Java).";
        String topic = "test/topic/java";

        try (EventStreamRPCConnection eventStreamRPCConnection =
                IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient = new
                    GreengrassCoreIPCClient(eventStreamRPCConnection);
            while (true) {
                PublishToTopicRequest publishRequest = new PublishToTopicRequest();
                PublishMessage publishMessage = new PublishMessage();
                BinaryMessage binaryMessage = new BinaryMessage();
                binaryMessage.setMessage(message.getBytes(StandardCharsets.UTF_8));
                publishMessage.setBinaryMessage(binaryMessage);
                publishRequest.setPublishMessage(publishMessage);
                publishRequest.setTopic(topic);
                CompletableFuture<PublishToTopicResponse> futureResponse = ipcClient
                        .publishToTopic(publishRequest, Optional.empty()).getResponse();
                try {
                    futureResponse.get(10, TimeUnit.SECONDS);
                    System.out.println("Successfully published to topic: " + topic);
                } catch (TimeoutException e) {
                    System.err.println("Timeout occurred while publishing to topic: "
                            + topic);
                } catch (ExecutionException e) {
                    if (e.getCause() instanceof UnauthorizedError) {
                        System.err.println("Unauthorized error while publishing to topic: "
                                + topic);
                    } else {
                        System.err.println("Execution exception while publishing to topic:
                                " + topic);
                    }
                    throw e;
                }
                Thread.sleep(5000);
            }
        } catch (InterruptedException e) {
            System.out.println("Publisher interrupted.");
        } catch (Exception e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }
}
Example publish/subscribe subscriber (Java)

The following example recipe allows the component to subscribe to all topics.

**JSON**

```json
{
    "RecipeFormatVersion": "2020-01-25",
    "ComponentName": "com.example.PubSubSubscriberJava",
    "ComponentVersion": "1.0.0",
    "ComponentDescription": "A component that subscribes to messages.",
    "ComponentPublisher": "Amazon",
    "ComponentConfiguration": {
        "accessControl": {
            "aws.greengrass.ipc.pubsub": {
                "com.example.PubSubSubscriberJava:pubsub:1": {
                    "policyDescription": "Allows access to subscribe to all topics.",
                    "operations": [
                        "aws.greengrass#SubscribeToTopic"
                    ],
                    "resources": [
                        "*"
                    ]
                }
            }
        }
    },
    "Manifests": [
        {
            "Lifecycle": {
                "Run": "java -jar {artifacts:path}/PubSubSubscriber.jar"
            }
        }
    ]
}
```

**YAML**

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubSubscriberJava
ComponentVersion: '1.0.0'
ComponentDescription: A component that subscribes to messages.
ComponentPublisher: Amazon
ComponentConfiguration:
    DefaultConfiguration:
        accessControl:
            aws.greengrass.ipc.pubsub:
                com.example.PubSubSubscriberJava:pubsub:1':
                    policyDescription: Allows access to subscribe to all topics.
                    operations:
                        - 'aws.greengrass#SubscribeToTopic'
                    resources:
                        - '*'
Manifests:
    - Lifecycle:
        Run: |
            java -jar {artifacts:path}/PubSubSubscriber.jar
```

421
The following example Java application demonstrates how to use the publish/subscribe IPC service to subscribe to messages to other components.

```java
/* Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved. * SPDX-License-Identifier: Apache-2.0 */
package com.example.ipc.pubsub;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.SubscribeToTopicResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.SubscribeToTopicRequest;
import software.amazon.awssdk.aws.greengrass.model.SubscribeToTopicResponse;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;
import software.amazon.awssdk.eventstreamrpc.StreamResponseHandler;
import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class PubSubSubscriber {
    public static void main(String[] args) {
        String topic = "test/topic/java";

        try (EventStreamRPCConnection eventStreamRPCConnection = IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient = new GreengrassCoreIPCClient(eventStreamRPCConnection);

            SubscribeToTopicRequest subscribeRequest = new SubscribeToTopicRequest();
            subscribeRequest.setTopic(topic);
            SubscribeToTopicResponseHandler operationResponseHandler = ipcClient.subscribeToTopic(subscribeRequest, Optional.of(new SubscribeResponseHandler()));

            CompletableFuture<SubscribeToTopicResponse> futureResponse = operationResponseHandler.getResponse();

            try {
                futureResponse.get(10, TimeUnit.SECONDS);
                System.out.println("Successfully subscribed to topic: " + topic);
            } catch (TimeoutException e) {
                System.err.println("Timeout occurred while subscribing to topic: " + topic);
                throw e;
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.println("Unauthorized error while subscribing to topic: " + topic);
                } else {
                    System.err.println("Execution exception while subscribing to topic: " + topic);
                }
                throw e;
            }

            // Keep the main thread alive, or the process will exit.
            try {
                while (true) {
                    Thread.sleep(10000);
                }
            }
        }
    }
}
```
```java
private static class SubscribeResponseHandler implements StreamResponseHandler<SubscriptionResponseMessage> {
    @Override
    public void onStreamEvent(SubscriptionResponseMessage subscriptionResponseMessage) {
        try {
            String message = new String(subscriptionResponseMessage.getBinaryMessage().
                    getMessage(), StandardCharsets.UTF_8);
            System.out.println("Received new message: " + message);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    public boolean onStreamError(Throwable error) {
        System.err.println("Received a stream error.");
        error.printStackTrace();
        return false; // Return true to close stream, false to keep stream open.
    }

    @Override
    public void onStreamClosed() {
        System.out.println("Subscribe to topic stream closed.");
    }
}
```

Example publish/subscribe publisher (Python)

The following example recipe allows the component to publish to all topics.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubPublisherPython",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubPublisherPython:pubsub:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [ "*" ]
        }
      }
    }
  }
}
```
The following example Python application demonstrates how to use the publish/subscribe IPC service to publish messages to other components.

```python
import concurrent.futures
import sys
import time
import traceback
import awsiot.greengrasscoreipc
from awsiot.greengrasscoreipc.model import (
    PublishToTopicRequest,
    PublishMessage,
    BinaryMessage,
    UnauthorizedError
)

topic = "test/topic/python"
message = "Hello from the pub/sub publisher (Python)."
TIMEOUT = 10

try:
    ipc_client = awsiot.greengrasscoreipc.connect()
    while True:
```
Example publish/subscribe subscriber (Python)

The following example recipe allows the component to subscribe to all topics.

JSON

```json
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubSubscriberPython",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubSubscriberPython:pubsub:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "Install": "python3 -m pip install --user awsiotsdk",
        "Run": "python3 -u {artifacts:path}/pubsub_subscriber.py"
      }
    }
  ]
}
```
The following example Python application demonstrates how to use the publish/subscribe IPC service to subscribe to messages to other components.

```python
import concurrent.futures
import sys
import time
import traceback
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import (SubscribeToTopicRequest, SubscriptionResponseMessage, UnauthorizedError)

topic = "test/topic/python"
TIMEOUT = 10

class StreamHandler(client.SubscribeToTopicStreamHandler):
    def __init__(self):
        super().__init__()

    def on_stream_event(self, event: SubscriptionResponseMessage) -> None:
        try:
            message = str(event.binary_message.message, "utf-8")
            print("Received new message: " + message)
        except:
            traceback.print_exc()

    def on_stream_error(self, error: Exception) -> bool:
        print("Received a stream error.", file=sys.stderr)
        traceback.print_exc()
```

YAML

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubSubscriberPython
ComponentVersion: 1.0.0
ComponentDescription: A component that subscribes to messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        com.example.PubSubSubscriberPython:pubsub:1:
          policyDescription: Allows access to subscribe to all topics.
          operations:
            - aws.greengrass#SubscribeToTopic
          resources:
            - "*"

Manifests:
- Lifecycle:
  Install: python3 -m pip install --user awsiotsdk
  Run: python3 -u {artifacts:path}/pubsub_subscriber.py
```
def on_stream_closed(self) -> None:
    print('Subscribe to topic stream closed.')

try:
    ipc_client = awsiot.greengrasscoreipc.connect()
    request = SubscribeToTopicRequest()
    request.topic = topic
    handler = StreamHandler()
    operation = ipc_client.new_subscribe_to_topic(handler)
    future = operation.activate(request)

    try:
        future.result(TIMEOUT)
        print('Successfully subscribed to topic: ' + topic)
    except concurrent.futures.TimeoutError as e:
        print('Timeout occurred while subscribing to topic: ' + topic, file=sys.stderr)
        raise e
    except UnauthorizedError as e:
        print('Unauthorized error while subscribing to topic: ' + topic, file=sys.stderr)
        raise e
    except Exception as e:
        print('Exception while subscribing to topic: ' + topic, file=sys.stderr)
        raise e

    # Keep the main thread alive, or the process will exit.
    try:
        while True:
            time.sleep(10)
    except InterruptedError:
        print('Subscribe interrupted.')
except Exception:
    print('Exception occurred when using IPC.', file=sys.stderr)
    traceback.print_exc()
    exit(1)

Example publish/subscribe publisher (C++)

The following example recipe allows the component to publish to all topics.

JSON

```json
{
    "RecipeFormatVersion": "2020-01-25",
    "ComponentName": "com.example.PubSubPublisherCpp",
    "ComponentVersion": "1.0.0",
    "ComponentDescription": "A component that publishes messages.",
    "ComponentPublisher": "Amazon",
    "ComponentConfiguration": {
        "DefaultConfiguration": {
            "accessControl": {
                "aws.greengrass.ipc.pubsub": {
                    "com.example.PubSubPublisherCpp:pubsub:1": {
                        "policyDescription": "Allows access to publish to all topics.",
                        "operations": [
                            "aws.greengrass#PublishToTopic"
                        ],
                        "resources": [
                            "*"
                        ]
                    }
                }
            }
        }
    }
}
```
The following example C++ application demonstrates how to use the publish/subscribe IPC service to publish messages to other components.

```cpp
#include <iostream>
#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {

    void OnConnectCallback() override {
```

YAML

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubPublisherCpp
ComponentVersion: 1.0.0
ComponentDescription: A component that publishes messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        com.example.PubSubPublisherCpp:pubsub:1:
          policyDescription: Allows access to publish to all topics.
          operations:
            - aws.greengrass#PublishToTopic
          resources:
            - "*"
  Manifests:
    - Lifecycle:
      Run: '{artifacts:path}/greengrassv2_pubsub_publisher'
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.PubSubPublisherCpp/1.0.0/greengrassv2_pubsub_publisher
        Permission:
          Execute: OWNER
```
std::cout << "OnConnectCallback" << std::endl;
}

void OnDisconnectCallback(RpcError error) override {
    std::cout << "OnDisconnectCallback: " << error.StatusToString() << std::endl;
    exit(-1);
}

bool OnErrorCallback(RpcError error) override {
    std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
    return true;
}
};

int main() {
    String message("Hello from the pub/sub publisher (C++)." roomId);
    String topic("test/topic/cpp");
    int timeout = 10;

    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " << connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    while (true) {
        PublishToTopicRequest request;
        Vector<uint8_t> messageData({message.begin(), message.end()});
        BinaryMessage binaryMessage;
        binaryMessage.SetMessage(messageData);
        PublishMessage publishMessage;
        publishMessage.SetBinaryMessage(binaryMessage);
        request.SetTopic(topic);
        request.SetPublishMessage(publishMessage);

        PublishToTopicOperation operation =ipcClient.NewPublishToTopic();
        auto activate = operation.Activate(request, nullptr);
        activate.wait();

        auto responseFuture = operation.GetResult();
        if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
            std::future_status::timeout) {
            std::cerr << "Operation timed out while waiting for response from Greengrass Core." << std::endl;
            exit(-1);
        }

        auto response = responseFuture.get();
        if (response) {
            std::cout << "Successfully published to topic: " << topic << std::endl;
        } else {
            // An error occurred.
            std::cout << "Failed to publish to topic: " << topic << std::endl;
            auto errorType = response.GetResultType();
            if (errorType == OPERATION_ERROR) {
                auto *error = response.GetOperationError();
                std::cout << "Operation error: " << error->GetMessage().value() << std::endl;
            } else {

        } else {

    } // end main
} // end namespace

429
std::cout << "RPC error: " << response.GetRpcError() << std::endl;
} exit(-1);
std::this_thread::sleep_for(std::chrono::seconds(5));
return 0;

Example publish/subscribe subscriber (C++)

The following example recipe allows the component to subscribe to all topics.

JSON

```json
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubSubscriberCpp",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubSubscriberCpp:pubsub:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "*
            ]
          }
        }
      }
    },
    "Manifests": [
      {
        "Lifecycle": {
          "Run": "{artifacts:path}/greengrassv2_pub_sub_subscriber"
        },
        "Artifacts": [
          {
            "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.PubSubSubscriberCpp/1.0.0/greengrassv2_pub_sub_subscriber",
            "Permission": {
              "Execute": "OWNER"
            }
          }
        ]
      }
    ]
  }
}
```

YAML

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubSubscriberCpp
```
ComponentVersion: 1.0.0
ComponentDescription: A component that subscribes to messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        com.example.PubSubSubscriberCpp:pubsub:1:
          policyDescription: Allows access to subscribe to all topics.
          operations:
            - aws.greengrass#SubscribeToTopic
          resources:
            - "*"

Manifests:
  - Lifecycle:
    Run: "{artifacts:path}/greengrassv2_pub_sub_subscriber"
  Artifacts:
    - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.PubSubSubscriberCpp/1.0.0/greengrassv2_pub_sub_subscriber
      Permission:
        Execute: OWNER

The following example C++ application demonstrates how to use the publish/subscribe IPC service to subscribe to messages to other components.

```cpp
#include <iostream>
#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>
using namespace Aws::Crt;
using namespace Aws::Greengrass;

class SubscribeResponseHandler : public SubscribeToTopicStreamHandler {
  void OnStreamEvent(SubscriptionResponseMessage *response) override {
    auto jsonMessage = response->GetJsonMessage();
    if (jsonMessage.has_value() && jsonMessage.value().GetMessage().has_value()) {
      auto messageString = jsonMessage.value().GetMessage().value().View().WriteReadable();
      std::cout << "Received new message: " << messageString << std::endl;
    } else {
      auto binaryMessage = response->GetBinaryMessage();
      if (binaryMessage.has_value() &&
        binaryMessage.value().GetMessage().has_value()) {
        auto messageBytes = binaryMessage.value().GetMessage().value();
        std::string messageString(messageBytes.begin(), messageBytes.end());
        std::cout << "Received new message: " << messageString << std::endl;
      }
    }
  }

  bool OnStreamError(OperationError *error) override {
    std::cout << "Received an operation error: ";
    if (error->GetMessage().has_value()) {
      std::cout << error->GetMessage().value();
    }
    std::cout << std::endl;
    return false; // Return true to close stream, false to keep stream open.
  }

  void OnStreamClosed() override {
    std::cout << "Subscribe to topic stream closed." << std::endl;
  }
};
```

431
class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
    }

    void OnDisconnectCallback(RpcError error) override {
        std::cout << "OnDisconnectCallback: " << error.StatusToString() << std::endl;
        exit(-1);
    }

    bool OnErrorCallback(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
};

int main() {
    String topic("test/topic/cpp");
    int timeout = 10;

    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
                   connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    SubscribeToTopicRequest request;
    request.SetTopic(topic);
    SubscribeResponseHandler streamHandler;
    SubscribeToTopicOperation operation = ipcClient.NewSubscribeToTopic(streamHandler);
    auto activate = operation.Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation.GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
        std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass Core."
                   << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (response) {
        std::cout << "Successfully subscribed to topic: " << topic << std::endl;
    } else {
        // An error occurred.
        std::cout << "Failed to subscribe to topic: " << topic << std::endl;
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            std::cout << "Operation error: " << error->GetMessage().value() << std::endl;
        } else {
            std::cout << "RPC error: " << response.GetRpcError() << std::endl;
        }
        exit(-1);
    }
}
Publish/subscribe AWS IoT Core MQTT messages

The AWS IoT Core MQTT messaging IPC service lets you send and receive MQTT messages to and from AWS IoT Core. Components can publish messages to AWS IoT Core and subscribe to topics to act on MQTT messages from other sources. For more information about the AWS IoT Core implementation of MQTT, see MQTT in the AWS IoT Core Developer Guide.

**Note**
This MQTT messaging IPC service lets you exchange messages with AWS IoT Core. For more information about how to exchange messages between components, see Publish/subscribe local messages (p. 410).

**Topics**
- Authorization (p. 433)
- PublishToIoTCore (p. 434)
- SubscribeToIoTCore (p. 436)
- Examples (p. 441)

**Authorization**

To use AWS IoT Core MQTT messaging in a custom component, you must define authorization policies that allow your component to send and receive messages on topics. For information about defining authorization policies, see Authorize components to perform IPC operations (p. 403).

Authorization policies for AWS IoT Core MQTT messaging have the following properties.

**IPC service identifier:** `aws.greengrass.ipc.mqttproxy`

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aws.greengrass#PublishToIoTCore</code></td>
<td>Allows a component to publish messages to AWS IoT Core on the MQTT topics that you specify.</td>
<td>A topic string, such as <code>test/topic</code>, or <code>*</code> to allow access to all topics. This topic string supports MQTT topic wildcards (# and +).</td>
</tr>
<tr>
<td><code>aws.greengrass#SubscribeToIoTCore</code></td>
<td>Allows a component to subscribe to messages from AWS IoT Core on the topics that you specify.</td>
<td>A topic string, such as <code>test/topic</code>, or <code>*</code> to allow access to all topics. This topic string supports MQTT topic wildcards (# and +).</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Allows a component to publish and subscribe to AWS IoT Core MQTT messages for the topics that you specify.</td>
<td>A topic string, such as <code>test/topic</code>, or <code>*</code> to allow access to all topics. This topic string supports MQTT topic wildcards (# and +).</td>
</tr>
</tbody>
</table>
**Example Example authorization policy**

The following example authorization policy allows a component to publish and subscribe to all topics.

```json
{
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
      "com.example.MyIoTCorePubSubComponent:mqttproxy:1": {
        "policyDescription": "Allows access to publish/subscribe to all topics.",
        "operations": [
          "aws.greengrass#PublishToIoTCore",
          "aws.greengrass#SubscribeToIoTCore"
        ],
        "resources": ["*"]
      }
    }
  }
}
```

**PublishToIoTCore**

Publishes an MQTT message to AWS IoT Core on a topic.

**Request**

This operation's request has the following parameters:

**topicName (Python: topic_name)**

The topic to which to publish the message.

**qos**

The MQTT QoS to use. This enum, `QOS`, has the following values:

- `AT_MOST_ONCE` – QoS 0. The MQTT message is delivered at most once.
- `AT_LEAST_ONCE` – QoS 1. The MQTT message is delivered at least once.

**payload**

(Optional) The message payload as a blob.

**Response**

This operation doesn't provide any information in its response.

**Examples**

The following examples demonstrate how to call this operation in custom component code.

**Java**

**Example Example: Publish a message**

```java
String topic = "my/topic";
String message = "Hello, World!";
QOS qos = QOS.AT_LEAST_ONCE;
```
Pythont

Example Example: Publish a message

Note
This example assumes that you are using version 1.5.4 or later of the AWS IoT Device SDK for Python v2. If you are using version 1.5.3 of the SDK, see Use AWS IoT Device SDK for Python v2 (p. 399) for information about connecting to the AWS IoT Greengrass Core IPC service.

```python
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import (QOS, PublishToIoTCoreRequest)

TIMEOUT = 10

ipc_client = awsiot.greengrasscoreipc.connect()

topic = "my/topic"
message = "Hello, World"
qos = QOS.AT_LEAST_ONCE

request = PublishToIoTCoreRequest()
request.topic_name = topic
request.payload = bytes(message, "utf-8")
request.qos = qos
operation = ipc_client.new_publish_to_iot_core()
operation.activate(request)
future = operation.get_response()
future.result(TIMEOUT)
```

C++

Example Example: Publish a message

```cpp
#include <iostream>
#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>
using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }

    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }
};
```
bool OnErrorCallback(RpcError error) override {
    // Handle IPC service connection error.
    return true;
}

int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
            connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    String message("Hello, World!");
    String topic("my/topic");
    QOS qos = QOS_AT_MOST_ONCE;
    int timeout = 10;

    PublishToIoTCoreRequest request;
    Vector<uint8_t> messageData({message.begin(), message.end()});
    request.SetTopicName(topic);
    request.SetPayload(messageData);
    request.SetQos(qos);

    PublishToIoTCoreOperation operation = ipcClient.NewPublishToIoTCore();
    auto activate = operation.Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation.GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
        std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
            Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (!response) {
        // Handle error.
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            // Handle operation error.
        } else {
            // Handle RPC error.
        }
    }

    return 0;
}

SubscribeToIoTCore

Subscribe to MQTT messages from AWS IoT Core on a topic or topic filter. The AWS IoT Greengrass Core
software removes subscriptions when the component reaches the end of its lifecycle.
This operation is a subscription operation where you subscribe to a stream of event messages. To use this operation, define a stream response handler with functions that handle event messages, errors, and stream closure. For more information, see Subscribe to IPC event streams (p. 405).

**Event message type:** IoTCoreMessage

**Request**

This operation's request has the following parameters:

- **topicName (Python: topic_name)**
  The topic to which to subscribe. You can use MQTT topic wildcards (# and +) to subscribe to multiple topics.

- **qos**
  The MQTT QoS to use. This enum, QOS, has the following values:
  - **AT_MOST_ONCE** – QoS 0. The MQTT message is delivered at most once.
  - **AT_LEAST_ONCE** – QoS 1. The MQTT message is delivered at least once.

**Response**

This operation's response has the following information:

- **messages**
  The stream of MQTT messages. This object, IoTCoreMessage, contains the following information:

  - **message**
    The MQTT message. This object, MQTTMessage, contains the following information:
    - **topicName (Python: topic_name)**
      The topic to which the message was published.
    - **payload**
      (Optional) The message payload as a blob.

**Examples**

The following examples demonstrate how to call this operation in custom component code.

**Java**

```java
Example Example: Subscribe to messages

```
String message = new String(iotCoreMessage.getMessage().getPayload(),
    StandardCharsets.UTF_8);
String topicName = iotCoreMessage.getMessage().getTopicName();
// Handle message.
} catch (Exception e) {
    e.printStackTrace();
}

@Override
public boolean onStreamError(Throwable throwable) {
    // Handle error.
    return false; // Return true to close stream, false to keep stream open.
}

@Override
public void onStreamClosed() {
    // Handle close.
}

SubscribeToIoTCoreResponseHandler operationResponseHandler = greengrassCoreIPCClient
    .subscribeToIoTCore(subscribeToIoTCoreRequest,
    Optional.of(streamResponseHandler));
operationResponseHandler.getResponse().get();

// Keep the main thread alive, or the process will exit.
try {
    while (true) {
        Thread.sleep(10000);
    }
} catch (InterruptedException e) {
    System.out.println("Subscribe interrupted.");
}

// To stop subscribing, close the stream.
operationResponseHandler.closeStream();

Python

Example Example: Subscribe to messages

Note
This example assumes that you are using version 1.5.4 or later of the AWS IoT Device SDK
for Python v2. If you are using version 1.5.3 of the SDK, see Use AWS IoT Device SDK for
Python v2 (p. 399) for information about connecting to the AWS IoT Greengrass Core IPC
service.

time
import traceback

import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import (
    IoTCoreMessage,
    QOS,
    SubscribeToIoTCoreRequest
)

TIMEOUT = 10

ipc_client = awsiot.greengrasscoreipc.connect()

class StreamHandler(client.SubscribeToIoTCoreStreamHandler):
def __init__(self):
    super().__init__()

def on_stream_event(self, event: IoTCoreMessage) -> None:
    try:
        message = str(event.message.payload, "utf-8")
        topic_name = event.message.topic_name
        # Handle message.
    except:
        traceback.print_exc()

def on_stream_error(self, error: Exception) -> bool:
    # Handle error.
    return True  # Return True to close stream, False to keep stream open.

def on_stream_closed(self) -> None:
    # Handle close.
    pass

topic = "my/topic"
qos = QOS.AT_MOST_ONCE

request = SubscribeToIoTCoreRequest()
request.topic_name = topic
request.qos = qos
handler = StreamHandler()
operation = ipc_client.new_subscribe_to_iot_core(handler)
future = operation.activate(request)
future.result(TIMEOUT)

# Keep the main thread alive, or the process will exit.
while True:
    time.sleep(10)

# To stop subscribing, close the operation stream.
operation.close()

C++

Example Example: Subscribe to messages

```cpp
#include <iostream>
#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IoTCoreResponseHandler : public SubscribeToIoTCoreStreamHandler {
    void OnStreamEvent(IoTCoreMessage *response) override {
        auto message = response->GetMessage();
        if (message.has_value() && message.value().GetPayload().has_value()) {
            auto messageBytes = message.value().GetPayload().value();
            std::string messageString(messageBytes.begin(), messageBytes.end());
            std::string topicName = message.value().GetTopicName().value().c_str();
            // Handle message.
        }
    }

    bool OnStreamError(OperationError *error) override {
        // Handle error.
        return false;  // Return true to close stream, false to keep stream open.
    }
};
```
class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }
    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }
    bool OnErrorCallback(RpcError error) override {
        // Handle IPC service connection error.
        return true;
    }
};

int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " << connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }
    String topic("my/topic");
    QOS qos = QOS_AT_MOST_ONCE;
    int timeout = 10;
    SubscribeToIoTCoreRequest request;
    request.SetTopicName(topic);
    request.SetQos(qos);
    IoTCoreResponseHandler streamHandler;
    SubscribeToIoTCoreOperation operation = ipcClient.NewSubscribeToIoTCore(streamHandler);
    auto activate = operation.Activate(request, nullptr);
    activate.wait();
    auto responseFuture = operation.GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
        std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass Core." << std::endl;
        exit(-1);
    }
    auto response = responseFuture.get();
    if (!response) {
        // Handle error.
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            // Handle operation error.
        } else {
            // Handle RPC error.
        }
    }
}
Examples

Use the following examples to learn how to use the AWS IoT Core MQTT IPC service in your components.

Example AWS IoT Core MQTT publisher (C++)

The following example recipe allows the component to publish to all topics.

JSON

```json
{
"RecipeFormatVersion": "2020-01-25",
"ComponentName": "com.example.IoTCorePublisherCpp",
"ComponentVersion": "1.0.0",
"ComponentDescription": "A component that publishes MQTT messages to IoT Core.",
"ComponentPublisher": "Amazon",
"ComponentConfiguration": {
"DefaultConfiguration": {
"accessControl": {
"aws.greengrass.ipc.mqttproxy": {
"com.example.IoTCorePublisherCpp:mqttproxy:1": {
"policyDescription": "Allows access to publish to all topics.",
"operations": [
"aws.greengrass#PublishToIoTCore"
],
"resources": ["
"*
" ]
},
}
},
"Manifests": [
{"Lifecycle": {
"Run": "/{artifacts:path}/greengrassv2_iotcore_publisher"
},
"Artifacts": [
{"URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.IoTCorePublisherCpp/1.0.0/greengrassv2_iotcore_publisher",
"Permission": {
"Execute": "OWNER"
}
}]
}]
```
The following example C++ application demonstrates how to use the AWS IoT Core MQTT IPC service to publish messages to AWS IoT Core.

```cpp
#include <iostream>
#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
    }

    void OnDisconnectCallback(RpcError error) override {
        std::cout << "OnDisconnectCallback: " << error.StatusToString() << std::endl;
        exit(-1);
    }

    bool OnErrorMessage(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
};

int main() {
    String message("Hello from the Greengrass IPC MQTT publisher (C++)." 山)
    String topic("test/topic/cpp");
    QOS qos = QOS_AT_LEAST_ONCE;
    int timeout = 10;
```
ApiHandle apiHandle(g_allocator);
Io::EventLoopGroup eventLoopGroup(1);
Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
IpcClientLifecycleHandler ipcLifecycleHandler;
GreengrassCoreIpcClient ipcClient(bootstrap);
auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
if (!connectionStatus) {
    std::cerr << "Failed to establish IPC connection: " <<
    connectionStatus.StatusToString() << std::endl;
    exit(-1);
}

while (true) {
    PublishToIoTCoreRequest request;
    Vector<uint8_t> messageData({message.begin(), message.end()});
    request.SetTopicName(topic);
    request.SetPayload(messageData);
    request.SetQos(qos);

    PublishToIoTCoreOperation operation = ipcClient.NewPublishToIoTCore();
    auto activate = operation.Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation.GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
    std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
        Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (response) {
        std::cout << "Successfully published to topic: " << topic << std::endl;
    } else {
        // An error occurred.
        std::cout << "Failed to publish to topic: " << topic << std::endl;
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            std::cout << "Operation error: " << error->GetMessage().value() <<
            std::endl;
        } else {
            std::cout << "RPC error: " << response.GetRpcError() << std::endl;
        }
        exit(-1);
    }

    std::this_thread::sleep_for(std::chrono::seconds(5));
}

return 0;

Example AWS IoT Core MQTT subscriber (C++)

The following example recipe allows the component to subscribe to all topics.

JSON

```json
{
    "RecipeFormatVersion": "2020-01-25",
    "ComponentName": "com.example.IoTCoreSubscriberCpp",
}
"ComponentVersion": "1.0.0",
"ComponentDescription": "A component that subscribes to MQTT messages from IoT Core.",
"ComponentPublisher": "Amazon",
"ComponentConfiguration": {
  "DefaultConfiguration": {
    "accessControl": {
      "aws.greengrass.ipc.mqttproxy": {
        "com.example.IoTCoreSubscriberCpp:mqttproxy:1": {
          "policyDescription": "Allows access to subscribe to all topics.",
          "operations": [
            "aws.greengrass#SubscribeToIoTCore"
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
  }
},
"Manifests": [
  {
    "Lifecycle": {
      "Run": "{artifacts:path}/greengrassv2_iotcore_subscriber"
    },
    "Artifacts": [
      {
        "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.IoTCoreSubscriberCpp/1.0.0/greengrassv2_iotcore_subscriber",
        "Permission": {
          "Execute": "OWNER"
        }
      }
    ]
  }
],

YAML

---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.IoTCoreSubscriberCpp
ComponentVersion: 1.0.0
ComponentDescription: A component that subscribes to MQTT messages from IoT Core.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.mqttproxy:
        com.example.IoTCoreSubscriberCpp:mqttproxy:1:
          policyDescription: Allows access to subscribe to all topics.
          operations:
            - aws.greengrass#SubscribeToIoTCore
          resources:
            - "*"
    Manifests:
      - Lifecycle:
          Run: "{artifacts:path}/greengrassv2_iotcore_subscriber"
        Artifacts:
          - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.IoTCoreSubscriberCpp/1.0.0/greengrassv2_iotcore_subscriber
            Permission:
The following example C++ application demonstrates how to use the AWS IoT Core MQTT IPC service to subscribe to messages from AWS IoT Core.

```cpp
#include <iostream>
#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IoTCoreResponseHandler : public SubscribeToIoTCoreStreamHandler {
    void OnStreamEvent(IoTCoreMessage *response) override {
        auto message = response->GetMessage();
        if (message.has_value() && message.value().GetPayload().has_value()) {
            auto messageBytes = message.value().GetPayload().value();
            std::string messageString(messageBytes.begin(), messageBytes.end());
            std::string messageTopic = message.value().GetTopicName().value().c_str();
            std::cout << "Received new message on topic: " << messageTopic << std::endl;
            std::cout << "Message: " << messageString << std::endl;
        }
    }
    
    bool OnStreamError(OperationError *error) override {
        std::cout << "Received an operation error: ";
        if (error->GetMessage().has_value()) {
            std::cout << error->GetMessage().value();
        }
        std::cout << std::endl;
        return false; // Return true to close stream, false to keep stream open.
    }
    
    void OnStreamClosed() override {
        std::cout << "Subscribe to IoT Core stream closed." << std::endl;
    }
};

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
    }
    
    void OnDisconnectCallback(RpcError error) override {
        std::cout << "OnDisconnectCallback: " << error.StatusToString() << std::endl;
        exit(-1);
    }
    
    bool OnErrorCallback(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
};

int main() {
    String topic("test/topic/cpp");
    QOS qos = QOS_AT_LEAST_ONCE;
    int timeout = 10;
    
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    
    // Your code here...
}
```
Interact with component lifecycle

Use the component lifecycle IPC service to:

- Update the component state on the core device.
- Subscribe to component state updates.
- Prevent the nucleus from stopping the component to apply an update during a deployment.
- Pause and resume component processes.
Authorization

To pause or resume other components from a custom component, you must define authorization policies that allows your component to manage other components. For information about defining authorization policies, see Authorize components to perform IPC operations (p. 403).

Authorization policies for component lifecycle management have the following properties.

**IPC service identifier:** `aws.greengrass.ipc.lifecycle`

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aws.greengrass#PauseComponent</code></td>
<td>Allows a component to pause the components that you specify.</td>
<td>A component name, or * to allow access to all components.</td>
</tr>
<tr>
<td><code>aws.greengrass#ResumeComponent</code></td>
<td>Allows a component to resume the components that you specify.</td>
<td>A component name, or * to allow access to all components.</td>
</tr>
<tr>
<td>*</td>
<td>Allows a component to pause and resume the components that you specify.</td>
<td>A component name, or * to allow access to all components.</td>
</tr>
</tbody>
</table>

**Example Example authorization policy**

The following example authorization policy allows a component to pause and resume all components.

```
{
    "accessControl": {
        "aws.greengrass.ipc.lifecycle": {
            "com.example.MyLocalLifecycleComponent:lifecycle:1": {
                "policyDescription": "Allows access to pause/resume all components.",
                "operations": [
                    "aws.greengrass#PauseComponent",
                    "aws.greengrass#ResumeComponent"
                ],
                "resources": ["*"]
            }
        }
    }
}
```
UpdateState

Update the state of the component on the core device.

Request

This operation's request has the following parameters:

state

The state to set. This enum, LifecycleState, has the following values:

- RUNNING
- ERRORED

Response

This operation doesn't provide any information in its response.

SubscribeToComponentUpdates

Subscribe to receive notifications before the AWS IoT Greengrass Core software updates a component. The notification specifies whether or not the nucleus will restart as part of the update.

The nucleus sends update notifications only if the deployment's component update policy specifies to notify components. The default behavior is to notify components. For more information, see Create deployments (p. 384) and the DeploymentComponentUpdatePolicy object that you can provide when you call the CreateDeployment operation.

Important

Local deployments don't notify components before updates.

This operation is a subscription operation where you subscribe to a stream of event messages. To use this operation, define a stream response handler with functions that handle event messages, errors, and stream closure. For more information, see Subscribe to IPC event streams (p. 405).

Event message type: ComponentUpdatePolicyEvents

Request

This operation's request doesn't have any parameters.

Response

This operation's response has the following information:

messages

The stream of notification messages. This object, ComponentUpdatePolicyEvents, contains the following information:

preUpdateEvent (Python: pre_update_event)

(Optional) An event that indicates that the nucleus wants to update a component. You can respond with the DeferComponentUpdate (p. 449) operation to acknowledge or defer the update until your component is ready to restart. This object, PreComponentUpdateEvent, contains the following information:
DeferComponentUpdate

Acknowledge or defer a component update that you discover with SubscribeToComponentUpdates (p. 448). You specify the amount of time to wait before the nucleus checks again if your component is ready to let the component update proceed. You can also use this operation to tell the nucleus that your component is ready for the update.

If a component doesn't respond to the component update notification, the nucleus waits the amount of time that you specify in the deployment's component update policy. After that timeout, the nucleus proceeds with the deployment. The default component update timeout is 60 seconds. For more information, see Create deployments (p. 384) and the DeploymentComponentUpdatePolicy object that you can provide when you call the CreateDeployment operation.

Request

This operation's request has the following parameters:

deploymentId (Python: deployment_id)
   The ID of the AWS IoT Greengrass deployment to defer.
message
   (Optional) The name of the component for which to defer updates.
   Defaults to the name of the component that makes the request.
recheckAfterMs (Python: recheck_after_ms)
   The amount of time in milliseconds for which to defer the update. The nucleus waits for this amount of time and then sends another PreComponentUpdateEvent that you can discover with SubscribeToComponentUpdates (p. 448).
   Specify 0 to acknowledge the update. This tells the nucleus that your component is ready for the update.
   Defaults to zero milliseconds, which means to acknowledge the update.

Response

This operation doesn't provide any information in its response.

PauseComponent

This feature is available for v2.4.0 and later of the Greengrass nucleus component (p. 136).
Pauses a component's processes on the core device. To resume a component, use the ResumeComponent (p. 450) operation.

You can pause only generic components. If you try to pause any other type of component, this operation throws an InvalidRequestError.

**Note**  
This operation can't pause containerized processes, such as Docker containers. To pause and resume a Docker container, you can use the `docker pause` and `docker resume` commands.

This operation doesn't pause component dependencies or components that depend on the component that you pause. Consider this behavior when you pause a component that is a dependency of another component, because the dependent component might encounter issues when its dependency is paused.

When you restart or shut down a paused component, such as through a deployment, the Greengrass nucleus resumes the component and runs its shutdown lifecycle.

**Important**  
To use this operation, you must define an authorization policy that grants permission to use this operation. For more information, see Authorization (p. 447).

### Minimum SDK versions

The following table lists the minimum versions of the AWS IoT Device SDK that you must use to pause and resume components.

<table>
<thead>
<tr>
<th>SDK</th>
<th>Minimum version</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS IoT Device SDK for Java v2</td>
<td>v1.4.3</td>
</tr>
<tr>
<td>AWS IoT Device SDK for Python v2</td>
<td>v1.6.2</td>
</tr>
<tr>
<td>AWS IoT Device SDK for C++ v2</td>
<td>v1.13.1</td>
</tr>
</tbody>
</table>

### Request

This operation's request has the following parameters:

- **componentName** *(Python: component_name)*
  
  The name of the component to pause, which must be a generic component. For more information, see Component types (p. 323).

### Response

This operation doesn't provide any information in its response.
Important
To use this operation, you must define an authorization policy that grants permission to do so. For more information, see Authorization (p. 447).

Minimum SDK versions
The following table lists the minimum versions of the AWS IoT Device SDK that you must use to pause and resume components.

<table>
<thead>
<tr>
<th>SDK</th>
<th>Minimum version</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS IoT Device SDK for Java v2</td>
<td>v1.4.3</td>
</tr>
<tr>
<td>AWS IoT Device SDK for Python v2</td>
<td>v1.6.2</td>
</tr>
<tr>
<td>AWS IoT Device SDK for C++ v2</td>
<td>v1.13.1</td>
</tr>
</tbody>
</table>

Request
This operation's request has the following parameters:

componentName (Python: component_name)

The name of the component to resume.

Response
This operation doesn't provide any information in its response.

Interact with component configuration
The component configuration IPC service lets you do the following:

- Get and set component configuration parameters.
- Subscribe to component configuration updates.
- Validate component configuration updates before the nucleus applies them.

Topics
- GetConfiguration (p. 451)
- UpdateConfiguration (p. 452)
- SubscribeToConfigurationUpdate (p. 453)
- SubscribeToValidateConfigurationUpdates (p. 454)
- SendConfigurationValidityReport (p. 454)

GetConfiguration
Gets a configuration value for a component on the core device. You specify the key path for which to get a configuration value.
Request

This operation's request has the following parameters:

componentName

(Optional) The name of the component.

Defaults to the name of the component that makes the request.

keyPath

The key path to the configuration value. Specify a list where each entry is the key for a single level in the configuration object. For example, specify ["mqtt", "port"] to get the value of port in the following configuration.

```json
{
  "mqtt": {
    "port": 443
  }
}
```

Response

This operation's response has the following information:

componentName

The name of the component.

value

The requested configuration as an object.

UpdateConfiguration

Updates a configuration value for a component on the core device.

Request

This operation's request has the following parameters:

keyPath

(Optional) The key path to the container node (the object) to update. Specify a list where each entry is the key for a single level in the configuration object. For example, specify the key path ["mqtt"] and the merge value { "port": 443 } to set the value of port in the following configuration.

```json
{
  "mqtt": {
    "port": 443
  }
}
```

The key path must specify a container node (an object) in the configuration. If the node doesn't exist in the component's configuration, this operation creates it and sets its value to the object in valueToMerge.
SubscribeToConfigurationUpdate

Defaults to the root of the configuration object.

timestamp

The current Unix epoch time in milliseconds. This operation uses this timestamp to resolve concurrent updates to the key. If the key in the component configuration has a greater timestamp than the timestamp in the request, then the request fails.

valueToMerge (Python: value_to_merge)

The configuration object to merge at the location that you specify in keyPath. For more information, see Update component configurations (p. 390).

Response

This operation doesn't provide any information in its response.

SubscribeToConfigurationUpdate

Subscribe to receive notifications when a component's configuration updates. When you subscribe to a key, you receive a notification when any child of that key updates.

This operation is a subscription operation where you subscribe to a stream of event messages. To use this operation, define a stream response handler with functions that handle event messages, errors, and stream closure. For more information, see Subscribe to IPC event streams (p. 405).

Event message type: ConfigurationUpdateEvents

Request

This operation's request has the following parameters:

componentName (Python: component_name)

(Optional) The name of the component.

keyPath (Python: key_path)

The key path to the configuration value for which to subscribe. Specify a list where each entry is the key for a single level in the configuration object. For example, specify ["mqtt", "port"] to get the value of port in the following configuration.

```
{
  "mqtt": {
    "port": 443
  }
}
```

Response

This operation's response has the following information:

messages

The stream of notification messages. This object, ConfigurationUpdateEvents, contains the following information:
configurationUpdateEvent (Python: configuration_update_event)

The configuration update event. This object, ConfigurationUpdateEvent, contains the following information:
- componentName (Python: component_name)
  - The name of the component.
- keyPath (Python: key_path)
  - The key path to the configuration value that updated.

SubscribeToValidateConfigurationUpdates

Subscribe to receive notifications before this component's configuration updates. This lets components validate updates to their own configuration. Use the SendConfigurationValidityReport (p. 454) operation to tell the nucleus whether or not the configuration is valid.

Important
Local deployments don't notify components of updates.

This operation is a subscription operation where you subscribe to a stream of event messages. To use this operation, define a stream response handler with functions that handle event messages, errors, and stream closure. For more information, see Subscribe to IPC event streams (p. 405).

Event message type: ValidateConfigurationUpdateEvents

Request

This operation's request doesn't have any parameters.

Response

This operation's response has the following information:

messages

The stream of notification messages. This object, ValidateConfigurationUpdateEvents, contains the following information:

validateConfigurationUpdateEvent (Python: validate_configuration_update_event)

The configuration update event. This object, ValidateConfigurationUpdateEvent, contains the following information:
- deploymentId (Python: deployment_id)
  - The ID of the AWS IoT Greengrass deployment that updates the component.
- configuration
  - The object that contains the new configuration.

SendConfigurationValidityReport

Tell the nucleus whether or not a configuration update to this component is valid. The deployment fails if you tell the nucleus that the new configuration isn't valid. Use the SubscribeToValidateConfigurationUpdates (p. 454) operation to subscribe to validate configuration updates.
If a component doesn't respond to a validate configuration update notification, the nucleus waits the amount of time that you specify in the deployment's configuration validation policy. After that timeout, the nucleus proceeds with the deployment. The default component validation timeout is 20 seconds. For more information, see Create deployments (p. 384) and the ConfigurationValidationPolicy object that you can provide when you call the CreateDeployment operation.

**Request**

This operation's request has the following parameters:

configurationValidityReport (Python: configuration_validity_report)

The report that tells the nucleus whether or not the configuration update is valid. This object, ConfigurationValidityReport, contains the following information:

status

The validity status. This enum, ConfigurationValidityStatus, has the following values:
- ACCEPTED – The configuration is valid and the nucleus can apply it to this component.
- REJECTED – The configuration isn't valid and the deployment fails.

deploymentId (Python: deployment_id)

The ID of the AWS IoT Greengrass deployment that requested the configuration update.

message

(Optional) A message that reports why the configuration isn't valid.

**Response**

This operation doesn't provide any information in its response.

**Retrieve secret values**

Use the secret manager IPC service to retrieve secret values from secrets on the core device. You use the secret manager component (p. 285) to deploy encrypted secrets to core devices. Then, you can use an IPC operation to decrypt the secret and use its value in your custom components.

**Topics**

- Authorization (p. 455)
- GetSecretValue (p. 456)
- Examples (p. 458)

**Authorization**

To use secret manager in a custom component, you must define authorization policies that allow your component to get the value of secrets that you store on the core device. For information about defining authorization policies, see Authorize components to perform IPC operations (p. 403).

Authorization policies for secret manager have the following properties.

**IPC service identifier:** `aws.greengrass.SecretManager`
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aws.greengrass#GetSecretValue</code> or <code>*</code></td>
<td>Allows a component to get the value of secrets that are encrypted on the core device.</td>
<td>A Secrets Manager secret ARN, or <code>*</code> to allow access to all secrets.</td>
</tr>
</tbody>
</table>

**Example Example authorization policy**

The following example authorization policy allows a component to get the value of any secret on the core device.

**Note**

We recommend that in a production environment, you reduce the scope of the authorization policy, so that the component retrieves only the secrets that it uses. You can change the `*` wildcard to a list of secret ARNs when you deploy the component.

```json
{
  "accessControl": {
    "aws.greengrass.SecretManager": {
      "com.example.MySecretComponent:secrets:1": {
        "policyDescription": "Allows access to a secret.",
        "operations": [
          "aws.greengrass#GetSecretValue"
        ],
        "resources": ["*"]
      }
    }
  }
}
```

**GetSecretValue**

Gets the value of a secret that you store on the core device.

This operation is similar to the Secrets Manager operation that you can use to get the value of a secret in the AWS Cloud. For more information, see `GetSecretValue` in the *AWS Secrets Manager API Reference*.

**Request**

This operation's request has the following parameters:

- **secretId**
  The name of the secret to get. You can specify either the Amazon Resource Name (ARN) or the friendly name of the secret.

- **versionId**
  (Optional) The ID of the version to get.
  You can specify either `versionId` or `versionStage`.
  If you don’t specify `versionId` or `versionStage`, this operation defaults to the version with the `AWSCURRENT` label.

- **versionStage**
  (Optional) The staging label of the version to get.
You can specify either versionId or versionStage.

If you don’t specify versionId or versionStage, this operation defaults to the version with the AWSCURRENT label.

**Response**

This operation’s response has the following information:

- **secretId**
  The ID of the secret.
- **versionId**
  The ID of this version of the secret.
- **versionStage**
  The list of staging labels attached to this version of the secret.
- **secretValue**
  The value of this version of the secret. This object, SecretValue, contains the following information.
  - **secretString**
    The decrypted part of the protected secret information that you provided to Secrets Manager as a string.
  - **secretBinary**
    (Optional) The decrypted part of the protected secret information that you provided to Secrets Manager as binary data in the form of a byte array. This property contains the binary data as a base64-encoded string.

This property isn’t used if you created the secret in the Secrets Manager console.

**Examples**

The following examples demonstrate how to call this operation in custom component code.

**Java**

**Example Example: Get a secret value**

```java
String versionStage = "AWSCURRENT";
int TIMEOUT = 10;

GetSecretValueRequest request = new GetSecretValueRequest();
request.setSecretId(secretId);
request.setVersionStage(versionStage);
CompletableFuture<GetSecretValueResponse> futureResponse = greengrassCoreIPCClient
    .getSecretValue(request, Optional.empty()).getResponse();
GetSecretValueResponse result = futureResponse.get(TIMEOUT, TimeUnit.SECONDS);
result.getSecretValue().postFromJson();  // Set the SecretValue’s internal union.
String secretString = result.getSecretValue().getSecretString();
```
# Handle secret value.

**Python**

**Example Example: Get a secret value**

**Note**
This example assumes that you are using version 1.5.4 or later of the AWS IoT Device SDK for Python v2. If you are using version 1.5.3 of the SDK, see [Use AWS IoT Device SDK for Python v2](p. 399) for information about connecting to the AWS IoT Greengrass Core IPC service.

```python
import json
import awsiot.greengrasscoreipc
from awsiot.greengrasscoreipc.model import (GetSecretValueRequest, GetSecretValueResponse, UnauthorizedError)

TIMEOUT = 10

ipc_client = awsiot.greengrasscoreipc.connect()

request = GetSecretValueRequest()
request.secret_id = secret_id
request.version_stage = 'AWSCURRENT'
operation = ipc_client.new_get_secret_value()
operation.activate(request)
futureResponse = operation.get_response()
response = futureResponse.result(TIMEOUT)

secret_json = json.loads(response.secret_value.secret_string)

# Handle secret value.
```

**Examples**

Use the following examples to learn how to use the secret manager IPC service in your components.

**Example: Print secret (Python)**

This example component prints the value of a secret that you deploy to the core device.

**Important**
This example component prints the value of a secret, so use it only with secrets that store test data. Don't use this component to print the value of a secret that stores important information.

**Topics**
- Recipe (p. 458)
- Artifacts (p. 460)
- Usage (p. 461)

**Recipe**

The following example recipe defines a secret ARN configuration parameter and allows the component to get the value of any secret on the core device.
**Note**

We recommend that in a production environment, you reduce the scope of the authorization policy, so that the component retrieves only the secrets that it uses. You can change the * wildcard to a list of secret ARNs when you deploy the component.

**JSON**

```json
{
    "RecipeFormatVersion": "2020-01-25",
    "ComponentName": "com.example.PrintSecret",
    "ComponentVersion": "1.0.0",
    "ComponentDescription": "Prints the value of an AWS Secrets Manager secret.",
    "ComponentPublisher": "Amazon",
    "ComponentDependencies": {
        "aws.greengrass.SecretManager": {
            "VersionRequirement": "^2.0.0",
            "DependencyType": "HARD"
        }
    },
    "ComponentConfiguration": {
        "DefaultConfiguration": {
            "SecretArn": "",
            "accessControl": {
                "aws.greengrass.SecretManager": {
                    "com.example.PrintSecret:secrets:1": {
                        "policyDescription": "Allows access to a secret.",
                        "operations": [
                            "aws.greengrass#GetSecretValue"
                        ],
                        "resources": [
                            "*"
                        ]
                    }
                }
            }
        }
    },
    "Manifests": [
        {
            "Lifecycle": {
                "Install": "python3 -m pip install --user awsiotsdk",
                "Run": "python3 -u {artifacts:path}/print_secret.py '{configuration:/SecretArn}''
            }
        }
    ]
}
```

**YAML**

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PrintSecret
ComponentVersion: 1.0.0
ComponentDescription: Prints the value of a Secrets Manager secret.
ComponentPublisher: Amazon
ComponentDependencies:
    aws.greengrass.SecretManager:
        VersionRequirement: "^2.0.0"
        DependencyType: HARD
ComponentConfiguration:
    DefaultConfiguration:
        SecretArn: ''
        accessControl:
```

---
aws.greengrass.SecretManager:
  com.example.PrintSecret:secrets:1:
    policyDescription: Allows access to a secret.
    operations:
      - aws.greengrass#GetSecretValue
    resources:
      - "**"
Manifests:
  - Lifecycle:
    Install: python3 -m pip install --user awsiotsdk
    Run: python3 -u {artifacts:path}/print_secret.py '{configuration:/SecretArn}'

Artifacts

The following example Python application demonstrates how to use the secret manager IPC service to get the value of a secret on the core device.

```python
import concurrent.futures
import json
import sys
import traceback

import awsiot.greengrasscoreipc
from awsiot.greengrasscoreipc.model import (GetSecretValueRequest,
                                            GetSecretValueResponse,
                                            UnauthorizedError)

TIMEOUT = 10

if len(sys.argv) == 1:
    print('Provide SecretArn in the component configuration.', file=sys.stdout)
    exit(1)

secret_id = sys.argv[1]

try:
    ipc_client = awsiot.greengrasscoreipc.connect()

    request = GetSecretValueRequest()
    request.secret_id = secret_id
    operation = ipc_client.new_get_secret_value()
    operation.activate(request)
    futureResponse = operation.get_response()

    try:
        response = futureResponse.result(TIMEOUT)
        secret_json = json.loads(response.secret_value.secret_string)
        print(f'Successfully got secret: {secret_id}')
        print(f'Secret value: {str(secret_json)}')
    except concurrent.futures.TimeoutError:
        print(f'Timeout occurred while getting secret: {secret_id}', file=sys.stderr)
    except UnauthorizedError as e:
        print(f'Unauthorized error while getting secret: {secret_id}', file=sys.stderr)
        raise e
    except Exception as e:
        print(f'Exception while getting secret: {secret_id}', file=sys.stderr)
        raise e
except Exception:
    print('Exception occurred when using IPC.', file=sys.stderr)
    traceback.print_exc()
    exit(1)
```
Usage

You can use this example component with the secret manager component (p. 285) to deploy and print the value of a secret on your core device.

To create, deploy, and print a test secret

1. Create a Secrets Manager secret with test data.

   ```bash
   aws secretsmanager create-secret \
   --name MyTestGreengrassSecret \
   --secret-string '{"my-secret-key": "my-secret-value"}"
   ```

   Save the ARN of the secret to use in the following steps.

   For more information, see Creating a secret in the AWS Secrets Manager User Guide.

2. Deploy the secret manager component (p. 285) (aws.greengrass.SecretManager) with the following configuration merge update. Specify the ARN of the secret that you created earlier.

   ```json
   {  
     "cloudSecrets": [  
       {  
       }
     ]
   }
   ```

   For more information, see Deploy AWS IoT Greengrass components to devices (p. 383) or the Greengrass CLI deployment command (p. 600).

3. Create and deploy the example component in this section with the following configuration merge update. Specify the ARN of the secret that you created earlier.

   ```json
   {  
     "accessControl": {  
       "aws.greengrass.SecretManager": {  
         "com.example.PrintSecret:secrets:1": {  
           "policyDescription": "Allows access to a secret.",  
           "operations": [  
             "aws.greengrass#GetSecretValue"
           ],  
           "resources": [  
           ]
         }
       }
     }
   }
   ```

   For more information, see Create local AWS IoT Greengrass components (p. 324)

4. View the AWS IoT Greengrass Core software logs to verify that the deployments succeed, and view the com.example.PrintSecret component log to see the secret value printed. For more information, see View AWS IoT Greengrass Core software logs (p. 751) and View component logs (p. 751).
Interact with local shadows

Use the shadow IPC service to interact with local shadows on a device. The device you choose to interact with can be your core device or a connected client device.

Include the shadow manager component (p. 294) as a dependency in your custom component. You can then use IPC operations in your custom components to interact with local shadows on your device through the shadow manager. To enable custom components to react to changes in local shadow states, you can also use the publish/subscribe IPC service to subscribe to shadow events. For more information about using the publish/subscribe service, see the Publish/subscribe local messages (p. 410).

Topics

• Minimum SDK versions (p. 462)
• Authorization (p. 462)
• GetThingShadow (p. 465)
• UpdateThingShadow (p. 468)
• DeleteThingShadow (p. 472)
• ListNamedShadowsForThing (p. 474)

Minimum SDK versions

The following table lists the minimum versions of the AWS IoT Device SDK that you must use to interact with local shadows.

<table>
<thead>
<tr>
<th>SDK</th>
<th>Minimum version</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS IoT Device SDK for Java v2</td>
<td>v1.4.0</td>
</tr>
<tr>
<td>AWS IoT Device SDK for Python v2</td>
<td>v1.6.0</td>
</tr>
<tr>
<td>AWS IoT Device SDK for C++ v2</td>
<td>v1.13.0</td>
</tr>
</tbody>
</table>

Authorization

To use the shadow IPC service in a custom component, you must define authorization policies that allow your component to interact with shadows. For information about defining authorization policies, see Authorize components to perform IPC operations (p. 403).

Authorization policies for shadow interaction have the following properties.

IPC service identifier: `aws.greengrass.ShadowManager`

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aws.greengrass#GetThingShadow</code></td>
<td>Allows a component to retrieve the shadow of a thing.</td>
<td>One of the following strings: $aws/things/thingName/shadow/ to allow access to the classic device shadow</td>
</tr>
</tbody>
</table>
### Authorization

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aws.greengrass#UpdateThingShadow</code></td>
<td>Allows a component to update the shadow of a thing.</td>
<td>One of the following strings:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• $aws/things/thingName/shadow, to allow access to the classic device shadow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• $aws/things/thingName/shadow/name/shadowName, to allow access to a named shadow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• *, to allow access to all topics.</td>
</tr>
<tr>
<td><code>aws.greengrass#DeleteThingShadow</code></td>
<td>Allows a component to delete the shadow of a thing.</td>
<td>One of the following strings:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• $aws/things/thingName/shadow, to allow access to the classic device shadow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• $aws/things/thingName/shadow/name/shadowName, to allow access to a named shadow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• *, to allow access to all topics.</td>
</tr>
<tr>
<td><code>aws.greengrass#ListNamedShadowsForThing</code></td>
<td>Allows a component to retrieve the list of named shadows for a thing.</td>
<td>A thing name string that allows access to the thing to list its shadows.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use * to allow access to things.</td>
</tr>
</tbody>
</table>

**IPC service identifier:** `aws.greengrass.ipc.pubsub`

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aws.greengrass#SubscribeToTopic</code></td>
<td>Allows a component to subscribe to messages for the topics that you specify.</td>
<td>One of the following topic strings:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>shadowTopicPrefix/get/accepted</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>shadowTopicPrefix/get/rejected</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>shadowTopicPrefix/delete/accepted</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>shadowTopicPrefix/delete/rejected</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>shadowTopicPrefix/update/accepted</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>shadowTopicPrefix/update/delta</code></td>
</tr>
</tbody>
</table>
## Authorization

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Resources</th>
</tr>
</thead>
</table>
| • `shadowTopicPrefix/update/rejected` | The value of the topic prefix `shadowTopicPrefix` depends on the type of shadow: | • Classic shadow: `$aws/things/thingName/shadow`  
• Named shadow: `$aws/things/thingName/shadow/name/shadowName` |

Use `*` to allow access to all topics. This topic string doesn’t support MQTT topic wildcards (`#` and `+`).

### Example Example authorization policy

The following example authorization policy allows the component `com.example.MyShadowInteractionComponent` to interact with the classic device shadow and the named shadow `myNamedShadow` for the device `MyThingName`. This policy also allows this component to receive messages on local topics for these shadows.

**JSON**

```json
{
    "accessControl": {
        "aws.greengrass.ShadowManager": {
            "com.example.MyShadowInteractionComponent:shadow:1": {
                "policyDescription": "Allows access to shadows",
                "operations": [
                    "aws.greengrass#GetThingShadow",
                    "aws.greengrass#UpdateThingShadow",
                    "aws.greengrass#DeleteThingShadow"
                ],
                "resources": [
                    "$aws/things/MyThingName/shadow",
                    "$aws/things/MyThingName/shadow/name/myNamedShadow"
                ]
            },
            "com.example.MyShadowInteractionComponent:shadow:2": {
                "policyDescription": "Allows access to things with shadows",
                "operations": [
                    "aws.greengrass#ListNamedShadowsForThing"
                ],
                "resources": [
                    "MyThingName"
                ]
            }
        },
        "aws.greengrass.ipc.pubsub": {
            "com.example.MyShadowInteractionComponent:pubsub:1": {
                "policyDescription": "Allows access to shadow pubsub topics",
                "operations": [
                    "aws.greengrass#SubscribeToTopic"
                ]
            }
        }
    }
}
```
YAML

```yaml
accessControl:
  aws.greengrass.ShadowManager:
    'com.example.MyShadowInteractionComponent:shadow:1':
      policyDescription: 'Allows access to shadows'
      operations:
        - 'aws.greengrass#GetThingShadow'
        - 'aws.greengrass#UpdateThingShadow'
        - 'aws.greengrass#DeleteThingShadow'
      resources:
        - $aws/things/MyThingName/shadow
        - $aws/things/MyThingName/shadow/name/myNamedShadow
  'com.example.MyShadowInteractionComponent:shadow:2':
    policyDescription: 'Allows access to things with shadows'
    operations:
      - 'aws.greengrass#ListNamedShadowsForThing'
    resources:
      - MyThingName
  aws.greengrass.ipc.pubsub:
    'com.example.MyShadowInteractionComponent:pubsub:1':
      policyDescription: 'Allows access to shadow pubsub topics'
      operations:
        - 'aws.greengrass#SubscribeToTopic'
      resources:
        - $aws/things/MyThingName/shadow/get/accepted
        - $aws/things/MyThingName/shadow/name/myNamedShadow/get/accepted
```

GetThingShadow

Get the shadow for a specified thing.

Request

This operation's request has the following parameters:

thingName (Python: thing_name)

The name of the thing.

Type: string

shadowName (Python: shadow_name)

The name of the shadow. To specify the thing's classic shadow, set this parameter to an empty string ("").

Type: string
Response

This operation's response has the following information:

payload

The response state document as a blob.

Type: object that contains the following information:

state

The state information.

This object contains the following information.

desired

The state properties and values requested to be updated in the device.

Type: map of key-value pairs

reported

The state properties and values reported by the device.

Type: map of key-value pairs

delta

The difference between the desired and reported state properties and values. This property is present only if the desired and reported states are different.

Type: map of key-value pairs

metadata

The timestamps for each attribute in the desired and reported sections so that you can determine when the state was updated.

Type: string

timestamp

The epoch date and time that the response was generated.

Type: integer

cientToken (Python: clientToken)

The token that is used to match the request and corresponding response

Type: string

version

The version of the local shadow document.

Type: integer

Errors

This operation can return the following errors.
InvalidArgumentsError

The local shadow service is unable to validate the request parameters. This can occur if the request contains malformed JSON or unsupported characters.

ResourceNotFoundError

The requested local shadow document can't be found.

ServiceError

An internal service error occurred, or the number of requests to the IPC service exceeded the limits specified in the maxLocalRequestsPerSecondPerThing and maxTotalLocalRequestsRate configuration parameters in the shadow manager component.

UnauthorizedError

The component's authorization policy doesn't include required permissions for this operation.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java

```java
byte[] sampleGetThingShadowRequest(String thingName, String shadowName) {
    try {
        // set up IPC client to connect to the IPC server
        EventStreamRPCConnection eventStreamRpcConnection = IPCUtils.getEventStreamRpcConnection();
        GreengrassCoreIPCClient greengrassCoreIPCClient = new GreengrassCoreIPCClient(eventStreamRpcConnection);

        // create the GetThingShadow request
        GetThingShadowRequest getThingShadowRequest = new GetThingShadowRequest();
        getThingShadowRequest.setThingName(thingName);
        getThingShadowRequest.setShadowName(shadowName);

        // retrieve the GetThingShadow response after sending the request to the IPC server
        GetThingShadowResponse getThingShadowResponse = greengrassCoreIPCClient.getThingShadow(getThingShadowRequest,
            Optional.empty()).getResponse().get();

        byte[] payload = getThingShadowResponse.getPayload();
        return payload;
    } catch (ExecutionException e) {
        if (e.getCause() instanceof InvalidArgumentsError) {
            // add error handling
        } else {
            // catch ResourceNotFoundError | UnauthorizedError | ServiceError
        }
    }
}
```

Python

```python
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import GetThingShadowRequest
```
TIMEOUT = 10

def sample_get_thing_shadow_request(thingName, shadowName):
    try:
        # set up IPC client to connect to the IPC server
        ipc_client = awsiot.greengrasscoreipc.connect()

        # create the GetThingShadow request
        get_thing_shadow_request = GetThingShadowRequest()
        get_thing_shadow_request.thing_name = thingName
        get_thing_shadow_request.shadow_name = shadowName

        # retrieve the GetThingShadow response after sending the request to the IPC
        server
        op = ipc_client.new_get_thing_shadow()
        op.activate(get_thing_shadow_request)
        fut = op.get_response()
        result = fut.result(TIMEOUT)
        return result.payload

    except InvalidArgumentsError as e:
        # add error handling
        ...
    # except ResourceNotFoundError | UnauthorizedError | ServiceError

UpdateThingShadow

Update the shadow for the specified thing.

Request

This operation's request has the following parameters:

**thingName (Python: thing_name)**

The name of the thing.

Type: string

**shadowName (Python: shadow_name)**

The name of the shadow. To specify the thing's classic shadow, set this parameter to an empty string ("").

Type: string

**payload**

The request state document as a blob.

Type: object that contains the following information:

**state**

The state information to update. This IPC operation affects only the specified fields.

This object contains the following information. Typically, you'll use either the *desired* property or the *reported* property, but not both in the same request.

**desired**

The state properties and values requested to be updated in the device.
Type: `map` of key-value pairs

reported

The state properties and values reported by the device.

Type: `map` of key-value pairs

clientToken (Python: `client_token`)

(Optional) The token that is used to match the request and corresponding response by the client.

Type: `string`

version

(Optional) The version of the local shadow document to update. The shadow service processes the update only if the specified version matches the latest version that it has.

Type: `integer`

**Response**

This operation's response has the following information:

payload

The response state document as a blob.

Type: `object` that contains the following information:

state

The state information.

This object contains the following information.

desired

The state properties and values requested to be updated in the device.

Type: `map` of key-value pairs

reported

The state properties and values reported by the device.

Type: `map` of key-value pairs

delta

The state properties and values reported by the device.

Type: `map` of key-value pairs

metadata

The timestamps for each attribute in the desired and reported sections so that you can determine when the state was updated.

Type: `string`

timestamp

The epoch date and time that the response was generated.
UpdateThingShadow

Type: integer

clientToken (Python: client_token)

The token that is used to match the request and corresponding response.

Type: string

version

The version of local shadow document after the update is complete.

Type: integer

Errors

This operation can return the following errors.

ConflictError

The local shadow service encountered a version conflict during the update operation. This occurs when the version in the request payload doesn't match the version in the latest available local shadow document.

InvalidArgumentsError

The local shadow service is unable to validate the request parameters. This can occur if the request contains malformed JSON or unsupported characters.

A valid payload has the following properties:

- The state node exists, and is an object that contains the desired or reported state information.
- The desired and reported nodes are either objects or null. At least one of these objects must contain valid state information.
- The depth of the desired and reported objects can't exceed eight nodes.
- The length of the clientToken value can't exceed 64 characters.
- The version value must be 1 or higher.

ServiceError

An internal service error occurred, or the number of requests to the IPC service exceeded the limits specified in the maxLocalRequestsPerSecondPerThing and maxTotalLocalRequestsRate configuration parameters in the shadow manager component.

UnauthorizedError

The component's authorization policy doesn't include required permissions for this operation.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java

```java
byte[] sampleUpdateThingShadowRequest(String thingName, String shadowName, byte[] updateDocument) {
    try {
        // set up IPC client to connect to the IPC server
```
EventStreamRPCConnection eventStreamRpcConnection =
IPCUUils.getEventStreamRpcConnection();
GreengrassCoreIPCClient greengrassCoreIPCClient = new
GreengrassCoreIPCClient(eventStreamRpcConnection);

// create the UpdateThingShadow request
UpdateThingShadowRequest updateThingShadowRequest = new
UpdateThingShadowRequest();
updateThingShadowRequest.setThingName(thingName);
updateThingShadowRequest.setShadowName(shadowName);
updateThingShadowRequest.setPayload(updateDocument);

// retrieve the UpdateThingShadow response after sending the request to the IPC
server
UpdateThingShadowResponse updateThingShadowResponse =
greengrassCoreIPCClient.updateThingShadow(updateThingShadowRequest,
Optional.empty())
.getResponse()
.get();

byte[] payload = updateThingShadowResponse.getPayload();
return payload;

} catch (ExecutionException e) {
if (e.getCause() instanceof InvalidArgumentsError) {
// add error handling
}
...
// catch ConflictError | UnauthorizedError | ServiceError
}

Python

import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import UpdateThingShadowRequest

TIMEOUT = 10

def sample_update_thing_shadow_request(thingName, shadowName, payload):
try:
    # set up IPC client to connect to the IPC server
    ipc_client = awsiot.greengrasscoreipc.connect()

    # create the UpdateThingShadow request
    update_thing_shadow_request = UpdateThingShadowRequest()
    update_thing_shadow_request.thing_name = thingName
    update_thing_shadow_request.shadow_name = shadowName
    update_thing_shadow_request.payload = payload

    # retrieve the UpdateThingShadow response after sending the request to the IPC
    server
    op = ipc_client.new_update_thing_shadow()
    op.activate(update_thing_shadow_request)
    fut = op.get_response()

    result = fut.result(TIMEOUT)
    return result.payload

except InvalidArgumentsError as e:
    # add error handling
    ...
    # except ConflictError | UnauthorizedError | ServiceError
DeleteThingShadow
Delete the shadow for the specified thing.

Request
This operation's request has the following parameters:

- thingName (Python: thing_name)
  The name of the thing.
  Type: string
- shadowName (Python: shadow_name)
  The name of the shadow. To specify the thing's classic shadow, set this parameter to an empty string (" ").
  Type: string

Response
This operation's response has the following information:

- payload
  An empty response state document.

Errors
This operation can return the following errors:

- InvalidArgumentsError
  The local shadow service is unable to validate the request parameters. This can occur if the request contains malformed JSON or unsupported characters.

- ResourceNotFoundError
  The requested local shadow document can't be found.

- ServiceError
  An internal service error occurred, or the number of requests to the IPC service exceeded the limits specified in the maxLocalRequestsPerSecondPerThing and maxTotalLocalRequestsRate configuration parameters in the shadow manager component.

- UnauthorizedError
  The component's authorization policy doesn't include required permissions for this operation.

Examples
The following examples demonstrate how to call this operation in custom component code.

Java

```java
byte[] sampleDeleteThingShadowRequest(String thingName, String shadowName) {
```
try {
    // set up IPC client to connect to the IPC server
    EventStreamRPConnection eventStreamRpcConnection = IPCUtils.getEventStreamRpcConnection();
    GreengrassCoreIPCClient greengrassCoreIPCClient = new GreengrassCoreIPCClient(eventStreamRpcConnection);

    // create the DeleteThingShadow request
    DeleteThingShadowRequest deleteThingShadowRequest = new DeleteThingShadowRequest();
    deleteThingShadowRequest.setThingName(thingName);
    deleteThingShadowRequest.setShadowName(shadowName);

    // retrieve the DeleteThingShadow response after sending the request to the IPC server
    DeleteThingShadowResponse deleteThingShadowResponse =
        greengrassCoreIPCClient.deleteThingShadow(deleteThingShadowRequest, Optional.empty())
            .getResponse()
            .get();

    byte[] payload = deleteThingShadowResponse.getPayload();
    return payload;
}

// catch Exception or Error

Python

```python
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import DeleteThingShadowRequest

TIMEOUT = 10

def sample_delete_thing_shadow_request(thingName, shadowName):
    try:
        # set up IPC client to connect to the IPC server
        ipc_client = awsiot.greengrasscoreipc.connect()

        # create the DeleteThingShadow request
        delete_thing_shadow_request = DeleteThingShadowRequest()
        delete_thing_shadow_request.thing_name = thingName
        delete_thing_shadow_request.shadow_name = shadowName

        # retrieve the DeleteThingShadow response after sending the request to the IPC server
        op = ipc_client.new_delete_thing_shadow()
        op.activate(delete_thing_shadow_request)
        fut = op.get_response()

        result = fut.result(TIMEOUT)
        return result.payload
    except InvalidArgumentsError as e:
        # add error handling
    ...
    # except ResourceNotFound | Unauthorized | ServiceError
```
ListNamedShadowsForThing

List the named shadows for the specified thing.

Request

This operation's request has the following parameters:

thingName (Python: thing_name)
  The name of the thing.
  Type: string

pageSize (Python: page_size)
  (Optional) The number of shadow names to return in each call.
  Type: integer
  Default: 25
  Maximum: 100

nextToken (Python: next_token)
  (Optional) The token to retrieve the next set of results. This value is returned on paged results and is
  used in the call that returns the next page.
  Type: string

Response

This operation's response has the following information:

results
  The list of shadow names.
  Type: array

timestamp
  (Optional) The date and time that the response was generated.
  Type: integer

nextToken (Python: next_token)
  (Optional) The token value to use in paged requests to retrieve the next page in the sequence. This
token isn't present when there are no more shadow names to return.
  Type: string

Note
  If the requested page size exactly matches the number of shadow names in the response,
  then this token is present; however, when used, it returns an empty list.

Errors

This operation can return the following errors.
InvalidArgumentsError

The local shadow service is unable to validate the request parameters. This can occur if the request contains malformed JSON or unsupported characters.

ResourceNotFoundError

The requested local shadow document can't be found.

ServiceError

An internal service error occurred, or the number of requests to the IPC service exceeded the limits specified in the `maxLocalRequestsPerSecondPerThing` and `maxTotalLocalRequestsRate` configuration parameters in the shadow manager component.

UnauthorizedError

The component's authorization policy doesn't include required permissions for this operation.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java

```java
List<String> sampleListNamedShadowsForThingRequest(String thingName, String nextToken, int pageSize) {
    try {
        // set up IPC client to connect to the IPC server
        EventStreamRPCConnection eventStreamRpcConnection = IPCUtils.getEventStreamRpcConnection();
        GreengrassCoreIPCClient greengrassCoreIPCClient = new GreengrassCoreIPCClient(eventStreamRpcConnection);

        // create the ListNamedShadowsForThing request
        ListNamedShadowsForThingRequest request = new ListNamedShadowsForThingRequest();
        request.setThingName(thingName);
        request.setNextToken(nextToken);
        request.setPageSize(pageSize);

        // retrieve the ListNamedShadowsForThing response after sending the request to the IPC server
        ListThingShadowsForThingResponse response =
            greengrassCoreIPCClient.listNamedShadowsForThing(request, Optional.empty())
                .getResponse()
                .get();

        List<String> listOfNamedShadows = response.getResults().get()
        // pagination token used to get next set of data
        // null indicates that all named shadows were received
        String tokenForNextQuery = response.getNextToken().get()
        return listOfNamedShadows;
    }
    catch (ExecutionException e) {
        if (e.getCause() instanceof InvalidArgumentsError) {
            // add error handling
        } ...
        // catch ResourceNotFoundError | UnauthorizedError | ServiceError
    }
}
```
Python

```python
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import ListNamedShadowsForThingRequest

TIMEOUT = 10

def sample_list_named_shadows_for_thing_request(thingName, nextToken, pageSize):
    try:
        # set up IPC client to connect to the IPC server
        ipc_client = awsiot.greengrasscoreipc.connect()

        # create the ListNamedShadowsForThingRequest request
        list_named_shadows_for_thing_request = ListNamedShadowsForThingRequest()
        list_named_shadows_for_thing_request.thing_name = thingName
        list_named_shadows_for_thing_request.next_token = nextToken
        list_named_shadows_for_thing_request.page_size = pageSize

        # retrieve the ListNamedShadowsForThingRequest response after sending the request to the IPC server
        op = ipc_client.new_list_named_shadows_for_thing()
        op.activate(list_named_shadows_for_thing_request)
        fut = op.get_response()

        list_result = fut.result(TIMEOUT)

        # additional returned fields
        timestamp = list_result.timestamp
        next_token = result.next_token
        named_shadow_list = list_result.results

        return named_shadow_list, next_token, timestamp
    except InvalidArgumentsError as e:
        # add error handling
        ...
    # except ResourceNotFoundError | UnauthorizedError | ServiceError
```
Interact with local IoT devices

Client devices are local IoT devices that connect to and communicate with a Greengrass core device over MQTT. You can connect client devices to core devices to do the following:

• Relay messages and data between client devices and AWS IoT Core.
• Interact with MQTT messages in Greengrass components.

To connect to a core device, client devices can use cloud discovery. Client devices connect to the AWS IoT Greengrass cloud service to retrieve information about core devices to which they can connect. Then, they can connect to a core device to process their messages and sync their data with the AWS IoT Core cloud service.

You can follow a tutorial that walks through how to configure a core device to connect and communicate with an AWS IoT thing. This tutorial also explores how to develop a custom Greengrass component that interacts with client devices. For more information, see Tutorial: Connect and test client devices (p. 477).

Topics
• Tutorial: Connect and test client devices (p. 477)
• AWS-provided client device components (p. 488)
• Connect client devices to core devices (p. 482)
• Relay MQTT messages between client devices and AWS IoT Core (p. 506)
• Interact with client devices in components (p. 508)
• Troubleshooting client devices (p. 510)

Tutorial: Connect and test client devices

You can complete this tutorial to configure a core device to interact with local IoT devices, called client devices, that connect to the core device over MQTT. In this tutorial, you configure AWS IoT things to use cloud discovery to connect to the core device as client devices. When you configure cloud discovery, a client device can send a request to the AWS IoT Greengrass cloud service to discover core devices. The response from AWS IoT Greengrass includes connectivity information and certificates for the core devices that you configure the client device to discover. Then, the client device can use this information to connect to an available core device where it can communicate over MQTT.

In this tutorial, you do the following:

• Review and update the core device's permissions, if needed.
• Associate client devices to the core device, so they can discover the core device using cloud discovery.
• Deploy Greengrass components to the core device to enable client device support.
• Connect client devices to the core device and test communication with the AWS IoT Core cloud service.
• Develop a custom Greengrass component that interacts with the client devices.

This tutorial uses a single core device and a single client device. You can also follow the tutorial to connect and test multiple client devices.

You can expect to spend 30–50 minutes on this tutorial.
Prerequisites

To complete this tutorial, you must meet the following prerequisites:

- An AWS account. If you don't have one, see Set up an AWS account (p. 44).
- An AWS Identity and Access Management (IAM) user with administrator permissions.
- A Greengrass core device. For more information about how to set up a core device, see Setting up AWS IoT Greengrass core devices (p. 41).
  - The core device must be on the same network as the client devices to connect.
  - (Optional) To complete the module where you develop a custom component that interacts with client devices, the core device must run the Greengrass CLI. For more information, see Install the Greengrass CLI (p. 595).
- An AWS IoT thing to connect as a client device in this tutorial. For more information, see Create AWS IoT resources in the AWS IoT Core Developer Guide.
  - The client device's AWS IoT policy must allow the greengrass:Discover permission. For more information, see Minimal AWS IoT policy for client devices (p. 617).
  - The client device must be on the same network as the core device.
  - The client device must run Python 3.
  - The client device must run Git.

Review and update the core device AWS IoT policy

To support client devices, a core device's AWS IoT policy must allow the following permissions:

- greengrass:PutCertificateAuthorities
- greengrass:VerifyClientDeviceIdentity
- greengrass:VerifyClientDeviceIoTCertificateAssociation
- greengrass:GetConnectivityInfo
- greengrass:UpdateConnectivityInfo – (Optional) This permission is required to use the IP detector component (p. 177), which reports the core device's network connectivity information to the AWS IoT Greengrass cloud service.

For more information, see AWS IoT policies for data plane operations (p. 611) and Minimal AWS IoT policy to support client devices (p. 616).

In this section, you review the AWS IoT policies for your core device and add any required permissions that are missing. If you used the AWS IoT Greengrass Core software installer to provision resources (p. 46), your core device has an AWS IoT policy that allows access to all AWS IoT Greengrass actions (greengrass:*).

To review and update a core device's AWS IoT policy

1. In the AWS IoT Greengrass console navigation menu, choose Core devices.
2. On the Core devices page, choose the core device to update.
3. On the core device details page, choose the link to the core device's Thing. This link opens the thing details page in the AWS IoT console.
4. On the thing details page, choose Certificates.
5. In the Certificates tab, choose the thing's active certificate.
7. In the Policies tab, choose the AWS IoT policy to review and update. You can add the required permissions to any policy that is attached to the core device's active certificate.
Enable client device support

For a client device to use cloud discovery to connect to a core device, you must associate the devices.
When you associate a client device to a core device, you enable that client device to retrieve the core
device’s IP addresses and certificates to use to connect.

To enable client devices to securely connect to a core device and communicate with Greengrass
components and AWS IoT Core, you deploy the following Greengrass components to the core device:

- **Client device auth (p. 146)** (aws.greengrass.clientdevices.Auth)

  Deploy the client device auth component to authenticate client devices and authorize client device
  actions. This component allows your AWS IoT things to connect to a core device.

  This component requires some configuration to use it. You must specify groups of client devices and
  the operations that each group is authorized to perform, such as to connect and communicate over
  MQTT. For more information, see client device auth component configuration (p. 148).

- **MQTT broker (Moquette) (p. 283)** (aws.greengrass.clientdevices.mqtt.Moquette)

  Deploy the Moquette MQTT broker component to run the open source Moquette MQTT broker. The
  Moquette MQTT broker is compliant with MQTT 3.1.1 and includes local support for QoS 0, QoS 1,
  QoS 2, retained messages, last will messages, and persistent subscriptions.

  You aren’t required to configure this component to use it. However, you can configure the port where
  this component operates the MQTT broker. By default, it uses port 8883.

- **MQTT bridge (p. 280)** (aws.greengrass.clientdevices.mqtt.Bridge)

  (Optional) Deploy the MQTT bridge component to relay messages between client devices (local MQTT),
  local publish/subscribe, and AWS IoT Core MQTT. Configure this component to sync client devices with
  AWS IoT Core and interact with client devices from Greengrass components.

  This component requires configuration to use. You must specify the topic mappings where
  this component relays messages. For more information, see MQTT bridge component
  configuration (p. 281).

- **IP detector (p. 177)** (aws.greengrass.clientdevices.IPDetector)

  (Optional) Deploy the IP detector component to automatically report the core device’s MQTT broker
  endpoints to the AWS IoT Greengrass cloud service. You cannot use this component if you have a
  complex network setup, such as one where a router forwards the MQTT broker port to the core device.

  You aren’t required to configure this component to use it.

In this section, you use the AWS IoT Greengrass console to associate client devices and deploy client
device components to a core device.
To enable client device support

1. Navigate to the AWS IoT Greengrass console.
2. In the left navigation menu, choose Core devices.
3. On the Core devices page, choose the core device where you want to enable client device support.
4. On the core device details page, choose the Client devices tab.
5. On the Client devices tab, choose Configure cloud discovery.

The Configure core device discovery page opens. On this page, you can associate client devices to a core device and deploy client device components. This page selects the core device for you in Step 1: Select target core devices.

Note
You can also use this page to configure core device discovery for a thing group. If you choose this option, you can deploy client device components to all core devices in a thing group. However, if you choose this option, you must manually associate client devices to each core device later after you create the deployment. In this tutorial, you configure a single core device.

6. In Step 2: Associate client devices, associate the client device's AWS IoT thing to the core device. This enables the client device to use cloud discovery to retrieve the core device's connectivity information and certificates. Do the following:
   a. Choose Associate client devices.
   b. In the Associate client devices with core device modal, enter the name of the AWS IoT thing to associate.
   c. Choose Add.
   d. Choose Associate.

7. In Step 3: Configure and deploy Greengrass components, deploy components to enable client device support. If the target core device has a previous deployment, this page revises that deployment. Otherwise, this page creates a new deployment for the core device. Do the following to configure and deploy the client device components:
   a. The core device must run Greengrass nucleus (p. 136) v2.2.0 or later to support client devices. If the core device runs an earlier version, select the box to deploy the aws.greengrass.Nucleus component.
      Note
      If you upgrade the Greengrass nucleus from an earlier minor version, and the core device runs public components (p. 132) that depend on the nucleus, you must also update the public components to newer versions. You can configure the version of these components when you review the deployment later in this tutorial. For more information, see Update the AWS IoT Greengrass Core software (OTA) (p. 129).
   b. For the aws.greengrass.clientdevices.Auth component, choose Edit configuration.
   c. In the Edit configuration modal for the client device auth component, configure an authorization policy that allows client devices to publish and subscribe to the MQTT broker on the core device. Do the following:
      i. Under Configuration, in the Configuration to merge code block, enter the following configuration, which contains a device group authorization policy. Each device group authorization policy specifies a set of actions and the resources on which a client device can perform those actions.
         • This policy allows client devices whose names start with MyClientDevice to connect and communicate on all MQTT topics. Replace MyClientDevice* with the name of the AWS IoT thing to connect as a client device. You can also specify a name with the *
wildcard that matches the client device's name. The * wildcard must be at the end of the name.

If you have a second client device to connect, replace $\text{MyOtherClientDevice}^*$ with the name of that client device, or a wildcard pattern that matches that client device's name. Otherwise, you can remove or keep this section of the selection rule that allows client devices with names that match $\text{MyOtherClientDevice}^*$ to connect and communicate.

- This policy uses an OR operator to also allow client devices whose names start with $\text{MyOtherClientDevice}$ to connect and communicate on all MQTT topics. You can remove this clause in the selection rule or modify it to match the client devices to connect.

- This policy allows the client devices to publish and subscribe on all MQTT topics. To follow best security practices, restrict the mqtt:publish and mqtt:subscribe operations to the minimal set of topics that the client devices use to communicate.

```json
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: $\text{MyClientDevice}^*$ OR thingName: $\text{MyOtherClientDevice}^*$",
        "policyName": "MyClientDevicePolicy"
      }
    },
    "policies": {
      "MyClientDevicePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish to all topics.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "*
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to all topics.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "*
          ]
        }
      }
    }
  }
}
```
Connect client devices

For more information, see Client device auth component configuration (p. 148).

ii. Choose Confirm.

d. For the `aws.greengrass.clientdevices.mqtt.Bridge` component, choose Edit configuration.

e. In the Edit configuration modal for the MQTT bridge component, configure a topic mapping that relays MQTT messages from client devices to AWS IoT Core. Do the following:

i. Under Configuration, in the Configuration to merge code block, enter the following configuration. This configuration specifies to relay MQTT messages on the clients/+/hello/world topic filter from client devices to the AWS IoT Core cloud service. For example, this topic filter matches the clients/MyClientDevice1/hello/world topic.

```json
{
  "mqttTopicMapping": {
    "HelloWorldIotCoreMapping": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

For more information, see MQTT bridge component configuration (p. 281).

ii. Choose Confirm.

8. Choose Review and deploy to review the deployment that this page creates for you.

9. If you haven't previously set up the Greengrass service role (p. 630) in this Region, the console opens a modal to set up the service role for you. The client device auth component uses this service role to verify the identity of client devices, and the IP detector component uses this service role to manage core device connectivity information. Choose Grant permissions.

10. On the Review page, choose Deploy to start the deployment to the core device.

11. To verify that the deployment succeeds, check the status of the deployment, and check the logs on the core device. To check the status of the deployment on the core device, you can choose Target in the deployment Overview. For more information, see the following:

   - Check deployment status (p. 395)
   - View AWS IoT Greengrass Core software logs (p. 751)

Connect client devices

Client devices can use the AWS IoT Device SDK to discover, connect, and communicate with a core device. In this section, you install the AWS IoT Device SDK v2 for Python and run the Greengrass discovery sample application from the AWS IoT Device SDK.

**Note**

The AWS IoT Device SDK is also available in other programming languages. This tutorial uses the AWS IoT Device SDK v2 for Python, but you can explore the other SDKs for your use case. For more information, see AWS IoT Device SDKs in the AWS IoT Core Developer Guide.

**To connect a client device to a core device**

1. Download and install the AWS IoT Device SDK v2 for Python to the AWS IoT thing to connect as a client device.

   On the client device, do the following:

   a. Clone the AWS IoT Device SDK v2 for Python repository to download it.
b. Install the AWS IoT Device SDK v2 for Python.

```bash
python3 -m pip install --user ./aws-iot-device-sdk-python-v2
```

2. Change to the samples folder in the AWS IoT Device SDK v2 for Python.

```bash
cd aws-iot-device-sdk-python-v2/samples
```

3. Run the sample Greengrass discovery application. This application expects arguments that specify the client device thing name, the MQTT topic and message to use, and the certificates that authenticate and secure the connection. The following example sends a Hello World message to the clients/MyClientDevice1/hello/world topic.

**Note**
This topic is the same topic where you configured the MQTT bridge to relay messages to AWS IoT Core earlier.

- Replace both instances of `MyClientDevice1` with the client device's thing name.
- Replace `~/certs/AmazonRootCA1.pem` with the path to the Amazon root CA certificate on the client device.
- Replace `~/certs/device.pem.crt` with the path to the device certificate on the client device.
- Replace `~/certs/private.pem.key` with the path to the private key file on the client device.
- Replace `us-east-1` with the AWS Region where your client device and core device operate.

```bash
python3 basic_discovery.py \
--thing-name MyClientDevice1 \
--topic 'clients/MyClientDevice1/hello/world' \
--message 'Hello World!' \
--root-ca ~/certs/AmazonRootCA1.pem \
--cert ~/certs/device.pem.crt \
--key ~/certs/private.pem.key \
--region us-east-1 \
--verbosity Warn
```

The discovery sample application sends the message 10 times and disconnects. It also subscribes to the same topic where it publishes messages. If the output indicates that the application received MQTT messages on the topic, the client device can successfully communicate with the core device.
Develop a component that interacts with client devices

Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!", "sequence": 1}

Publish received on topic clients/MyClientDevice1/hello/world
b'"message": "Hello World!", "sequence": 1'}

...  

Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!", "sequence": 9}

Publish received on topic clients/MyClientDevice1/hello/world
b'"message": "Hello World!", "sequence": 9'}

If the application outputs an error instead, see Troubleshooting Greengrass discovery issues (p. 510).

You can also view the Greengrass logs on the core device to verify if the client device successfully connects and sends messages. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

4. Verify that the MQTT bridge relays the messages from the client device to AWS IoT Core. You can use the MQTT test client in the AWS IoT Core console to subscribe to an MQTT topic filter. Do the following:
   a. Navigate to the AWS IoT console.
   b. In the left navigation menu, under Test, choose MQTT test client.
   c. On the Subscribe to a topic tab, for Topic filter, enter clients/+/hello/world to subscribe to client device messages from the core device.
   d. Choose Subscribe.
   e. Run the publish/subscribe application on the client device again.

   The MQTT test client displays the messages that you send from the client device on topics that match this topic filter.

Develop a component that interacts with client devices

You can develop Greengrass components that interact with client devices. Components use interprocess communication (IPC) (p. 396) and the local publish/subscribe interface (p. 410) to communicate on a core device. To interact with client devices, configure the MQTT bridge component to relay messages between client devices and local publish/subscribe.

In this section, you update the MQTT bridge component to relay messages from client devices to the local publish/subscribe interface. Then, you develop a component that subscribes to these messages and prints the messages when it receives them.

To develop a component that interacts with client devices

1. Revise the deployment to the core device and configure the MQTT bridge component to relay messages from client devices to local publish/subscribe. Do the following:
   a. Navigate to the AWS IoT Greengrass console.
   b. In the left navigation menu, choose Core devices.
   c. On the Core devices page, choose the core device that you are using for this tutorial.
   d. On the core device details page, choose the Client devices tab.
Develop a component that interacts with client devices

On the **Client devices** tab, choose **Configure cloud discovery**.

The **Configure core device discovery** page opens. On this page, you can change or configure which client device components deploy to the core device.

In **Step 3**, for the `aws.greengrass.clientdevices.mqtt.Bridge` component, choose **Edit configuration**.

In the **Edit configuration** modal for the MQTT bridge component, configure a topic mapping that relays MQTT messages from client devices to the local publish/subscribe interface. Do the following:

- **Under Configuration**, in the **Configuration to merge** code block, enter the following configuration. This configuration specifies to relay MQTT messages on topics that match the `clients/+/hello/world` topic filter from client devices to the AWS IoT Core cloud service and the local Greengrass publish/subscribe broker.

```json
{
   "mqttTopicMapping": {
      "HelloWorldIoTCoreMapping": {
         "topic": "clients/+/hello/world",
         "source": "LocalMqtt",
         "target": "IotCore"
      },
      "HelloWorldPubsubMapping": {
         "topic": "clients/+/hello/world",
         "source": "LocalMqtt",
         "target": "Pubsub"
      }
   }
}
```

For more information, see **MQTT bridge component configuration** (p. 281).

Choose **Confirm**.

**On the Review page, choose Deploy** to start the deployment to the core device.

To verify that the deployment succeeds, check the status of the deployment, and check the logs on the core device. To check the status of the deployment on the core device, you can choose **Target** in the deployment **Overview**. For more information, see the following:

- Check deployment status (p. 395)
- View AWS IoT Greengrass Core software logs (p. 751)

**Develop and deploy a Greengrass component that subscribes to Hello World messages from client devices.** For more information, see **Create local AWS IoT Greengrass components** (p. 324).

Create a component recipe with the following contents. This recipe specifies installing the AWS IoT Device SDK v2 for Python and running a script that subscribes to the topic and prints messages.

```json
{
   "RecipeFormatVersion": "2020-01-25",
   "ComponentName": "com.example.clientdevices.MyHelloWorldSubscriber",
   "ComponentVersion": "1.0.0",
   "ComponentDescription": "A component that subscribes to Hello World messages from client devices.",
   "ComponentPublisher": "Amazon",
   "ComponentConfiguration": {
      "DefaultConfiguration": {
         "accessControl": {
            "aws.greengrass.ipc.pubsub": {
```
Develop a component that interacts with client devices

```
"com.example.clientdevices.MyHelloWorldSubscriber:pubsub:1": {
  "policyDescription": "Allows access to subscribe to all topics.",
  "operations": [
    "aws.greengrass#SubscribeToTopic"
  ],
  "resources": [
    "*"
  ]
},
}
"Manifests": [
  {
    "Lifecycle": {
      "Install": "python3 -m pip install --user awsiotsdk",
      "Run": "python3 -u {artifacts:path}/hello_world_subscriber.py"
    }
  }
]
```

b. Create a Python script artifact named `hello_world_subscriber.py` with the following contents. This application uses the publish/subscribe IPC service to subscribe to the `clients/MyClientDevice1/hello/world` topic and print messages that it receives. Replace `MyClientDevice1` with the name of the client device.

```python
import concurrent.futures
import sys
import time
import traceback
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import (SubscribetToTopicRequest,
                                             SubscriptionResponseMessage,
                                             UnauthorizedError)

topic = "clients/MyClientDevice1/hello/world"
TIMEOUT = 10

class StreamHandler(client.SubscribeToTopicStreamHandler):
    def __init__(self):
        super().__init__()

    def on_stream_event(self, event: SubscriptionResponseMessage) -> None:
        try:
            message = str(event.binary_message.message, "utf-8")
            print("Received new message: " + message)
        except:
            traceback.print_exc()

    def on_stream_error(self, error: Exception) -> bool:
        print("Received a stream error.", file=sys.stderr)
        traceback.print_exc()
        return False  # Return True to close stream, False to keep stream open.

    def on_stream_closed(self) -> None:
        print("Subscribe to topic stream closed.")
```

486
Develop a component that interacts with client devices

```python
try:
    ipc_client = awsiot.greengrasscoreipc.connect()
    request = SubscribeToTopicRequest()
    request.topic = topic
    handler = StreamHandler()
    operation = ipc_client.new_subscribe_to_topic(handler)
    future = operation.activate(request)

    try:
        future.result(TIMEOUT)
        print('Successfully subscribed to topic: ' + topic)
    except concurrent.futures.TimeoutError as e:
        print('Timeout occurred while subscribing to topic: ' + topic, file=sys.stderr)
        raise e
    except UnauthorizedError as e:
        print('Unauthorized error while subscribing to topic: ' + topic, file=sys.stderr)
        raise e
    except Exception as e:
        print('Exception while subscribing to topic: ' + topic, file=sys.stderr)
        raise e

# Keep the main thread alive, or the process will exit.
try:
    while True:
        time.sleep(10)
except InterruptedError:
    print('Subscribe interrupted.')
except Exception:
    print('Exception occurred when using IPC.', file=sys.stderr)
    traceback.print_exc()
exit(1)
```

3. **Use the Greengrass CLI to deploy the component.**

```bash
sudo /greengrass/v2/bin/greengrass-cli deployment create \
   --recipeDir recipes \
   --artifactDir artifacts \
   --merge "com.example.clientdevices.MyHelloWorldSubscriber=1.0.0"
```

4. **View the component logs to verify that the component installs successfully and subscribes to the topic.**

```bash
sudo tail -f /greengrass/v2/logs/com.example.clientdevices.MyHelloWorldSubscriber.log
```

You can keep the log feed open to verify that the core device receives messages.

4. **On the client device, run the sample Greengrass discovery application again to send messages to the core device.**

```bash
python3 basic_discovery.py \
   --thing-name MyClientDevice1 \
   --topic 'clients/MyClientDevice1/hello/world' \
   --message 'Hello World!' \
   --root-ca ~/certs/AmazonRootCA1.pem \
   --cert ~/certs/device.pem.crt \
   --key ~/certs/private.pem.key \
   --region us-east-1 \
   --verbosity Warn
```

487
5. View the component logs again to verify that the component receives and prints the messages from the client device.

```
sudo tail -f /greengrass/v2/logs/com.example.clientdevices.MyHelloWorldSubscriber.log
```

You've completed this tutorial. The client device connects to the core device and sends MQTT messages to AWS IoT Core and Greengrass components. For more information about the topics covered in this tutorial, see the following:

- Associate client devices (p. 491)
- Manage core device endpoints (p. 492)
- Test client device communications (p. 495)
- Greengrass discovery RESTful API (p. 502)
- Relay MQTT messages between client devices and AWS IoT Core (p. 506)
- Interact with client devices in components (p. 508)

## AWS-provided client device components

AWS IoT Greengrass provides the following public components that you can deploy to core devices. These components enable client devices to connect and communicate with a core device.

**Note**

Several AWS-provided components depend on specific minor versions of the Greengrass nucleus. Because of this dependency, you need to update these components when you update the Greengrass nucleus to a new minor version. For information about the specific versions of the nucleus that each component depends on, see the corresponding component topic. For more information about updating the nucleus, see Update the AWS IoT Greengrass Core software (OTA) (p. 129).

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Depends on nucleus</th>
<th>Component type (p. 323)</th>
<th>Open source (p. 761)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client device auth (p. 146)</td>
<td>Enables local IoT devices, called client devices, to connect to the core device.</td>
<td>Yes</td>
<td>Plugin</td>
<td>Yes</td>
</tr>
<tr>
<td>IP detector (p. 177)</td>
<td>Reports MQTT broker connectivity information to AWS IoT Greengrass, so client devices can discover how to connect.</td>
<td>Yes</td>
<td>Plugin</td>
<td>Yes</td>
</tr>
<tr>
<td>MQTT bridge (p. 280)</td>
<td>Relays MQTT messages between client devices, local AWS IoT</td>
<td>No</td>
<td>Plugin</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Connect client devices to core devices

You can configure cloud discovery to connect client devices to core devices. When you configure cloud discovery, client devices can connect to the AWS IoT Greengrass cloud service to retrieve information about core devices to which they can connect. Then, the client devices can attempt to connect to each core device until they successfully connect.

To use cloud discovery, you must do the following:

- Associate client devices to the core devices to which they can connect.
- Specify the MQTT broker endpoints where client devices can connect to each core device.
- Deploy components to the core device that enable support for client devices.

You can also deploy optional components to do the following:

- Relay messages between client devices, Greengrass components, and the AWS IoT Core cloud service.
- Automatically manage core device MQTT broker endpoints for you.

You must also review and update the core device's AWS IoT policy to ensure that it has the permissions required to connect client devices. For more information, see Review and update the core device AWS IoT policy (p. 478).

After you configure cloud discovery, you can test communications between a client device and a core device. For more information, see Test client device communications (p. 495).

Topics

- Greengrass components for client device support (p. 489)
- Configure cloud discovery (console) (p. 490)
- Configure cloud discovery (AWS CLI) (p. 490)
- Associate client devices (p. 491)
- Manage core device endpoints (p. 492)
- Test client device communications (p. 495)
- Greengrass discovery RESTful API (p. 502)

Greengrass components for client device support

To enable client devices to connect and communicate with a core device, you deploy the following Greengrass components to the core device:
Configure cloud discovery (console)

You can use the AWS IoT Greengrass console to associate client devices, manage core device endpoints, and deploy components to enable client device support. For more information, see Enable client device support (p. 479).

Configure cloud discovery (AWS CLI)

You can use the AWS Command Line Interface (AWS CLI) to associate client devices, manage core device endpoints, and deploy components to enable client device support. For more information, see the following:

- Manage client device associations (AWS CLI) (p. 492)
- Manage core device endpoints (p. 492)
- AWS-provided client device components (p. 488)
- Create deployments (p. 384)
Associate client devices

To use cloud discovery, associate client devices with a core device so that they can discover the core device. Then, they can use the Greengrass discovery API (p. 502) to retrieve connectivity information and certificates for their associated core devices.

 Likewise, disassociate client devices from a core device to stop them from discovering the core device.

Topics

• Manage client device associations (console) (p. 491)
• Manage client device associations (AWS CLI) (p. 492)
• Manage client device associations (API) (p. 492)

Manage client device associations (console)

You can use the AWS IoT Greengrass console to view, add, and delete client device associations.

To view client device associations for a core device (console)

1. Navigate to the AWS IoT Greengrass console.
2. Choose Core devices.
3. Choose the core device to manage.
4. On the core device's details page, choose the Client devices tab.
5. In the Associated client devices section, you can see which client devices (AWS IoT things) are associated with the core device.

To associate client devices with a core device (console)

1. Navigate to the AWS IoT Greengrass console.
2. Choose Core devices.
3. Choose the core device to manage.
4. On the core device's details page, choose the Client devices tab.
5. In the Associated client devices section, choose Associate client devices.
6. In the Associate client devices with core device modal, do the following for each client device to associate:
   a. Enter the name of the AWS IoT thing to associate as a client device.
   b. Choose Add.
7. Choose Associate.

The client devices that you associated can now use the Greengrass discovery API to discover this core device.

To disassociate client devices from a core device (console)

1. Navigate to the AWS IoT Greengrass console.
2. Choose Core devices.
3. Choose the core device to manage.
4. On the core device's details page, choose the Client devices tab.
5. In the **Associated client devices** section, select each client device to disassociate.
6. Choose **Disassociate**.
7. In the confirmation modal, choose **Disassociate**.

The client devices that you disassociated can no longer use the Greengrass discovery API to discover this core device.

### Manage client device associations (AWS CLI)

You can use the AWS Command Line Interface (AWS CLI) to manage client device associations for a core device.

**To view client device associations for a core device (AWS CLI)**

- Use the following command: `list-client-devices-associated-with-core-device`.

**To associate client devices with a core device (AWS CLI)**

- Use the following command: `batch-associate-client-device-with-core-device`.

**To disassociate client devices from a core device (AWS CLI)**

- Use the following command: `batch-disassociate-client-device-from-core-device`.

### Manage client device associations (API)

You can use the AWS API to manage client device associations for a core device.

**To view client device associations for a core device (AWS API)**

- Use the following operation: `ListClientDevicesAssociatedWithCoreDevice`.

**To associate client devices with a core device (AWS API)**

- Use the following operation: `BatchAssociateClientDeviceWithCoreDevice`.

**To disassociate client devices from a core device (AWS API)**

- Use the following operation: `BatchDisassociateClientDeviceFromCoreDevice`.

### Manage core device endpoints

When you use cloud discovery, you store MQTT broker endpoints for core devices in the AWS IoT Greengrass cloud service. Client devices connect to AWS IoT Greengrass to retrieve these endpoints and other information for their associated core devices.

For each core device, you can choose to automatically or manually manage endpoints.

- **Automatically manage endpoints with IP detector**

  You can deploy the IP detector component (p. 177) to automatically manage core device endpoints for you if you have a non-complex network setup, such as where the client devices are on the same network.
network as the core device. You can't use the IP detector component if the core device is behind a router that forwards the MQTT broker port to the core device, for example.

The IP detector component is also useful if you deploy to thing groups, because it manages the endpoints for all core devices in the thing group. For more information, see Use IP detector to automatically manage endpoints (p. 493).

**Note**
If you use the IP detector component to manage endpoints, you must use the default port 8883 for the Moquette MQTT broker component (p. 283).

- **Manually manage endpoints**

If you can't use the IP detector component, you must manually manage core device endpoints. You can update these endpoints with the console or the API. For more information, see Manually manage endpoints (p. 493).

**Topics**
- Use IP detector to automatically manage endpoints (p. 493)
- Manually manage endpoints (p. 493)

### Use IP detector to automatically manage endpoints

If you have a simple network setup, such as the client devices on the same network as the core device, you can deploy the IP detector component (p. 177) to do the following:

- Monitor the Greengrass core device's local network connectivity information. This information includes the core device's network endpoints and the port where the MQTT broker operates.
- Report the core device's connectivity information to the AWS IoT Greengrass cloud service.

The core device's AWS IoT policy must allow the `greengrass:UpdateConnectivityInfo` permission to use the IP detector component. For more information, see AWS IoT policies for data plane operations (p. 611) and Minimal AWS IoT policy to support client devices (p. 616).

The IP detector component overwrites endpoints that you set manually.

You can do either of the following to deploy the IP detector component:

- Use the Configure discovery page in the console. For more information, see Configure cloud discovery (console) (p. 490).
- Create and revise deployments to include the IP detector. You can use the console, AWS CLI, or AWS API to manage deployments. For more information, see Create deployments (p. 384).

### Manually manage endpoints

You can manually manage MQTT broker endpoints for core devices.

Each MQTT broker endpoint has the following information:

**Endpoint** *(HostAddress)*

An IP address or DNS address where client devices can connect to an MQTT broker on the core device.

**Port** *(PortNumber)*

The port where the MQTT broker operates on the core device.
You can configure this port on the Moquette MQTT broker component (p. 283), which defaults to use port 8883.

**Metadata**

Additional metadata to provide to client devices that connect to this endpoint.

**Topics**

- Manage endpoints (console) (p. 494)
- Manage endpoints (AWS CLI) (p. 494)
- Manage endpoints (API) (p. 494)

**Manage endpoints (console)**

You can use the AWS IoT Greengrass console to view, update, and remove endpoints for a core device.

**To manage endpoints for a core device (console)**

1. Navigate to the AWS IoT Greengrass console.
2. Choose Core devices.
3. Choose the core device to manage.
4. On the core device's details page, choose the Client devices tab.
5. In the MQTT broker endpoints section, you can see the core device's MQTT broker endpoints. Choose Manage endpoints.
6. In the Manage endpoints modal, add or remove MQTT broker endpoints for the core device.
7. Choose Update.

**Manage endpoints (AWS CLI)**

You can use the AWS Command Line Interface (AWS CLI) to manage endpoints for a core device.

**Note**

Because client device support in AWS IoT Greengrass V2 is backward compatible with AWS IoT Greengrass V1, you use AWS IoT Greengrass V1 API operations to manage core device endpoints.

**To get endpoints for a core device (AWS CLI)**

- Use the following command: get-connectivity-info.

**To update endpoints for a core device (AWS CLI)**

- Use the following command: update-connectivity-info.

**Manage endpoints (API)**

You can use the AWS API to manage endpoints for a core device.

**Note**

Because client device support in AWS IoT Greengrass V2 is backward compatible with AWS IoT Greengrass V1, you use AWS IoT Greengrass V1 API operations to manage endpoints.

**To get endpoints for a core device (AWS API)**

- Use the following operation: GetConnectivityInfo.
To update endpoints for a core device (AWS API)

- Use the following command: `UpdateConnectivityInfo`.

Test client device communications

Client devices can use the AWS IoT Device SDK to discover, connect, and communicate with a core device. You can use the Greengrass discovery client in the AWS IoT Device SDK to use the Greengrass discovery API (p. 502), which returns information about core devices to which a client device can connect. The API response includes MQTT broker endpoints to connect and certificates to use to verify the identity of each core device. Then, the client device can try each endpoint until it successfully connects to a core device.

Client devices can discover only core devices to which you associate them. Before you test communications between a client device and a core device, you must associate the client device to the core device. For more information, see Associate client devices (p. 491).

The Greengrass discovery API returns the core device MQTT broker endpoints that you specify. You can use the IP detector component (p. 177) to manage these endpoints for you, or you can manually manage them for each core device. For more information, see Manage core device endpoints (p. 492).

**Note**

To use the Greengrass discovery API, a client device must have the `greengrass:Discover` permission. For more information, see Minimal AWS IoT policy for client devices (p. 617).

The AWS IoT Device SDK is available in multiple programming languages. For more information, see AWS IoT Device SDKs in the AWS IoT Core Developer Guide.

**Topics**

- Test communications (Python) (p. 495)
- Test communications (C++) (p. 497)
- Test communications (JavaScript) (p. 499)
- Test communications (Java) (p. 501)

Test communications (Python)

In this section, you use Greengrass discovery sample in the AWS IoT Device SDK v2 for Python to test communications between a client device and a core device.

**Important**

To use the AWS IoT Device SDK v2 for Python, a device must run Python 3.6 or later.

**To test communications (AWS IoT Device SDK v2 for Python)**

1. Download and install the AWS IoT Device SDK v2 for Python to the AWS IoT thing to connect as a client device.

   On the client device, do the following:

   a. Clone the AWS IoT Device SDK v2 for Python repository to download it.

      ```bash
      git clone https://github.com/aws/aws-iot-device-sdk-python-v2.git
      ```

   b. Install the AWS IoT Device SDK v2 for Python.

      ```bash
      python3 -m pip install --user ./aws-iot-device-sdk-python-v2
      ```

2. Change to the samples folder in the AWS IoT Device SDK v2 for Python.
3. Run the sample Greengrass discovery application. This application expects arguments that specify the client device thing name, the MQTT topic and message to use, and the certificates that authenticate and secure the connection. The following example sends a Hello World message to the clients/MyClientDevice1/hello/world topic.

- Replace both instances of **MyClientDevice1** with the client device's thing name.
- Replace `~/certs/AmazonRootCA1.pem` with the path to the Amazon root CA certificate on the client device.
- Replace `~/certs/device.pem.crt` with the path to the device certificate on the client device.
- Replace `~/certs/private.pem.key` with the path to the private key file on the client device.
- Replace `us-east-1` with the AWS Region where your client device and core device operate.

```python3
python3 basic_discovery.py \
  --thing-name MyClientDevice1 \ 
  --topic 'clients/MyClientDevice1/hello/world' \ 
  --message 'Hello World!' \ 
  --root-ca ~/certs/AmazonRootCA1.pem \ 
  --cert ~/certs/device.pem.crt \ 
  --key ~/certs/private.pem.key \ 
  --region us-east-1 \ 
  --verbosity Warn
```

The discovery sample application sends the message 10 times and disconnects. It also subscribes to the same topic where it publishes messages. If the output indicates that the application received MQTT messages on the topic, the client device can successfully communicate with the core device.

```
Performing greengrass discovery...
awsiot.greengrass_discovery.DiscoverResponse(gg_groups=[awsiot.greengrass_discovery.GGGroup(gg_group_id='greengrassV2-coreDevice-MyGreengrassCore',
  cores=[awsiot.greengrass_discovery.GGCore(thing_arn='arn:aws:iot:us-east-1:123456789012:thing/MyGreengrassCore',
    connectivity=[awsiot.greengrass_discovery.ConnectivityInfo(id='203.0.113.0',
      host_address='203.0.113.0', metadata='', port=8883))],
  certificateAuthorities=['-----BEGIN CERTIFICATE-----
  MIICiT...EXAMPLE=
  -----END CERTIFICATE-----'])
Trying core arn:aws:iot:us-east-1:123456789012:thing/MyGreengrassCore at host 203.0.113.0 port 8883 Connected!
Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!", "sequence": 0}
Publish received on topic clients/MyClientDevice1/hello/world b'"message": "Hello World!", "sequence": 0'}
Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!", "sequence": 1}
Publish received on topic clients/MyClientDevice1/hello/world b'"message": "Hello World!", "sequence": 1'}

...
If the application outputs an error instead, see Troubleshooting Greengrass discovery issues (p. 510).

You can also view the Greengrass logs on the core device to verify if the client device successfully connects and sends messages. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

Test communications (C++)

In this section, you use Greengrass discovery sample in the AWS IoT Device SDK v2 for C++ to test communications between a client device and a core device.

Important
To build the AWS IoT Device SDK v2 for C++, a device must have the following tools:

- C++ 11 or later
- CMake 3.1 or later
- One of the following compilers:
  - GCC 4.8 or later
  - Clang 3.9 or later
  - MSVC 2015 or later

To test communications (AWS IoT Device SDK v2 for C++)

1. Download and build the AWS IoT Device SDK v2 for C++ to the AWS IoT thing to connect as a client device.

   On the client device, do the following:
   
   a. Create a folder for the AWS IoT Device SDK v2 for C++ workspace, and change to it.

   ```
   cd
   mkdir iot-device-sdk-cpp
   cd iot-device-sdk-cpp
   ```

   b. Clone the AWS IoT Device SDK v2 for C++ repository to download it. The `--recursive` flag specifies to download submodules.

   ```
   git clone --recursive https://github.com/aws/aws-iot-device-sdk-cpp-v2.git
   ```

   c. Create a folder for the AWS IoT Device SDK v2 for C++ build output, and change to it.

   ```
   mkdir aws-iot-device-sdk-cpp-v2-build
   cd aws-iot-device-sdk-cpp-v2-build
   ```

   d. Build the AWS IoT Device SDK v2 for C++.

   ```
   cmake -DCMAKE_INSTALL_PREFIX="~/iot-device-sdk-cpp" -DCMAKE_BUILD_TYPE="Release" ../aws-iot-device-sdk-cpp
   cmake --build . --target install
   ```

2. Build the Greengrass discovery sample application in the AWS IoT Device SDK v2 for C++. Do the following:

   a. Change to the Greengrass discovery sample folder in the AWS IoT Device SDK v2 for C++.
b. Create a folder for the Greengrass discovery sample build output, and change to it.

```bash
mkdir build
cd build
```

c. Build the Greengrass discovery sample application.

```bash
cmake -DCMAKE_PREFIX_PATH="~/iot-device-sdk-cpp" -DCMAKE_BUILD_TYPE="Release" ..
cmake --build . --config "Release"
```

3. Run the sample Greengrass discovery application. This application expects arguments that specify the client device thing name, the MQTT topic to use, and the certificates that authenticate and secure the connection. The following example subscribes to the clients/MyClientDevice1/hello/world topic and publishes a message that you enter on the command line to the same topic.

- Replace both instances of `MyClientDevice1` with the client device's thing name.
- Replace `~/certs/AmazonRootCA1.pem` with the path to the Amazon root CA certificate on the client device.
- Replace `~/certs/device.pem.crt` with the path to the device certificate on the client device.
- Replace `~/certs/private.pem.key` with the path to the private key file on the client device.
- Replace `us-east-1` with the AWS Region where your client device and core device operate.

```bash
./basic-discovery 
  --thing_name MyClientDevice1 
  --topic 'clients/MyClientDevice1/hello/world' 
  --ca_file ~/certs/AmazonRootCA1.pem 
  --cert ~/certs/device.pem.crt 
  --key ~/certs/private.pem.key 
  --region us-east-1
```

The discovery sample application subscribes to the topic and prompts you to enter a message to publish.

```
Connecting to group greengrassV2-coreDevice-MyGreengrassCore with thing arn arn:aws:iot:us-east-1:123456789012:thing/MyGreengrassCore, using endpoint 203.0.113.0:8883
Connected to group greengrassV2-coreDevice-MyGreengrassCore, using connection to 203.0.113.0:8883
Successfully subscribed to clients/MyClientDevice1/hello/world
```

Enter the message you want to publish to topic clients/MyClientDevice1/hello/world and press enter. Enter 'exit' to exit this program.

If the application outputs an error instead, see Troubleshooting Greengrass discovery issues (p. 510).

4. Enter a message, such as **Hello World!**.

```
Enter the message you want to publish to topic clients/MyClientDevice1/hello/world and press enter. Enter 'exit' to exit this program.
Hello World!
```
If the output indicates that the application received the MQTT message on the topic, the client device can successfully communicate with the core device.

| Operation on packetId 2 Succeeded  
| Publish received on topic clients/MyClientDevice1/hello/world  
| Message: Hello World! |

You can also view the Greengrass logs on the core device to verify if the client device successfully connects and sends messages. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

**Test communications (JavaScript)**

In this section, you use Greengrass discovery sample in the AWS IoT Device SDK v2 for JavaScript to test communications between a client device and a core device.

**Important**

To use the AWS IoT Device SDK v2 for JavaScript, a device must run Node v10.0 or later.

**To test communications (AWS IoT Device SDK v2 for JavaScript)**

1. Download and install the AWS IoT Device SDK v2 for JavaScript to the AWS IoT thing to connect as a client device.
   
   On the client device, do the following:
   
   a. Clone the AWS IoT Device SDK v2 for JavaScript repository to download it.
      
      ```
      git clone https://github.com/aws/aws-iot-device-sdk-js-v2.git
      ```
   
   b. Install the AWS IoT Device SDK v2 for JavaScript.
      
      ```
      cd aws-iot-device-sdk-js-v2
      npm install
      ```

2. Change to the Greengrass discovery sample folder in the AWS IoT Device SDK v2 for JavaScript.
   
   ```
   cd samples/node/basic_discovery
   ```

3. Install the Greengrass discovery sample application.
   
   ```
   npm install
   ```

4. Run the sample Greengrass discovery application. This application expects arguments that specify the client device thing name, the MQTT topic and message to use, and the certificates that authenticate and secure the connection. The following example sends a Hello World message to the clients/MyClientDevice1/hello/world topic.
   
   ```
   • Replace both instances of MyClientDevice1 with the client device's thing name.
   • Replace ~/certs/AmazonRootCA1.pem with the path to the Amazon root CA certificate on the client device.
   • Replace ~/certs/device.pem.crt with the path to the device certificate on the client device.
   • Replace ~/certs/private.pem.key with the path to the private key file on the client device.
   • Replace us-east-1 with the AWS Region where your client device and core device operate.
   ```
node dist/index.js \
  --thing_name MyClientDevice1 \
  --topic 'clients/MyClientDevice1/hello/world' \
  --message 'Hello World!' \
  --ca_file ~/certs/AmazonRootCA1.pem \
  --cert ~/certs/device.pem.crt \
  --key ~/certs/private.pem.key 
  --region us-east-1 
  --verbose warn

The discovery sample application sends the message 10 times and disconnects. It also subscribes
to the same topic where it publishes messages. If the output indicates that the application received
MQTT messages on the topic, the client device can successfully communicate with the core device.

Discovery Response:
{"gg_groups":[{"gg_group_id":"greengrassV2-coreDevice-
MyGreengrassCore","cores":[{"thing_arn":"arn:aws:iot:us-
east-1:123456789012:thing/MyGreengrassCore","connectivity":
[{"id":"203.0.113.0","host_address":"203.0.113.0","port":8883,"metadata":""}]},"certificate_authorities":
["-----BEGIN CERTIFICATE-----
MIICiT...EXAMPLE=
"-----END CERTIFICATE----\n"]}
Trying
endpoint={"id":"203.0.113.0","host_address":"203.0.113.0","port":8883,"metadata":""}
[WARN] [2021-06-12T00:46:45Z] [00007f90c0e8d700] [socket] - id=0x7f90b8018710 fd=26: setsockopt() for NO_SIGNAL failed with errno 92. If you are having SIGPIPE signals
thrown, you may want to install a signal trap in your application layer.
Connected to
epoint={"id":"203.0.113.0","host_address":"203.0.113.0","port":8883,"metadata":""}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
  retain:false
  {"message":"Hello World!","sequence":1}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
  retain:false
  {"message":"Hello World!","sequence":2}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
  retain:false
  {"message":"Hello World!","sequence":3}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
  retain:false
  {"message":"Hello World!","sequence":4}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
  retain:false
  {"message":"Hello World!","sequence":5}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
  retain:false
  {"message":"Hello World!","sequence":6}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
  retain:false
  {"message":"Hello World!","sequence":7}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
  retain:false
  {"message":"Hello World!","sequence":8}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
  retain:false
  {"message":"Hello World!","sequence":9}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
  retain:false
  {"message":"Hello World!","sequence":10}
Complete!
If the application outputs an error instead, see Troubleshooting Greengrass discovery issues (p. 510).

You can also view the Greengrass logs on the core device to verify if the client device successfully connects and sends messages. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

Test communications (Java)

In this section, you use Greengrass discovery sample in the AWS IoT Device SDK v2 for Java to test communications between a client device and a core device.

Important
To build the AWS IoT Device SDK v2 for Java, a device must have the following tools:

- Java 8 or later, with JAVA_HOME pointing to the Java folder.
- Apache Maven

To test communications (AWS IoT Device SDK v2 for Java)

1. Download and build the AWS IoT Device SDK v2 for Java to the AWS IoT thing to connect as a client device.

   On the client device, do the following:

   a. Clone the AWS IoT Device SDK v2 for Java repository to download it.

   ```
   git clone https://github.com/aws/aws-iot-device-sdk-java-v2.git
   ```

   b. Change to the AWS IoT Device SDK v2 for Java folder.

   ```
   cd aws-iot-device-sdk-java-v2
   ```

   c. Build the AWS IoT Device SDK v2 for Java.

   ```
   mvn versions:use-latest-versions -Dincludes="software.amazon.awssdk.crt*"
   mvn clean install
   ```

2. Run the sample Greengrass discovery application. This application expects arguments that specify the client device thing name, the MQTT topic to use, and the certificates that authenticate and secure the connection. The following example subscribes to the `clients/MyClientDevice1/hello/world` topic and publishes a message that you enter on the command line to the same topic.

   - Replace both instances of `MyClientDevice1` with the client device's thing name.
   - Replace `~certs/AmazonRootCA1.pem` with the path to the Amazon root CA certificate on the client device.
   - Replace `~certs/device.pem.crt` with the path to the device certificate on the client device.
   - Replace `~certs/private.pem.key` with the path to the private key file on the client device.
   - Replace `us-east-1` with the AWS Region where your client device and core device operate.

   ```
   DISCOVERY_SAMPLE_ARGS="--thingName MyClientDevice1 \ 
   --topic 'clients/MyClientDevice1/hello/world' \ 
   --rootca ~/certs/AmazonRootCA1.pem \ 
   ```
The discovery sample application subscribes to the topic and prompts you to enter a message to publish.

```java
mvn exec:java -pl samples/Greengrass \
-Dexec.mainClass=greengrass.BasicDiscovery \
-Dexec.args="$DISCOVERY_SAMPLE_ARGS"
```

3. Enter a message, such as **Hello World!**.

If the application outputs an error instead, see Troubleshooting Greengrass discovery issues (p. 510).

You can also view the Greengrass logs on the core device to verify if the client device successfully connects and sends messages. For more information, see View AWS IoT Greengrass Core software logs (p. 751).

### Greengrass discovery RESTful API

AWS IoT Greengrass provides the **Discover** API operation that client devices can use to identify Greengrass core devices where they can connect. Client devices use this data plane operation to retrieve information required to connect to Greengrass core devices where you associate them with the **BatchAssociateClientDeviceWithCoreDevice** API operation. When a client device comes online, it can connect to the AWS IoT Greengrass cloud service and use the discovery API to find:

- The IP address and port for each associated Greengrass core device.
- The core device CA certificate, which client devices can use to authenticate the Greengrass core device.

**Note**

Client devices can also use the discovery client in the AWS IoT Device SDK to discover connectivity information for Greengrass core devices. The discovery client uses the discovery API. For more information, see the following:

- Test client device communications (p. 495)
- Greengrass Discovery RESTful API in the *AWS IoT Greengrass Version 1 Developer Guide.*
To use this API operation, send HTTP requests to the discovery API on the Greengrass data plane endpoint. This API endpoint has the following format.

https://greengrass-ats.iot.region.amazonaws.com:port/greengrass/discover/thing/thing-name

For a list of supported AWS Regions and endpoints for the AWS IoT Greengrass discovery API, see AWS IoT Greengrass V2 endpoints and quotas in the AWS General Reference. This API operation is available only on the Greengrass data plane endpoint. The control plane endpoint that you use to manage components and deployments is different from the data plane endpoint.

**Note**
The discovery API is the same for AWS IoT Greengrass V1 and AWS IoT Greengrass V2. If you have client devices that connect to a AWS IoT Greengrass V1 core, you can connect them to AWS IoT Greengrass V2 core devices without changing the code on the client devices. For more information, see Greengrass Discovery RESTful API in the AWS IoT Greengrass Version 1 Developer Guide.

**Topics**
- Discovery authentication and authorization (p. 503)
- Request (p. 504)
- Response (p. 504)
- Example discover response documents (p. 504)
- Test the discovery API with cURL (p. 506)

**Discovery authentication and authorization**
To use the discovery API to retrieve connectivity information, a client device must use TLS mutual authentication with an X.509 client certificate to authenticate. For more information, see X.509 client certificates in the AWS IoT Core Developer Guide.

A client device must also have permission to perform the greengrass:Discover action. The following example AWS IoT policy allows an AWS IoT thing named MyClientDevice1 to perform Discover for itself.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": "greengrass:Discover",
         "Resource": [
            "arn:aws:iot:us-west-2:123456789012:thing/MyClientDevice1"
         ]
      }
   ]
}
```

**Important**
**Thing policy variables** (iot:Connection.Thing.*) aren't supported for in AWS IoT policies for core devices or Greengrass data plane operations. Instead, you can use a wildcard that matches multiple devices that have similar names. For example, you can specify MyGreengrassDevice* to match MyGreengrassDevice1, MyGreengrassDevice2, and so on.

For more information, see AWS IoT Core policies in the AWS IoT Core Developer Guide.
Request

The request contains the standard HTTP headers and is sent to the Greengrass discovery endpoint, as shown in the following examples.

The port number depends on whether the core device is configured to send HTTPS traffic over port 8443 or port 443. For more information, see the section called "Connect on port 443 or through a network proxy" (p. 125).

Note

These examples use the Amazon Trust Services (ATS) endpoint, which works with the recommended ATS root CA certificates. Endpoints must match the root CA certificate type.

Port 8443

HTTP GET https://greengrass-ats.iot.region.amazonaws.com:8443/greengrass/discover/thing/thing-name

Port 443

HTTP GET https://greengrass-ats.iot.region.amazonaws.com:443/greengrass/discover/thing/thing-name

Note

Clients that connect on port 443 must implement the Application Layer Protocol Negotiation (ALPN) TLS extension and pass x-amzn-http-ca as the ProtocolName in the ProtocolNameList. For more information, see Protocols in the AWS IoT Developer Guide.

Response

Upon success, the response includes the standard HTTP headers plus the following code and body:

HTTP 200
BODY: response document

For more information, see Example discover response documents (p. 504).

Example discover response documents

Because AWS IoT Greengrass V2 uses the same discovery API as AWS IoT Greengrass V1, the response organizes information according to AWS IoT Greengrass V1 concepts, such as Greengrass groups. The response contains a list of Greengrass groups. In AWS IoT Greengrass V2, each core device is in its own group, where the group contains only that core device and its connectivity information.

The following document shows the response for a client device that is associated to one Greengrass core device. The core device has one endpoint and one CA certificate.

```json
{
  "GGGroups": [
    {
      "GGGroupId": "greengrassV2-coreDevice-core-device-01-thing-name",
      "Cores": [
        {
          "thingArn": "core-device-01-thing-arn",
          "Connectivity": [
```
The following document shows the response for a client device that is associated to two core devices. The core devices have multiple endpoints and multiple group CA certificates.

```json
{
  "GGGroups": [
    {
      "GGGroupId": "greengrassV2-coreDevice-core-device-01-thing-name",
      "Cores": [
        {
          "thingArn": "core-device-01-thing-arn",
          "Connectivity": [
            {
              "id": "core-device-01-connection-id",
              "hostAddress": "core-device-01-address",
              "portNumber": "core-device-01-port",
              "metadata": "core-device-01-description"
            },
            {
              "id": "core-device-01-connection-id-2",
              "hostAddress": "core-device-01-address-2",
              "portNumber": "core-device-01-port-2",
              "metadata": "core-device-01-connection-2-description"
            }
          ]
        }
      ],
      "CAs": ["-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----", "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----", "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"]
    },
    {
      "GGGroupId": "greengrassV2-coreDevice-core-device-02-thing-name",
      "Cores": [
        {
          "thingArn": "core-device-02-thing-arn",
          "Connectivity": [
            {
              "id": "core-device-02-connection-id",
              "hostAddress": "core-device-02-address",
              "portNumber": "core-device-02-port",
              "metadata": "core-device-02-connection-1-description"
            }
          ]
        }
      ],
      "CAs": ["-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"]
    }
  ]
}
```
Test the discovery API with cURL

If you have cURL installed, you can test the discovery API. The following example specifies a client device's certificates to authenticate a request to the Greengrass discovery API endpoint.

```bash
curl -i \
   --cert 1a23bc4d56.cert.pem \ 
   --key 1a23bc4d56.private.key \ 
   https://greengrass-ats.iot.us-west-2.amazonaws.com:8443/greengrass/discover/thing/MyClientDevice1
```

**Note**
The `-i` argument specifies to output HTTP response headers. You can use this option to help identify errors.

If the request succeeds, this command outputs a response similar to the following example.

```json
{
   "GGGroups": [ 
      { 
         "GGGroupId": "greengrassV2-coreDevice-MyGreengrassCore", 
         "Cores": [ 
            { 
               "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore", 
               "Connectivity": [ 
                  { 
                     "Id": "AUTOIP_192.168.1.4_1", 
                     "HostAddress": "192.168.1.5", 
                     "PortNumber": 8883, 
                     "Metadata": ""
                  }
               ]
            }
         ], 
         "CAs": [ 
            "-----BEGIN CERTIFICATE-----
            cert-contents
            -----END CERTIFICATE-----"
         ]
      ]
   }
}
```

If the command outputs an error, see Troubleshooting Greengrass discovery issues (p. 510).

Relay MQTT messages between client devices and AWS IoT Core

You can relay MQTT messages and other data between client devices and AWS IoT Core. Client devices connect to the MQTT broker component that runs on the core device. By default, core devices don't relay
MQTT messages or data between client devices and AWS IoT Core. Client devices can communicate only with each other over MQTT by default.

To relay MQTT messages between client devices and AWS IoT Core, configure the MQTT bridge component (p. 280) to do the following:

- Relay messages from client devices to AWS IoT Core.
- Relay messages from AWS IoT Core to client devices.

**Note**
The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings (p. 128).

**Topics**
- Configure and deploy the MQTT bridge component (p. 507)
- Relay MQTT messages (p. 508)

## Configure and deploy the MQTT bridge component

The MQTT bridge component consumes a list of topic mappings that each specify a message source and a message destination. To relay messages between client devices and AWS IoT Core, deploy the MQTT bridge component, and specify each source and destination topic in the component configuration.

To deploy the MQTT bridge component to a core device or group of core devices, create a deployment (p. 384) that includes the `aws.greengrass.clientdevices.mqtt.Bridge` component. Specify the topic mappings, `mqttTopicMapping` in the MQTT bridge component configuration in the deployment.

The following example defines a deployment that configures the MQTT bridge component to relay messages on topics that match the `clients/+/hello/world` topic filter from client devices to AWS IoT Core. The `merge` configuration update requires a serialized JSON object. For more information, see Update component configurations (p. 390).

**Console**

```json
{
   "mqttTopicMapping": {
      "HelloWorldIotCore": {
         "topic": "clients/+/hello/world",
         "source": "LocalMqtt",
         "target": "IotCore"
      }
   }
}
```

**AWS CLI**

```json
{
   "components": {
      "aws.greengrass.clientdevices.mqtt.Bridge": {
         "version": "2.0.0",
         "configurationUpdate": {
```
Relay MQTT messages

To relay MQTT messages between client devices and AWS IoT Core, configure and deploy the MQTT Bridge component (p. 507) and specify the topics to relay.

Example Example: Relay messages on a topic from client devices to AWS IoT Core

The following MQTT bridge component configuration specifies relaying messages on topics that match the `clients/+/hello/world/event` topic filter from client devices to AWS IoT Core.

```json
{
  "mqttTopicMapping": {
    "HelloWorldEvent": {
      "topic": "clients/+/hello/world/event",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

Example Example: Relay messages on a topic from AWS IoT Core to client devices

The following MQTT bridge component configuration specifies relaying messages on topics that match the `clients/+/hello/world/event/response` topic filter from AWS IoT Core to client devices.

```json
{
  "mqttTopicMapping": {
    "HelloWorldEventConfirmation": {
      "topic": "clients/+/hello/world/event/response",
      "source": "IotCore",
      "target": "LocalMqtt"
    }
  }
}
```

Interact with client devices in components

You can develop custom Greengrass components that interact with client devices connected to a core device. For example, you can develop components that do the following:

- Act on MQTT messages from client devices and send data to AWS Cloud destinations.
- Send MQTT messages to client devices to initiate actions.

Client devices connect to and communicate with a core device through the MQTT broker component that runs on the core device. By default, client devices can communicate only with each other over MQTT, and Greengrass components can't receive these MQTT messages or send messages to client devices.
Greengrass components use the local publish/subscribe interface (p. 410) to communicate on a core device. To communicate with client devices in Greengrass components, configure the MQTT bridge component (p. 280) to do the following:

- Relay MQTT messages from client devices to local publish/subscribe.
- Relay MQTT messages from local publish/subscribe to client devices.

Topics
- Configure and deploy the MQTT bridge component (p. 509)
- Receive MQTT messages from client devices (p. 510)
- Send MQTT messages to client devices (p. 510)

Configure and deploy the MQTT bridge component

The MQTT bridge component consumes a list of topic mappings that each specify a message source and a message destination. To communicate with client devices, deploy the MQTT bridge component, and specify each source and destination topic in the component configuration.

To deploy the MQTT bridge component to a core device or group of core devices, create a deployment (p. 384) that includes the `aws.greengrass.clientdevices.mqtt.Bridge` component. Specify the topic mappings, `mqttTopicMapping` in the MQTT bridge component configuration in the deployment.

The following example defines a deployment that configures the MQTT bridge component to relay the `clients/MyClientDevice1/hello/world` topic from client devices to local publish/subscribe broker. The merge configuration update requires a serialized JSON object. For more information, see Update component configurations (p. 390).

**Console**

```json
{
  "mqttTopicMapping": {
    "HelloWorldPubsub": {
      "topic": "clients/MyClientDevice1/hello/world",
      "source": "LocalMqtt",
      "target": "Pubsub"
    }
  }
}
```

**AWS CLI**

```json
{
  "components": {
    "aws.greengrass.clientdevices.mqtt.Bridge": {
      "version": "2.0.0",
      "configurationUpdate": {
        "merge": "\"mqttTopicMapping\":\"HelloWorldPubsub\":\"clients/MyClientDevice1/hello/world\",\"source\":\"LocalMqtt\",\"target\":\"Pubsub\"\"\""
      }
    }
  }
}
```
Receive MQTT messages from client devices

You can subscribe to the local publish/subscribe topics that you configure for the MQTT bridge component to receive messages from client devices.

To receive MQTT messages from client devices in custom components

1. Configure and deploy the MQTT bridge component (p. 509) to relay messages from an MQTT topic where client devices publish to a local publish/subscribe topic.
2. Use the local publish/subscribe IPC interface to subscribe to the topic where the MQTT bridge relays messages. For more information, see Publish/subscribe local messages (p. 410) and SubscribeToTopic (p. 414).

The Connect and test client devices tutorial (p. 477) includes a section where you develop a component that subscribes to messages from a client device. For more information, see Develop a component that interacts with client devices (p. 484).

Send MQTT messages to client devices

You can publish to the local publish/subscribe topics that you configure for the MQTT bridge component to send messages to client devices.

To publish MQTT messages to client devices in custom components

1. Configure and deploy the MQTT bridge component (p. 509) to relay messages from a local publish/subscribe topic to an MQTT topic where client devices subscribe.
2. Use the local publish/subscribe IPC interface to publish to the topic where the MQTT bridge relays messages. For more information, see Publish/subscribe local messages (p. 410) and PublishToTopic (p. 411).

Troubleshooting client devices

Use the troubleshooting information and solutions in this section to help resolve issues with Greengrass client devices and client device components.

Topics
- Greengrass discovery issues (p. 510)

Greengrass discovery issues

Use the following information to troubleshoot issues with Greengrass discovery. These issues can occur when client devices use the Greengrass discovery API (p. 502) to identify a Greengrass core device to which they can connect.

Topics
- Greengrass discovery issues (HTTP API) (p. 511)
- Greengrass discovery issues (AWS IoT Device SDK v2 for Python) (p. 511)
Greengrass discovery issues (HTTP API)

Use the following information to troubleshoot issues with Greengrass discovery. You might see these errors if you test the discovery API with cURL (p. 506).

Topics

- curl: (52) Empty reply from server (p. 511)
- HTTP 403:{"message":null,"traceId":"a1b2c3d4-5678-90ab-cdef-11111EXAMPLE"} (p. 511)
- HTTP 404:{"errorMessage":"The thing provided for discovery was not found"} (p. 511)

curl: (52) Empty reply from server

You might see this error if you specify an inactive AWS IoT certificate in the request.

Check that the client device has an attached certificate, and that the certificate is active. For more information, see Attach a thing or policy to a client certificate and Activate or deactivate a client certificate in the AWS IoT Core Developer Guide.

HTTP 403:{"message":null,"traceId":"a1b2c3d4-5678-90ab-cdef-11111EXAMPLE"}

You might see this error if the client device doesn't have permission to call greengrass:Discover for itself.

Check that the client device's certificate has a policy that allows greengrass:Discover. You can't use thing policy variables (iot:Connection.Thing.*) in the Resource section for this permission. For more information, see Discovery authentication and authorization (p. 503).

HTTP 404:{"errorMessage":"The thing provided for discovery was not found"}

You might see this error if the client device isn't associated to any Greengrass core devices or AWS IoT Greengrass V1 groups.

Check that the client device is associated to the core device to which you want it to connect. For more information, see Associate client devices (p. 491).

Greengrass discovery issues (AWS IoT Device SDK v2 for Python)

Use the following information to troubleshoot issues with Greengrass discovery in the AWS IoT Device SDK v2 for Python.

Topics

- awscrt.exceptions.AwsCrtError: AWS_ERROR_HTTP_CONNECTION_CLOSED: The connection has closed or is closing. (p. 512)
- awsiot.greengrass_discovery.DiscoveryException: (Error during discover call: response_code=403, 403) (p. 512)
- awsiot.greengrass_discovery.DiscoveryException: (Error during discover call: response_code=404, 404) (p. 512)
aws.crt.exceptions.AwsCrtError: AWS_ERROR_HTTP_CONNECTION_CLOSED: The connection has closed or is closing.

You might see this error if you specify an inactive AWS IoT certificate in the request.

Check that the client device has an attached certificate, and that the certificate is active. For more information, see Attach a thing or policy to a client certificate and Activate or deactivate a client certificate in the AWS IoT Core Developer Guide.

aws.iot.greengrass_discovery.DiscoveryException: ('Error during discover call: response_code=403', 403)

You might see this error if the client device doesn't have permission to call greengrass:Discover for itself.

Check that the client device's certificate has a policy that allows greengrass:Discover. You can't use thing policy variables (iot:Connection.Thing.*) in the Resource section for this permission. For more information, see Discovery authentication and authorization (p. 503).

aws.iot.greengrass_discovery.DiscoveryException: ('Error during discover call: response_code=404', 404)

You might see this error if the client device isn't associated to any Greengrass core devices or AWS IoT Greengrass V1 groups.

Check that the client device is associated to the core device to which you want it to connect. For more information, see Associate client devices (p. 491).

Greengrass discovery issues (AWS IoT Device SDK v2 for C++)

Use the following information to troubleshoot issues with Greengrass discovery in the AWS IoT Device SDK v2 for C++.

Topics
- aws-c-http: AWS_ERROR_HTTP_CONNECTION_CLOSED, The connection has closed or is closing. (p. 512)
- aws-c-common: AWS_ERROR_UNKNOWN, Unknown error. (HTTP 403) (p. 512)
- aws-c-common: AWS_ERROR_UNKNOWN, Unknown error. (HTTP 404) (p. 513)

aws-c-http: AWS_ERROR_HTTP_CONNECTION_CLOSED, The connection has closed or is closing.

You might see this error if you specify an inactive AWS IoT certificate in the request.

Check that the client device has an attached certificate, and that the certificate is active. For more information, see Attach a thing or policy to a client certificate and Activate or deactivate a client certificate in the AWS IoT Core Developer Guide.

aws-c-common: AWS_ERROR_UNKNOWN, Unknown error. (HTTP 403)

You might see this error if the client device doesn't have permission to call greengrass:Discover for itself.

Check that the client device's certificate has a policy that allows greengrass:Discover. You can't use thing policy variables (iot:Connection.Thing.*) in the Resource section for this permission. For more information, see Discovery authentication and authorization (p. 503).
Greengrass discovery issues

**aws-c-common: AWS_ERROR_UNKNOWN, Unknown error. (HTTP 404)**

You might see this error if the client device isn't associated to any Greengrass core devices or AWS IoT Greengrass V1 groups.

Check that the client device is associated to the core device to which you want it to connect. For more information, see Associate client devices (p. 491).

**Greengrass discovery issues (AWS IoT Device SDK v2 for JavaScript)**

Use the following information to troubleshoot issues with Greengrass discovery in the AWS IoT Device SDK v2 for JavaScript.

**Topics**
- Error: aws-c-http: AWS_ERROR_HTTP_CONNECTION_CLOSED, The connection has closed or is closing. (p. 513)
- Error: Discovery failed (headers: [object Object]) { response_code: 403 } (p. 513)
- Error: Discovery failed (headers: [object Object]) { response_code: 404 } (p. 513)
- Error: Discovery failed (headers: [object Object]) (p. 513)

**Error: aws-c-http: AWS_ERROR_HTTP_CONNECTION_CLOSED, The connection has closed or is closing.**

You might see this error if you specify an inactive AWS IoT certificate in the request.

Check that the client device has an attached certificate, and that the certificate is active. For more information, see Attach a thing or policy to a client certificate and Activate or deactivate a client certificate in the AWS IoT Core Developer Guide.

**Error: Discovery failed (headers: [object Object]) { response_code: 403 }**

You might see this error if the client device doesn't have permission to call greengrass:Discover for itself.

Check that the client device's certificate has a policy that allows greengrass:Discover. You can't use thing policy variables (iot:Connection.Thing.*) in the Resource section for this permission. For more information, see Discovery authentication and authorization (p. 503).

**Error: Discovery failed (headers: [object Object]) { response_code: 404 }**

You might see this error if the client device isn't associated to any Greengrass core devices or AWS IoT Greengrass V1 groups.

Check that the client device is associated to the core device to which you want it to connect. For more information, see Associate client devices (p. 491).

**Error: Discovery failed (headers: [object Object])**

You might see this error (without an HTTP response code) when you run the Greengrass discovery sample. This error can occur for multiple reasons.

- You might see this error if the client device doesn't have permission to call greengrass:Discover for itself.
Check that the client device's certificate has a policy that allows `greengrass:Discover`. You can't use `thing policy variables` (`iot:Connection.Thing.*`) in the Resource section for this permission. For more information, see Discovery authentication and authorization (p. 503).

- You might see this error if the client device isn't associated to any Greengrass core devices or AWS IoT Greengrass V1 groups.

Check that the client device is associated to the core device to which you want it to connect. For more information, see Associate client devices (p. 491).

**Greengrass discovery issues (AWS IoT Device SDK v2 for Java)**

Use the following information to troubleshoot issues with Greengrass discovery in the AWS IoT Device SDK v2 for Java.

**Topics**

- `software.amazon.awssdk.crt.CrtRuntimeException: Error Getting Response Status Code from HttpStream. (aws_last_error: AWS_ERROR_HTTP_DATA_NOT_AVAILABLE(2062), This data is not yet available.)` (p. 514)

- `java.lang.RuntimeException: Error x-amzn-ErrorType(403)` (p. 514)

- `java.lang.RuntimeException: Error x-amzn-ErrorType(404)` (p. 514)

`software.amazon.awssdk.crt.CrtRuntimeException: Error Getting Response Status Code from HttpStream. (aws_last_error: AWS_ERROR_HTTP_DATA_NOT_AVAILABLE(2062), This data is not yet available.)`

You might see this error if you specify an inactive AWS IoT certificate in the request.

Check that the client device has an attached certificate, and that the certificate is active. For more information, see Attach a thing or policy to a client certificate and Activate or deactivate a client certificate in the AWS IoT Core Developer Guide.

`java.lang.RuntimeException: Error x-amzn-ErrorType(403)`

You might see this error if the client device doesn't have permission to call `greengrass:Discover` for itself.

Check that the client device's certificate has a policy that allows `greengrass:Discover`. You can't use `thing policy variables` (`iot:Connection.Thing.*`) in the Resource section for this permission. For more information, see Discovery authentication and authorization (p. 503).

`java.lang.RuntimeException: Error x-amzn-ErrorType(404)`

You might see this error if the client device isn't associated to any Greengrass core devices or AWS IoT Greengrass V1 groups.

Check that the client device is associated to the core device to which you want it to connect. For more information, see Associate client devices (p. 491).
Interact with device shadows

Greengrass devices can interact with AWS IoT device shadows using components. A shadow is a JSON document that stores the current or desired state information for an AWS IoT thing. Shadows can make a device's state available to other AWS IoT Greengrass components whether the device is connected to AWS IoT or not. Each AWS IoT device has its own classic, unnamed shadow. You can also create multiple named shadows for each device.

Devices and services can create, update, and delete cloud shadows by using MQTT and the reserved MQTT shadow topics, HTTP using the Device Shadow REST API, and the AWS CLI for AWS IoT.

The shadow manager (p. 294) component enables your Greengrass components to create, update, and delete local shadows by using the local shadow service (p. 462) and the local publish/subscribe shadow topics. The shadow manager also manages the storage of these local shadow documents on your core device, and handles the synchronization of shadow state information with cloud shadows.

For more information about AWS IoT device shadow concepts, see AWS IoT Device Shadow service in the AWS IoT Developer Guide.

Topics

• Interact with shadows in components (p. 515)
• Sync local device shadows with AWS IoT Core (p. 517)

Interact with shadows in components

You can develop custom components, including Lambda function components, that use the local shadow service to read and modify local shadow documents and perform local state management of connected devices.

Custom components interact with the local shadow service using the AWS IoT Greengrass Core IPC libraries in the AWS IoT Device SDK. The shadow manager (p. 294) component enables the local shadow service on your core device.

To deploy the shadow manager component to a Greengrass core device, create a deployment (p. 384) that includes the aws.greengrass.ShadowManager component.

Note

By default, deploying the shadow manager component enables local shadow operations only. To enable AWS IoT Greengrass to sync shadow state information for core device shadows or any shadows for connected devices to the corresponding cloud shadow documents in AWS IoT Core, you must create a configuration update for the shadow manager component that includes the synchronize parameter. For more information, see Sync local device shadows with AWS IoT Core (p. 517).

Topics

• Retrieve and modify shadow states (p. 515)
• React to shadow state changes (p. 516)

Retrieve and modify shadow states

The shadow IPC operations retrieve and update state information in local shadow documents. The shadow manager component handles the storage of these shadow documents on your core device.
To modify local shadow states

1. Add access control policies to the recipe for your custom component to allow the component to receive messages on local shadow topics. For example, the following example access control policy allows the component to create, update, and delete the classic device shadow and the named shadow myNamedShadow for the device MyThingName.

```json
{
  "accessControl": {
    "aws.greengrass.ShadowManager": {
      "policyId1": {
        "policyDescription": "Allows access to shadows",
        "operations": [
          "aws.greengrass#GetThingShadow",
          "aws.greengrass#UpdateThingShadow",
          "aws.greengrass#DeleteThingShadow"
        ],
        "resources": [
          "$aws/things/MyThingName/shadow",
          "$aws/things/MyThingName/shadow/name/myNamedShadow"
        ]
      }
    }
  }
}
```

2. Use the shadow IPC operations to retrieve and modify shadow state information. For more information about using shadow IPC operations in component code, see Interact with local shadows (p. 462).

React to shadow state changes

Greengrass components use the local publish/subscribe interface to communicate on a core device. To enable a custom component to react to shadow state changes, you can subscribe to the local publish/subscribe topics. This allows the component to receive messages on the local shadow topics, and then act on those messages.

Local shadow topics use the same format as the AWS IoT device shadow MQTT topics. For more information about shadow topics, see Device Shadow MQTT topics in the AWS IoT Developer Guide.

To react to local shadow state changes

1. Add access control policies to the recipe for your custom component to allow the component to receive messages on local shadow topics. For example, the following example access control policy allows the custom com.example.MyShadowReactiveComponent to receive messages on the /update/delta topic for the classic device shadow and the named shadow myNamedShadow for the device MyThingName.

```json
{
  "accessControl": {
    "aws.greengrass.ipc.pubsub": {
      "com.example.MyShadowReactiveComponent:pubsub:1": {
        "policyDescription": "Allows access to shadow pubsub topics",
        "operations": [
          "aws.greengrass#SubscribeToTopic"
        ],
        "resources": [
          "$aws/things/MyThingName/shadow/update/delta",
          "$aws/things/MyThingName/shadow/name/myNamedShadow/update/delta"
        ]
      }
    }
  }
}
```
2. To initiate a custom action in a component, use `SubscribeToTopic` IPC operations to subscribe to the shadow topics on which you want to receive messages. For more information about using local publish/subscribe IPC operations in component code, see `Publish/subscribe local messages (p. 410)`.

3. To invoke a Lambda function, use the event source configuration to provide the name of the shadow topic and specify that it's a local publish/subscribe topic. For information about creating Lambda function components, see `Run AWS Lambda functions (p. 361)`.

Sync local device shadows with AWS IoT Core

The shadow manager component enables AWS IoT Greengrass to sync local device shadow states with AWS IoT Core. You must modify the configuration of the shadow manager component to include the `synchronization` configuration parameter, and specify the AWS IoT thing names for your devices, and the shadows that you want to sync.

When you configure shadow manager to sync shadows, it syncs all state changes for specified shadows, regardless of whether the changes occur in local shadow documents or in cloud shadow documents.

Topics

- Prerequisites (p. 517)
- Configure the shadow manager component (p. 517)
- Sync local shadows (p. 518)

Prerequisites

To sync local shadows to the AWS IoT Core, you must configure the Greengrass core device's AWS IoT policy to allow the following AWS IoT Core shadow policy actions.

- `iot:GetThingShadow`
- `iot:UpdateThingShadow`
- `iot:DeleteThingShadow`

For more information about these AWS IoT Core policies, see AWS IoT Core policy actions in the AWS IoT Developer Guide.

For more information about the minimal AWS IoT policy, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices (p. 614)

Configure the shadow manager component

The shadow manager requires a list of shadow name mappings to sync shadow state information in local shadow documents to cloud shadow documents in AWS IoT Core.

To sync shadow states, create a deployment (p. 384) that includes the `aws.greengrass.ShadowManager` component, and specify the shadows that you want to sync in the `synchronize` configuration parameter in the shadow manager configuration in the deployment.

The following example configuration update instructs the shadow manager component to sync the following shadows with AWS IoT Core:
• The classic shadow for the core device
• The named MyCoreShadow for the core device
• The classic shadow for an IoT thing named MyDevice2
• The named shadows MyShadowA and MyShadowB for a IoT thing named MyDevice1

```json
{
  "synchronize":{
    "coreThing":{
      "classic":true,
      "namedShadows":["MyCoreShadow"]
    },
    "shadowDocuments":[
      {
        "thingName":"MyDevice1",
        "classic":false,
        "namedShadows":[
          "MyShadowA",
          "MyShadowB"
        ]
      },
      {
        "thingName":"MyDevice2",
        "classic":true,
        "namedShadows":[ ]
      }
    ]
  }
}
```

Sync local shadows

When the Greengrass core device is connected to the AWS IoT cloud, the shadow manager performs the following tasks for the shadows that you specify in the component configuration:

• Retrieves the reported state information from the cloud shadow document in AWS IoT Core.
• Updates locally stored shadow documents to synchronize the device state.
• Publishes the device's current state to the cloud shadow document.
Manage data streams on the AWS IoT Greengrass Core

AWS IoT Greengrass stream manager makes it more efficient and reliable to transfer high-volume IoT data to the AWS Cloud. Stream manager processes data streams on the AWS IoT Greengrass Core before it exports them to the AWS Cloud. Stream manager integrates with common edge scenarios, such as machine learning (ML) inference, where the AWS IoT Greengrass Core device processes and analyzes data before it exports the data to the AWS Cloud or local storage destinations.

Stream manager provides a common interface to simplify custom component development so that you don't need to build custom stream management functionality. Your components can use a standardized mechanism to process high-volume streams and manage local data retention policies. You can define policies for storage type, size, and data retention for each stream to control how stream manager processes and exports data.

Stream manager works in environments with intermittent or limited connectivity. You can define bandwidth use, timeout behavior, and how the AWS IoT Greengrass Core handles stream data when it is connected or disconnected. You can also set priorities to control the order in which the AWS IoT Greengrass Core exports streams to the AWS Cloud. This makes it possible for you to handle critical data sooner than other data.

You can configure stream manager to automatically export data to the AWS Cloud for storage or further processing and analysis. Stream manager supports exports to the following AWS Cloud destinations:

- Channels in AWS IoT Analytics. AWS IoT Analytics lets you perform advanced analysis on your data to help make business decisions and improve machine learning models. For more information, see What is AWS IoT Analytics? in the AWS IoT Analytics User Guide.
- Streams in Amazon Kinesis Data Streams. You can use Kinesis Data Streams to aggregate high-volume data and load it into a data warehouse or MapReduce cluster. For more information, see What is Amazon Kinesis Data Streams? in the Amazon Kinesis Data Streams Developer Guide.
- Asset properties in AWS IoT SiteWise. AWS IoT SiteWise lets you collect, organize, and analyze data from industrial equipment at scale. For more information, see What is AWS IoT SiteWise? in the AWS IoT SiteWise User Guide.
- Objects in Amazon Simple Storage Service Amazon S3. You can use Amazon S3 to store and retrieve large amounts of data. For more information, see What is Amazon S3? in the Amazon Simple Storage Service Developer Guide.

Stream management workflow

Your IoT applications interact with stream manager through the Stream Manager SDK.

In a simple workflow, a component on the AWS IoT Greengrass core consumes IoT data, such as time-series temperature and pressure metrics. The component might filter or compress the data, and then call the Stream Manager SDK to write the data to a stream in stream manager. Stream manager can export the stream to the AWS Cloud automatically based on the policies that you define for the stream. Components can also send data directly to local databases or storage repositories.

Your IoT applications can include multiple custom components that read or write to streams. These components can read and write to streams to filter, aggregate, and analyze data on the AWS IoT
Greengrass core device. This makes it possible to respond quickly to local events and extract valuable information before the data transfers from the core to the AWS Cloud or local destinations.

To get started, deploy the stream manager component to your AWS IoT Greengrass core device. In the deployment, configure the stream manager component parameters to define settings that apply to all streams on the Greengrass core device. Use these parameters to control how stream manager stores, processes, and exports streams based on your business needs and environment constraints.

After you configure stream manager, you can create and deploy your IoT applications. These are typically custom components that use StreamManagerClient in the Stream Manager SDK to create and interact with streams. When you create a stream, you can define per-stream policies, such as export destinations, priority, and persistence.

Requirements

The following requirements apply for using stream manager:

- Stream manager requires a minimum of 70 MB RAM in addition to the AWS IoT Greengrass Core software. Your total memory requirement depends on your workload.
- AWS IoT Greengrass components must use the Stream Manager SDK to interact with stream manager. The Stream Manager SDK is available in the following languages:
  - Stream Manager SDK for Java (v1.1.0 or later)
  - Stream Manager SDK for Node.js (v1.1.0 or later)
  - Stream Manager SDK for Python (v1.1.0 or later)
- AWS IoT Greengrass components must specify the stream manager component (aws.greengrass.StreamManager) as a dependency in their recipe to use stream manager.
  
  **Note**
  If you use stream manager to export data to the cloud, you can't upgrade version 2.0.7 of the stream manager component to a version between v2.0.8 and v2.0.11. If you are deploying stream manager for the first time, we strongly recommend that you deploy the latest version of the stream manager component.
- If you define AWS Cloud export destinations for a stream, you must create your export targets and grant access permissions in the Greengrass device role (p. 627). Depending on the destination, other requirements might also apply. For more information, see:
  - the section called “AWS IoT Analytics channels” (p. 546)
  - the section called “Amazon Kinesis data streams” (p. 547)
  - the section called “AWS IoT SiteWise asset properties” (p. 548)
  - the section called “Amazon S3 objects” (p. 550)

You are responsible for maintaining these AWS Cloud resources.

Data security

When you use stream manager, be aware of the following security considerations.

Local data security

AWS IoT Greengrass does not encrypt stream data at rest or in transit between local components on the core device.
- **Data at rest.** Stream data is stored locally in a storage directory. For data security, AWS IoT Greengrass relies on Unix file permissions and full-disk encryption, if enabled. You can use the optional `STREAM_MANAGER_STORE_ROOT_DIR` parameter to specify the storage directory. If you change this parameter later to use a different storage directory, AWS IoT Greengrass does not delete the previous storage directory or its contents.

- **Data in transit locally.** AWS IoT Greengrass does not encrypt stream data in local transit between data sources, AWS IoT Greengrass components, the Stream Manager SDK, and stream manager.

- **Data in transit to the AWS Cloud.** Data streams exported by stream manager to the AWS Cloud use standard AWS service client encryption with Transport Layer Security (TLS).

---

### Client authentication

Stream manager clients use the Stream Manager SDK to communicate with stream manager. When client authentication is enabled, only Greengrass components can interact with streams in stream manager. When client authentication is disabled, any process running on the Greengrass core device can interact with streams in stream manager. You should disable authentication only if your business case requires it.

You use the `STREAM_MANAGER_AUTHENTICATE_CLIENT` parameter to set the client authentication mode. You can configure this parameter when you deploy the stream manager component to core devices.

<table>
<thead>
<tr>
<th>Enabled</th>
<th>Disabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter value</td>
<td><code>true</code> (default and recommended)</td>
</tr>
<tr>
<td>Allowed clients</td>
<td>Greengrass components on the core device</td>
</tr>
</tbody>
</table>

---

### See also

- the section called “Configure stream manager” (p. 555)
- the section called “Use StreamManagerClient to work with streams” (p. 529)
- the section called “Export configurations for supported cloud destinations” (p. 545)

---

### Create custom components that use stream manager

Use stream manager in custom Greengrass components to store, process, and export IoT device data. Use the procedures and examples in this section to create component recipes, artifacts, and applications that work with stream manager. For more information about how to develop and test components, see Create local AWS IoT Greengrass components (p. 324).

#### Topics

- Define component recipes that use stream manager (p. 522)
- Connect to stream manager in application code (p. 528)
Define component recipes that use stream manager

To use stream manager in a custom component, you must define the `aws.greengrass.StreamManager` component as a dependency. You must also provide the Stream Manager SDK. Complete the following tasks to download and use the Stream Manager SDK in the language of your choice.

Use the Stream Manager SDK for Java

The Stream Manager SDK for Java is available as a JAR file that you can use to compile your component. Then, you can create an application JAR that includes the Stream Manager SDK, define the application JAR as a component artifact, and run the application JAR in the component lifecycle.

To use the Stream Manager SDK for Java

1. Download the Stream Manager SDK for Java JAR file.
2. Do one of the following to create component artifacts from your Java application and the Stream Manager SDK JAR file:
   - Build your application as a JAR file that includes the Stream Manager SDK JAR, and run this JAR file in your component recipe.
   - Define the Stream Manager SDK JAR as a component artifact. Add that artifact to the classpath when you run your application in your component recipe.

Your component recipe might look like the following example. This component runs a modified version of the `StreamManagerS3.java` example, where `StreamManagerS3.jar` includes the Stream Manager SDK JAR.

**JSON**

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.StreamManagerS3Java",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Uses stream manager to upload a file to an S3 bucket.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.StreamManager": {
      "VersionRequirement": "^2.0.0"
    }
  }
},
"Manifests": [
{
  "Lifecycle": {
    "Run": "java -jar {artifacts:path}/StreamManagerS3.jar"
  },
  "Artifacts": [
    {
      "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.StreamManagerS3Java/1.0.0/StreamManagerS3.jar"
    }
  ]
},
]
```

**YAML**

```yaml
---
RecipeFormatVersion: '2020-01-25'
```
Define component recipes that use stream manager

ComponentName: com.example.StreamManagerS3Java  
ComponentVersion: 1.0.0  
ComponentDescription: Uses stream manager to upload a file to an S3 bucket.  
ComponentPublisher: Amazon  
ComponentDependencies:  
  aws.greengrass.StreamManager:  
    Version Requirement: "^2.0.0"

Manifests:  
- Lifecycle:  
  Run: java -jar {artifacts:path}/StreamManagerS3.jar  
Artifact:  
- URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.StreamManagerS3Java/1.0.0/StreamManagerS3.jar

For more information about how to develop and test components, see Create local AWS IoT Greengrass components (p. 324).

Use the Stream Manager SDK for Python

The Stream Manager SDK for Python is available as source code that you can include in your component. Create a ZIP file of the Stream Manager SDK, define the ZIP file as a component artifact, and install the SDK's requirements in the component lifecycle.

To use the Stream Manager SDK for Python

1. Clone or download the aws-greengrass-stream-manager-sdk-python repository.

```
  git clone git@github.com:aws-greengrass/aws-greengrass-stream-manager-sdk-python.git
```

2. Create a ZIP file that contains the stream_manager folder, which contains the source code of the Stream Manager SDK for Python. You can provide this ZIP file as a component artifact that the AWS IoT Greengrass Core software unzips when it installs your component. Do the following:

   a. Open the folder that contains the repository that you cloned or downloaded in the previous step.

```
  cd aws-greengrass-stream-manager-sdk-python
```

   b. Zip the stream_manager folder into a ZIP file named stream_manager_sdk.zip.

```
  zip -r stream_manager_sdk.zip stream_manager
```

   c. Verify that the stream_manager_sdk.zip file contains the stream_manager folder and its contents. Run the following command to list the contents of the ZIP file.

```
  unzip -l stream_manager_sdk.zip
```

The output should look similar to the following.

```
  Archive: aws-greengrass-stream-manager-sdk-python/stream_manager.zip  
  Length  Date     Time   Name
  -------- -------- ------- ------------------------
      913  02-24-2021 20:45  stream_manager/__init__.py
     9719  02-24-2021 20:45  stream_manager/utilinternal.py
     1412  02-24-2021 20:45  stream_manager/exceptions.py
     1004  02-24-2021 20:45  stream_manager/util.py
```
Define component recipes that use stream manager

3. Copy the Stream Manager SDK artifacts to your component's artifacts folder. In addition to the Stream Manager SDK ZIP file, your component uses the SDK's requirements.txt file to install the dependencies of the Stream Manager SDK. Replace ~/greengrass-components with the path to the folder that you use for local development.

```
cp {stream_manager_sdk.zip,requirements.txt} ~/greengrass-components/artifacts/com.example.StreamManagerS3Python/1.0.0/
```

4. Create your component recipe. In the recipe, do the following:
   a. Define stream_manager_sdk.zip and requirements.txt as artifacts.
   b. Define your Python application as an artifact.
   c. In the install lifecycle, install the Stream Manager SDK requirements from requirements.txt.
   d. In the run lifecycle, append the Stream Manager SDK to PYTHONPATH, and run your Python application.

Your component recipe might look like the following example. This component runs the stream_manager_s3.py example.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.StreamManagerS3Python",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Uses stream manager to upload a file to an S3 bucket.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.StreamManager": {
      "VersionRequirement": "^2.0.0"
    }
  }
},
"Manifests": [
  {
    "Platform": {
      "os": "linux"
    },
    "Lifecycle": {
      "Install": "pip3 install --user -r {artifacts:path}/requirements.txt",
      "Run": "export PYTHONPATH=$PYTHONPATH:{artifacts:decompressedPath}/stream_manager_sdk; python3 {artifacts:path}/stream_manager_s3.py"
    },
    "Artifacts": [
      {
        "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.StreamManagerS3Python/1.0.0/stream_manager_sdk.zip",
        "Unarchive": "ZIP"
      },
      {
        "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.StreamManagerS3Python/1.0.0/stream_manager_s3.py"
      },
      {
        "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.StreamManagerS3Python/1.0.0/requirements.txt"
      }
    ]
  }
}
Define component recipes that use stream manager

YAML

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.StreamManagerS3Python
ComponentVersion: 1.0.0
ComponentDescription: Uses stream manager to upload a file to an S3 bucket.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.StreamManager:
    VersionRequirement: "^2.0.0"
Manifests:
- Platform:
  os: linux
Lifecycle:
  Install: pip3 install --user -r {artifacts:path}/requirements.txt
  Run: |
    export PYTHONPATH=$PYTHONPATH:{artifacts:decompressedPath}/
    stream_manager_sdk
    python3 {artifacts:path}/stream_manager_s3.py
Artifacts:
- URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.StreamManagerS3Python/1.0.0/stream_manager_sdk.zip
  Unarchive: ZIP
- URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.StreamManagerS3Python/1.0.0/stream_manager_s3.py
- URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.StreamManagerS3Python/1.0.0/requirements.txt
```

For more information about how to develop and test components, see Create local AWS IoT Greengrass components (p. 324).

Use the Stream Manager SDK for JavaScript

The Stream Manager SDK for JavaScript is available as source code that you can include in your component. Create a ZIP file of the Stream Manager SDK, define the ZIP file as a component artifact, and install the SDK in the component lifecycle.

To use the Stream Manager SDK for JavaScript

1. Clone or download the aws-greengrass-stream-manager-sdk-js repository.

   ```bash
git clone git@github.com:aws-greengrass/aws-greengrass-stream-manager-sdk-js.git
```

2. Create a ZIP file that contains the aws-greengrass-stream-manager-sdk folder, which contains the source code of the Stream Manager SDK for JavaScript. You can provide this ZIP file as a component artifact that the AWS IoT Greengrass Core software unzips when it installs your component. Do the following:

   a. Open the folder that contains the repository that you cloned or downloaded in the previous step.
Define component recipes that use stream manager

```
cd aws-greengrass-stream-manager-sdk-js
```


```
zip -r stream-manager-sdk.zip aws-greengrass-stream-manager-sdk
```

c. Verify that the stream-manager-sdk.zip file contains the aws-greengrass-stream-manager-sdk folder and its contents. Run the following command to list the contents of the ZIP file.

```
unzip -l stream-manager-sdk.zip
```

The output should look similar to the following.

<table>
<thead>
<tr>
<th>Archive: stream-manager-sdk.zip</th>
<th>Length</th>
<th>Date</th>
<th>Time</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-------</td>
<td>---------</td>
<td>---------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>387855</td>
<td>9 files</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. Copy the Stream Manager SDK artifact to your component's artifacts folder. Replace `~/greengrass-components` with the path to the folder that you use for local development.

```
cp stream-manager-sdk.zip ~/greengrass-components/artifacts/com.example.StreamManagerS3JS/1.0.0/
```

4. Create your component recipe. In the recipe, do the following:

   a. Define stream-manager-sdk.zip as an artifact.
   
   b. Define your JavaScript application as an artifact.
   
   c. In the install lifecycle, install the Stream Manager SDK from the stream-manager-sdk.zip artifact. This `npm install` command creates a `node_modules` folder that contains the Stream Manager SDK and its dependencies.
   
   d. In the run lifecycle, append the `node_modules` folder to `NODE_PATH`, and run your JavaScript application.

Your component recipe might look like the following example. This component runs the `StreamManagerS3` example.

```
JSON

{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.StreamManagerS3JS",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Uses stream manager to upload a file to an S3 bucket."
}
```
Define component recipes that use stream manager

```
"ComponentPublisher": "Amazon",
"ComponentDependencies": {
  "aws.greengrass.StreamManager": {
    "VersionRequirement": "^2.0.0"
  }
},
"Manifests": [
  {
    "Platform": {
      "os": "linux"
    },
    "Lifecycle": {
      "Install": "npm install {artifacts:decompressedPath}/stream-manager-sdk/aws-greengrass-stream-manager-sdk",
      "Run": "export NODE_PATH=$NODE_PATH:{work:path}/node_modules; node {artifacts:path}/index.js"
    },
    "Artifacts": [
      {
        "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.StreamManagerS3JS/1.0.0/stream-manager-sdk.zip",
        "Unarchive": "ZIP"
      },
      {
        "URI": "s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.StreamManagerS3JS/1.0.0/index.js"
      }
    ]
  }
]
```

YAML

```yaml
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.StreamManagerS3JS
ComponentVersion: 1.0.0
ComponentDescription: Uses stream manager to upload a file to an S3 bucket.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.StreamManager:
    VersionRequirement: "^2.0.0"
Manifests:
- Platform:
  os: linux
Lifecycle:
  Install: npm install {artifacts:decompressedPath}/stream-manager-sdk/aws-greengrass-stream-manager-sdk
Run: |
  export NODE_PATH=$NODE_PATH:{work:path}/node_modules
  node {artifacts:path}/index.js
Artifacts:
- URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.StreamManagerS3JS/1.0.0/stream-manager-sdk.zip
  Unarchive: ZIP
- URI: s3://DOC-EXAMPLE-BUCKET/artifacts/com.example.StreamManagerS3JS/1.0.0/index.js
```

For more information about how to develop and test components, see Create local AWS IoT Greengrass components (p. 324).
Connect to stream manager in application code

To connect to stream manager in your application, create an instance of `StreamManagerClient` from the Stream Manager SDK. This client connects to the stream manager component on its default port 8088, or the port that you specify. For more information about how to use `StreamManagerClient` after you create an instance, see Use `StreamManagerClient` to work with streams (p. 529).

**Example Example: Connect to stream manager with default port**

Java

```java
import com.amazonaws.greengrass.streammanager.client.StreamManagerClient;

public class MyStreamManagerComponent {
    void connectToStreamManagerWithDefaultPort() {
        StreamManagerClient client = StreamManagerClientFactory.standard().build();
        // Use the client.
    }
}
```

Python

```python
from stream_manager import (StreamManagerClient)

def connect_to_stream_manager_with_default_port():
    client = StreamManagerClient()
    # Use the client.
```

JavaScript

```javascript
const {
    StreamManagerClient
} = require('aws-greengrass-stream-manager-sdk');

function connectToStreamManagerWithDefaultPort() {
    const client = new StreamManagerClient();
    // Use the client.
}
```

**Example Example: Connect to stream manager with non-default port**

If you configure stream manager with a port other than the default, you must use interprocess communication (p. 396) to retrieve the port from the component configuration.

**Note**

The `port` configuration parameter contains the value that you specify in `STREAM_MANAGER_SERVER_PORT` when you deploy stream manager.

Java

```java
void connectToStreamManagerWithCustomPort() {
```
Use StreamManagerClient to work with streams

User-defined Greengrass components that run on the Greengrass core device can use the StreamManagerClient object in the Stream Manager SDK to create streams in stream manager (p. 519) and then interact with the streams. When a component creates a stream, it defines the AWS Cloud destinations, prioritization, and other export and data retention policies for the stream.
To send data to stream manager, components append the data to the stream. If an export destination is defined for the stream, stream manager exports the stream automatically.

**Note**
Typically, clients of stream manager are user-defined Greengrass components. If your business case requires it, you can also allow non-component processes running on the Greengrass core (for example, a Docker container) to interact with stream manager. For more information, see the section called “Client authentication” (p. 521).

The snippets in this topic show you how clients call `StreamManagerClient` methods to work with streams. For implementation details about the methods and their arguments, use the links to the SDK reference listed after each snippet.

If you use stream manager in a Lambda function, your Lambda function should instantiate `StreamManagerClient` outside of the function handler. If instantiated in the handler, the function creates a client and connection to stream manager every time that it's invoked.

**Note**
If you do instantiate `StreamManagerClient` in the handler, you must explicitly call the `close()` method when the client completes its work. Otherwise, the client keeps the connection open and another thread running until the script exits.

`StreamManagerClient` supports the following operations:

- the section called “Create message stream” (p. 530)
- the section called “Append message” (p. 533)
- the section called “Read messages” (p. 537)
- the section called “List streams” (p. 539)
- the section called “Describe message stream” (p. 540)
- the section called “Update message stream” (p. 541)
- the section called “Delete message stream” (p. 544)

**Create message stream**

To create a stream, a user-defined Greengrass component calls the create method and passes in a `MessageStreamDefinition` object. This object specifies the unique name for the stream and defines how stream manager should handle new data when the maximum stream size is reached. You can use `MessageStreamDefinition` and its data types (such as `ExportDefinition`, `StrategyOnFull`, and `Persistence`) to define other stream properties. These include:

- The target AWS IoT Analytics, Kinesis Data Streams, AWS IoT SiteWise, and Amazon S3 destinations for automatic exports. For more information, see the section called “Export configurations for supported cloud destinations” (p. 545).
- Export priority. Stream manager exports higher priority streams before lower priority streams.
- Maximum batch size and batch interval for AWS IoT Analytics, Kinesis Data Streams, and AWS IoT SiteWise destinations. Stream manager exports messages when either condition is met.
- Time-to-live (TTL). The amount of time to guarantee that the stream data is available for processing. You should make sure that the data can be consumed within this time period. This is not a deletion policy. The data might not be deleted immediately after TTL period.
- Stream persistence. Choose to save streams to the file system to persist data across core restarts or save streams in memory.
- Starting sequence number. Specify the sequence number of the message to use as the starting message in the export.
Create message stream

For more information about `MessageStreamDefinition`, see the SDK reference for your target language:

- `MessageStreamDefinition` in the Java SDK
- `MessageStreamDefinition` in the Node.js SDK
- `MessageStreamDefinition` in the Python SDK

**Note**

StreamManagerClient also provides a target destination you can use to export streams to an HTTP server. This target is intended for testing purposes only. It is not stable or supported for use in production environments.

After a stream is created, your Greengrass components can append messages (p. 533) to the stream to send data for export and read messages (p. 537) from the stream for local processing. The number of streams that you create depends on your hardware capabilities and business case. One strategy is to create a stream for each target channel in AWS IoT Analytics or Kinesis data stream, though you can define multiple targets for a stream. A stream has a durable lifespan.

**Requirements**

This operation has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

**Examples**

The following snippet creates a stream named `StreamName`. It defines stream properties in the `MessageStreamDefinition` and subordinate data types.

**Python**

```python
client = StreamManagerClient()

try:
    client.create_message_stream(MessageStreamDefinition(
        name="StreamName",  # Required.
        max_size=268435456,  # Default is 256 MB.
        stream_segment_size=16777216,  # Default is 16 MB.
        time_to_live_millis=None,  # By default, no TTL is enabled.
        strategy_on_full=StrategyOnFull.OverwriteOldestData,  # Required.
        persistence=Persistence.File,  # Default is File.
        flush_on_write=False,  # Default is false.
        export_definition=ExportDefinition(  # Optional. Choose where/how the stream
            kinesis=None,
            iot_analytics=None,
            iot_sitewise=None,
            s3_task_executor=None
        )
    ))
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: [create_message_stream](create_message_stream) | [MessageStreamDefinition](MessageStreamDefinition)
Java

```java
try (final StreamManagerClient client = StreamManagerClientFactory.standard().build()) {
    client.createMessageStream(
        new MessageStreamDefinition()
            .withName("StreamName") // Required.
            .withMaxSize(268435456L) // Default is 256 MB.
            .withStreamSegmentSize(16777216L) // Default is 16 MB.
            .withTimeToLiveMillis(null) // By default, no TTL is enabled.
            .withStrategyOnFull(StrategyOnFull.OverwriteOldestData) // Required.
            .withPersistence(Persistence.File) // Default is File.
            .withFlushOnWrite(false) // Default is false.
            .withExportDefinition( // Optional. Choose where/how the stream is exported to the AWS Cloud.
                new ExportDefinition()
                    .withKinesis(null)
                    .withIotAnalytics(null)
                    .withIotSiteWise(null)
                    .withS3(null)
            )
    );
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: `createMessageStream` | `MessageStreamDefinition`

Node.js

```javascript
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        await client.createMessageStream(
            new MessageStreamDefinition()
                .withName("StreamName") // Required.
                .withMaxSize(268435456) // Default is 256 MB.
                .withStreamSegmentSize(16777216) // Default is 16 MB.
                .withTimeToLiveMillis(null) // By default, no TTL is enabled.
                .withStrategyOnFull(StrategyOnFull.OverwriteOldestData) // Required.
                .withPersistence(Persistence.File) // Default is File.
                .withFlushOnWrite(false) // Default is false.
                .withExportDefinition( // Optional. Choose where/how the stream is exported to the AWS Cloud.
                    new ExportDefinition()
                        .withKinesis(null)
                        .withIotAnalytics(null)
                        .withIotSiteWise(null)
                        .withS3(null)
                )
        );
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK reference: `createMessageStream` | `MessageStreamDefinition`
For more information about configuring export destinations, see the section called “Export configurations for supported cloud destinations” (p. 545).

## Append message

To send data to stream manager for export, your Greengrass components append the data to the target stream. The export destination determines the data type to pass to this method.

### Requirements

This operation has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

### Examples

#### AWS IoT Analytics or Kinesis Data Streams export destinations

The following snippet appends a message to the stream named `StreamName`. For AWS IoT Analytics or Kinesis Data Streams destinations, your Greengrass components append a blob of data.

This snippet has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

**Python**

```python
client = StreamManagerClient()
try:
    sequence_number = client.append_message(stream_name="StreamName", data=b'Arbitrary bytes data')
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: [append_message](#)

**Java**

```java
try (final StreamManagerClient client = StreamManagerClientFactory.standard().build()) {
    long sequenceNumber = client.appendMessage("StreamName", "Arbitrary byte array".getBytes());
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: [appendMessage](#)

**Node.js**

```javascript
const client = new StreamManagerClient();
```
client.onConnected(async () => {
  try {
    const sequenceNumber = await client.appendMessage("StreamName",
             Buffer.from("Arbitrary byte array"));
    } catch (e) {
      // Properly handle errors.
    }
  });
client.onError((err) => {
  // Properly handle connection errors.
  // This is called only when the connection to the StreamManager server fails.
});

Node.js SDK reference: appendMessage

AWS IoT SiteWise export destinations

The following snippet appends a message to the stream named StreamName. For AWS IoT SiteWise destinations, your Greengrass components append a serialized PutAssetPropertyValueEntry object. For more information, see the section called “Exporting to AWS IoT SiteWise” (p. 549).

   Note
   When you send data to AWS IoT SiteWise, your data must meet the requirements of the BatchPutAssetPropertyValue action. For more information, see BatchPutAssetPropertyValue in the AWS IoT SiteWise API Reference.

This snippet has the following requirements:

   • Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Python

client = StreamManagerClient()

try:
    # SiteWise requires unique timestamps in all messages and also needs timestamps not earlier
    # than 10 minutes in the past. Add some randomness to time and offset.

    # Note: To create a new asset property data, you should use the classes defined in the
    # greengrasssdk.stream_manager module.

    time_in_nanos = TimeInNanos(
        time_in_seconds=calendar.timegm(time.gmtime()) - random.randint(0, 60),
        offset_in_nanos=random.randint(0, 10000)
    )
    variant = Variant(double_value=random.random())
    asset = [AssetPropertyValue(value=variant, quality=Quality.GOOD, timestamp=time_in_nanos)]
    putAssetPropertyValueEntry = PutAssetPropertyValueEntry(entry_id=str(uuid.uuid4()),
                                         property_alias="PropertyAlias", property_values=asset)
    sequence_number = client.append_message(stream_name="StreamName",
                               Util.validate_and_serialize_to_json_bytes(putAssetPropertyValueEntry))
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
Python SDK reference: `append_message` | `PutAssetPropertyValueEntry`

Java

```java
try (final StreamManagerClient client = GreengrassClientBuilder.streamManagerClient().build()) {
    Random rand = new Random();
    // Note: To create a new asset property data, you should use the classes defined in the
    // com.amazonaws.greengrass.streammanager.model.sitewise package.
    List<AssetPropertyValue> entries = new ArrayList<>();

    // IoTSiteWise requires unique timestamps in all messages and also needs timestamps not earlier
    // than 10 minutes in the past. Add some randomness to time and offset.
    final int maxTimeRandomness = 60;
    final int maxOffsetRandomness = 10000;
    double randomValue = rand.nextDouble();
    TimeInNanos timestamp = new TimeInNanos()
        .withTimeInSeconds(Instant.now().getEpochSecond() -
        rand.nextInt(maxTimeRandomness))
        .withOffsetInNanos((long) (rand.nextInt(maxOffsetRandomness)));
    AssetPropertyValue entry = new AssetPropertyValue()
        .withValue(new Variant().withDoubleValue(randomValue))
        .withQuality(Quality.GOOD)
        .withTimestamp(timestamp);
    entries.add(entry);

    PutAssetPropertyValueEntry putAssetPropertyValueEntry = new PutAssetPropertyValueEntry()
        .withEntryId(UUID.randomUUID().toString())
        .withPropertyAlias("PropertyAlias")
        .withPropertyValues(entries);

    long sequenceNumber = client.appendMessage("StreamName", ValidateAndSerialize.validateAndSerializeToJsonBytes(putAssetPropertyValueEntry));
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: `appendMessage` | `PutAssetPropertyValueEntry`

Node.js

```javascript
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const maxTimeRandomness = 60;
        const maxOffsetRandomness = 10000;
        const randomValue = Math.random();
        // Note: To create a new asset property data, you should use the classes defined in the
        // aws-greengrass-core-sdk StreamManager module.
        const timestamp = new TimeInNanos()
            .withTimeInSeconds(Math.round(Date.now() / 1000) - Math.floor(Math.random()
            - maxTimeRandomness))
            .withOffsetInNanos(Math.floor(Math.random() * maxOffsetRandomness));
        const entry = new AssetPropertyValue()
            .withValue(new Variant().withDoubleValue(randomValue))
            .withQuality(Quality.GOOD)
            .withTimestamp(timestamp);

        const putAssetPropertyValueEntry = new PutAssetPropertyValueEntry()
            .withEntryId(`${ENTRY_ID_PREFIX}${i}`)
            .withPropertyAlias("PropertyAlias")
            .withPropertyValues([entry]);
```
const sequenceNumber = await client.appendMessage("StreamName",
util.validateAndSerializeToJsonBytes(putAssetPropertyValueEntry));
} catch (e) {
    // Properly handle errors.
}

client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});

Node.js SDK reference: appendMessage | PutAssetPropertyValueEntry

Amazon S3 export destinations

The following snippet appends an export task to the stream named StreamName. For Amazon S3 destinations, your Greengrass components append a serialized S3ExportTaskDefinition object that contains information about the source input file and target Amazon S3 object. If the specified object doesn’t exist, Stream Manager creates it for you. For more information, see the section called “Exporting to Amazon S3” (p. 551).

This snippet has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Python

```python
client = StreamManagerClient()

try:
    # Append an Amazon S3 Task definition and print the sequence number.
    s3_export_task_definition = S3ExportTaskDefinition(input_url="URLToFile",
                                                      bucket="BucketName", key="KeyName")
    sequence_number = client.append_message(stream_name="StreamName",
                                             Util.validate_and_serialize_to_json_bytes(s3_export_task_definition))
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: append_message | S3ExportTaskDefinition

Java

```java
try (final StreamManagerClient client =
    GreengrassClientBuilder.streamManagerClient().build()) {
    // Append an Amazon S3 export task definition and print the sequence number.
    S3ExportTaskDefinition s3ExportTaskDefinition = new S3ExportTaskDefinition()
        .withBucket("BucketName")
        .withKey("KeyName")
        .withInputUrl("URLToFile");
    long sequenceNumber = client.appendMessage("StreamName",
                                             ValidateAndSerialize.validateAndSerializeToJsonBytes(s3ExportTaskDefinition));
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: appendMessage | S3ExportTaskDefinition
Node.js

```javascript
const client = new StreamManagerClient();
client.onConnected(async () => {
  try {
    // Append an Amazon S3 export task definition and print the sequence number.
    const taskDefinition = new S3ExportTaskDefinition()
      .withBucket("BucketName")
      .withKey("KeyName")
      .withInputUrl("URLToFile");
    const sequenceNumber = await client.appendMessage("StreamName",
      util.validateAndSerializeToJsonBytes(taskDefinition));
  } catch (e) {
    // Properly handle errors.
  }
};
client.onError((err) => {
  // Properly handle connection errors.
  // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK reference: [appendMessage](#) | [S3ExportTaskDefinition](#)

**Read messages**

Read messages from a stream.

**Requirements**

This operation has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

**Examples**

The following snippet reads messages from the stream named `StreamName`. The read method takes an optional `ReadMessagesOptions` object that specifies the sequence number to start reading from, the minimum and maximum numbers to read, and a timeout for reading messages.

### Python

```python
client = StreamManagerClient()
try:
    message_list = client.read_messages(
        stream_name="StreamName",
        # By default, if no options are specified, it tries to read one message from
        the beginning of the stream.
        options=ReadMessagesOptions(
            desired_start_sequence_number=100,
            # Try to read from sequence number 100 or greater. By default, this is 0.
            min_message_count=10,
            # Try to read 10 messages. If 10 messages are not available, then
            max_message_count=100,    # Accept up to 100 messages. By default this is
            read_timeout_millis=5000
            # Try to wait at most 5 seconds for the min_msgssage_count to be fulfilled.
        )
```

By default, this is 0, which immediately returns the messages or an exception.
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.

Python SDK reference: read_messages | ReadMessagesOptions

Java

try (final StreamManagerClient client = StreamManagerClientFactory.standard().build())
{
    List<Message> messages = client.readMessages("StreamName",
        // By default, if no options are specified, it tries to read one message
        from the beginning of the stream.
            new ReadMessagesOptions()
            // Try to read from sequence number 100 or greater. By default this
            is 0.
                .withDesiredStartSequenceNumber(100L)
            // Try to read 10 messages. If 10 messages are not available, then
NotEnoughMessagesException is raised. By default, this is 1.
                .withMinMessageCount(10L)
            // Accept up to 100 messages. By default this is 1.
                .withMaxMessageCount(100L)
            // Try to wait at most 5 seconds for the min_message_count to
be fulfilled. By default, this is 0, which immediately returns the messages or an
exception.
                .withReadTimeoutMillis(Duration.ofSeconds(5L).toMillis())
        );
    } catch (StreamManagerException e) {
        // Properly handle exception.
    }

Java SDK reference: readMessages | ReadMessagesOptions

Node.js

const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const messages = await client.readMessages("StreamName",
            // By default, if no options are specified, it tries to read one message
            from the beginning of the stream.
            new ReadMessagesOptions()
            // Try to read from sequence number 100 or greater. By default this
            is 0.
                .withDesiredStartSequenceNumber(100)
            // Try to read 10 messages. If 10 messages are not available, then
NotEnoughMessagesException is thrown. By default, this is 1.
                .withMinMessageCount(10)
            // Accept up to 100 messages. By default this is 1.
                .withMaxMessageCount(100)
            // Try to wait at most 5 seconds for the minMessageCount to be
fulfilled. By default, this is 0, which immediately returns the messages or an
exception.
                .withReadTimeoutMillis(5 * 1000)
        );
    } catch (e) {
        // Properly handle errors.
    }
});
List streams

Get the list of streams in stream manager.

Requirements

This operation has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Examples

The following snippet gets a list of the streams (by name) in stream manager.

Python

```python
client = StreamManagerClient()
try:
    stream_names = client.list_streams()
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: `list_streams`

Java

```java
try (final StreamManagerClient client = StreamManagerClientFactory.standard().build())
{
    List<String> streamNames = client.listStreams();
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: `listStreams`

Node.js

```javascript
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const streams = await client.listStreams();
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
```

Node.js SDK reference: `readMessages` | `ReadMessagesOptions`
Describe message stream

Get metadata about a stream, including the stream definition, size, and export status.

Requirements

This operation has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Examples

The following snippet gets metadata about the stream named StreamName, including the stream's definition, size, and exporter statuses.

Python

```python
client = StreamManagerClient()

try:
    stream_description = client.describe_message_stream(stream_name="StreamName")
    if stream_description.export_statuses[0].error_message:
        # The last export of export destination 0 failed with some error
        # Here is the last sequence number that was successfully exported
        stream_description.export_statuses[0].last_exported_sequence_number

    if (stream_description.storage_status.newest_sequence_number >
        stream_description.export_statuses[0].last_exported_sequence_number):
        pass
        # The end of the stream is ahead of the last exported sequence number
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: describe_message_stream

Java

```java
try (final StreamManagerClient client = StreamManagerClientFactory.standard().build()) {
    MessageStreamInfo description = client.describeMessageStream("StreamName");
    String lastErrorMessage = description.getExportStatuses().get(0).getErrorMessages();
    if (lastErrorMessage != null && !lastErrorMessage.equals("") ) {
        // The last export of export destination 0 failed with some error.
        // Here is the last sequence number that was successfully exported.
        description.getExportStatuses().get(0).getLastExportedSequenceNumber();
    }

    if (description.getStorageStatus().getNewestSequenceNumber() >
```
description.getExportStatuses().get(0).getLastExportedSequenceNumber()) {
    // The end of the stream is ahead of the last exported sequence number.
}
} catch (StreamManagerException e) {
    // Properly handle exception.
}

Java SDK reference: `describeMessageStream`

Node.js

```javascript
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const description = await client.describeMessageStream("StreamName");
        const lastErrorMessage = description.exportStatuses[0].errorMessage;
        if (lastErrorMessage) {
            // The last export of export destination 0 failed with some error.
            // Here is the last sequence number that was successfully exported.
            description.exportStatuses[0].lastExportedSequenceNumber;
        }
        if (description.storageStatus.newestSequenceNumber >
            description.exportStatuses[0].lastExportedSequenceNumber) {
            // The end of the stream is ahead of the last exported sequence number.
        }
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK reference: `describeMessageStream`

### Update message stream

Update properties of an existing stream. You might want to update a stream if your requirements change after the stream was created. For example:

- Add a new export configuration (p. 545) for an AWS Cloud destination.
- Increase the maximum size of a stream to change how data is exported or retained. For example, the stream size in combination with your strategy on full settings might result in data being deleted or rejected before stream manager can process it.
- Pause and resume exports; for example, if export tasks are long running and you want to ration your upload data.

Your Greengrass components follow this high-level process to update a stream:

1. Get the description of the stream. (p. 540)
2. Update the target properties on the corresponding `MessageStreamDefinition` and subordinate objects.
3. Pass in the updated `MessageStreamDefinition`. Make sure to include the complete object definitions for the updated stream. Undefined properties revert to the default values.

You can specify the sequence number of the message to use as the starting message in the export.
Requirements

This operation has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Examples

The following snippet updates the stream named StreamName. It updates multiple properties of a stream that exports to Kinesis Data Streams.

Python

```python
client = StreamManagerClient()
try:
    message_stream_info = client.describe_message_stream(STREAM_NAME)
    message_stream_info.definition.max_size=536870912
    message_stream_info.definition.stream_segment_size=33554432
    message_stream_info.definition.time_to_live_millis=3600000
    message_stream_info.definition.strategy_on_full=StrategyOnFull.RejectNewData
    message_stream_info.definition.persistence=Persistence.Memory
    message_stream_info.definition.flush_on_write=False
    message_stream_info.definition.export_definition.kinesis=[KinesisConfig(
        identifier=str(uuid.uuid4()), kinesis_stream_name=str(uuid.uuid4()))]
    client.update_message_stream(message_stream_info.definition)
except StreamManagerException:
    pass
# Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
# Properly handle errors.
```

Python SDK reference: updateMessageStream | MessageStreamDefinition

Java

```java
try (final StreamManagerClient client = GreengrassClientBuilder.streamManagerClient().build()) {
    MessageStreamInfo messageStreamInfo = client.describeMessageStream(STREAM_NAME);
    // Update the message stream with new values.
    client.updateMessageStream(
        messageStreamInfo.getDefinition() // Required. Updating Strategy on full to reject new data.
            .withStrategyOnFull(StrategyOnFull.RejectNewData) // Required. Updating Strategy on full to reject new data.
            .withMaxSize(536870912L) // Update Max Size to 512 MB.
            .withStreamSegmentSize(33554432L) // Update Segment Size to 32 MB.
            .withFlushOnWrite(true) // Update flush on write to true.
            .withPersistence(Persistence.Memory) // Update the persistence to Memory.
            .withTimeToLiveMillis(3600000L) // Update TTL to 1 hour.
            .withExportDefinition( // Optional. Choose where/how the stream is exported to the AWS Cloud.
                messageStreamInfo.getDefinition().getExportDefinition().
                    withKinesis(new ArrayList<KinesisConfig>() {
                        add(new KinesisConfig())
                    }));
```
Constraints for updating streams

The following constraints apply when updating streams. Unless noted in the following list, updates take effect immediately.

- You can't update a stream's persistence. To change this behavior, delete the stream (p. 544) and create a stream (p. 530) that defines the new persistence policy.
- You can update the maximum size of a stream only under the following conditions:
• The maximum size must be greater or equal to the current size of the stream. To find this information, describe the stream (p. 540) and then check the storage status of the returned MessageStreamInfo object.
• The maximum size must be greater than or equal to the stream's segment size.
• You can update the stream segment size to a value less than the maximum size of the stream. The updated setting applies to new segments.
• Updates to the time to live (TTL) property apply to new append operations. If you decrease this value, stream manager might also delete existing segments that exceed the TTL.
• Updates to the strategy on full property apply to new append operations. If you set the strategy to overwrite the oldest data, stream manager might also overwrite existing segments based on the new setting.
• Updates to the flush on write property apply to new messages.
• Updates to export configurations apply to new exports. The update request must include all export configurations that you want to support. Otherwise, stream manager deletes them.
  • When you update an export configuration, specify the identifier of the target export configuration.
  • To add an export configuration, specify a unique identifier for the new export configuration.
  • To delete an export configuration, omit the export configuration.
• To update (p. 541) the starting sequence number of an export configuration in a stream, you must specify a value that's less than the latest sequence number. To find this information, describe the stream (p. 540) and then check the storage status of the returned MessageStreamInfo object.

Delete message stream

Deletes a stream. When you delete a stream, all of the stored data for the stream is deleted from the disk.

Requirements

This operation has the following requirements:

• Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Examples

The following snippet deletes the stream named StreamName.

Python

```python
client = StreamManagerClient()
try:
    client.delete_message_stream(stream_name="StreamName")
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: deleteMessageStream
Java

```java
try (final StreamManagerClient client = StreamManagerClientFactory.standard().build()) {
    client.deleteMessageStream("StreamName");
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: delete_message_stream

Node.js

```javascript
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        await client.deleteMessageStream("StreamName");
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK reference: deleteMessageStream

See also

- Manage data streams on the AWS IoT Greengrass Core (p. 519)
- Configure AWS IoT Greengrass stream manager (p. 555)
- Export configurations for supported AWS Cloud destinations (p. 545)
- StreamManagerClient in the Stream Manager SDK reference:
  - Python
  - Java
  - Node.js

Export configurations for supported AWS Cloud destinations

User-defined Greengrass components use StreamManagerClient in the Stream Manager SDK to interact with stream manager. When a component creates a stream (p. 530) or updates a stream (p. 530), it passes a MessageStreamDefinition object that represents stream properties, including the export definition. The ExportDefinition object contains the export configurations defined for the stream. Stream manager uses these export configurations to determine where and how to export the stream.
You can define zero or more export configurations on a stream, including multiple export configurations for a single destination type. For example, you can export a stream to two AWS IoT Analytics channels and one Kinesis data stream.

For failed export attempts, stream manager continually retries exporting data to the AWS Cloud at intervals of up to five minutes. The number of retry attempts doesn't have a maximum limit.

**Note**

StreamManagerClient also provides a target destination you can use to export streams to an HTTP server. This target is intended for testing purposes only. It is not stable or supported for use in production environments.

**Supported AWS Cloud destinations**

- AWS IoT Analytics channels (p. 546)
- Amazon Kinesis data streams (p. 547)
- AWS IoT SiteWise asset properties (p. 548)
- Amazon S3 objects (p. 550)

You are responsible for maintaining these AWS Cloud resources.

**AWS IoT Analytics channels**

Stream manager supports automatic exports to AWS IoT Analytics. AWS IoT Analytics lets you perform advanced analysis on your data to help make business decisions and improve machine learning models. For more information, see What is AWS IoT Analytics? in the AWS IoT Analytics User Guide.

In the Stream Manager SDK, your Greengrass components use the IoTAnalyticsConfig to define the export configuration for this destination type. For more information, see the SDK reference for your target language:

- **IoTAnalyticsConfig** in the Python SDK
- **IoTAnalyticsConfig** in the Java SDK
- **IoTAnalyticsConfig** in the Node.js SDK

**Requirements**

This export destination has the following requirements:
• Target channels in AWS IoT Analytics must be in the same AWS account and AWS Region as the Greengrass core device.

• The Authorize core devices to interact with AWS services (p. 627) must allow the `iotanalytics:BatchPutMessage` permission to target channels. For example:

```json
{
"Version": "2012-10-17",
"Statement": [
{
"Effect": "Allow",
"Action": [
"iotanalytics:BatchPutMessage"
],
"Resource": [
"arn:aws:iotanalytics:region:account-id:channel/channel_1_name",
"arn:aws:iotanalytics:region:account-id:channel/channel_2_name"
]
}
]
}
```

You can grant granular or conditional access to resources, for example, by using a wildcard * naming scheme. For more information, see Adding and removing IAM policies in the IAM User Guide.

### Exporting to AWS IoT Analytics

To create a stream that exports to AWS IoT Analytics, your Greengrass components create a stream (p. 530) with an export definition that includes one or more `IoTAnalyticsConfig` objects. This object defines export settings, such as the target channel, batch size, batch interval, and priority.

When your Greengrass components receive data from devices, they append messages (p. 533) that contain a blob of data to the target stream.

Then, stream manager exports the data based on the batch settings and priority defined in the stream's export configurations.

### Amazon Kinesis data streams

Stream manager supports automatic exports to Amazon Kinesis Data Streams. Kinesis Data Streams is commonly used to aggregate high-volume data and load it into a data warehouse or MapReduce cluster. For more information, see What is Amazon Kinesis Data Streams? in the Amazon Kinesis Developer Guide.

In the Stream Manager SDK, your Greengrass components use the `KinesisConfig` to define the export configuration for this destination type. For more information, see the SDK reference for your target language:

- `KinesisConfig` in the Python SDK
- `KinesisConfig` in the Java SDK
- `KinesisConfig` in the Node.js SDK

### Requirements

This export destination has the following requirements:

• Target streams in Kinesis Data Streams must be in the same AWS account and AWS Region as the Greengrass core device.
The Authorize core devices to interact with AWS services (p. 627) must allow the kinesis:PutRecords permission to target data streams. For example:

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": ["kinesis:PutRecords"],
         "Resource": [
            "arn:aws:kinesis:region:account-id:stream/stream_1_name",
            "arn:aws:kinesis:region:account-id:stream/stream_2_name"
         ]
      }
   ]
}
```

You can grant granular or conditional access to resources, for example, by using a wildcard * naming scheme. For more information, see Adding and removing IAM policies in the IAM User Guide.

### Exporting to Kinesis Data Streams

To create a stream that exports to Kinesis Data Streams, your Greengrass components create a stream (p. 530) with an export definition that includes one or more KinesisConfig objects. This object defines export settings, such as the target data stream, batch size, batch interval, and priority.

When your Greengrass components receive data from devices, they append messages (p. 533) that contain a blob of data to the target stream. Then, stream manager exports the data based on the batch settings and priority defined in the stream's export configurations.

Stream manager generates a unique, random UUID as a partition key for each record uploaded to Amazon Kinesis.

### AWS IoT SiteWise asset properties

Stream manager supports automatic exports to AWS IoT SiteWise. AWS IoT SiteWise lets you collect, organize, and analyze data from industrial equipment at scale. For more information, see What is AWS IoT SiteWise? in the AWS IoT SiteWise User Guide.

In the Stream Manager SDK, your Greengrass components use the IoTSiteWiseConfig to define the export configuration for this destination type. For more information, see the SDK reference for your target language:

- IoTSiteWiseConfig in the Python SDK
- IoTSiteWiseConfig in the Java SDK
- IoTSiteWiseConfig in the Node.js SDK

**Note**

AWS also provides the AWS IoT SiteWise connector, which is a pre-built solution that you can use with OPC-UA sources. For more information, see AWS IoT SiteWise connector in the AWS IoT Greengrass Developer Guide.

### Requirements

This export destination has the following requirements:
Export configurations for supported cloud destinations

- Target asset properties in AWS IoT SiteWise must be in the same AWS account and AWS Region as the Greengrass core device.

  **Note**
  For the list of AWS Regions that AWS IoT SiteWise supports, see [AWS IoT SiteWise endpoints and quotas](#) in the [AWS General Reference](#).

- The Authorize core devices to interact with AWS services (p. 627) must allow the iotsitewise:BatchPutAssetPropertyValue permission to target asset properties. The following example policy uses the iotsitewise:assetHierarchyPath condition key to grant access to a target root asset and its children. You can remove the Condition from the policy to allow access to all of your AWS IoT SiteWise assets or specify ARNs of individual assets.

  ```json
  {  
    "Version": "2012-10-17",
    "Statement": [
      {  
        "Effect": "Allow",
        "Action": "iotsitewise:BatchPutAssetPropertyValue",
        "Resource": "*",
        "Condition": {
          "StringLike": {
            "iotsitewise:assetHierarchyPath": [
              "/root node asset ID",
              "/root node asset ID/*"
            ]
          }
        }
      }
    ]
  }
  ```

  You can grant granular or conditional access to resources, for example, by using a wildcard * naming scheme. For more information, see [Adding and removing IAM policies](#) in the [IAM User Guide](#).

  For important security information, see [BatchPutAssetPropertyValue authorization](#) in the [AWS IoT SiteWise User Guide](#).

**Exporting to AWS IoT SiteWise**

To create a stream that exports to AWS IoT SiteWise, your Greengrass components create a stream (p. 530) with an export definition that includes one or more IoTSiteWiseConfig objects. This object defines export settings, such as the batch size, batch interval, and priority.

When your Greengrass components receive asset property data from devices, they append messages that contain the data to the target stream. Messages are JSON-serialized PutAssetPropertyValueEntry objects that contain property values for one or more asset properties. For more information, see [Append message](#) for AWS IoT SiteWise export destinations.

  **Note**
  When you send data to AWS IoT SiteWise, your data must meet the requirements of the BatchPutAssetPropertyValue action. For more information, see [BatchPutAssetPropertyValue](#) in the [AWS IoT SiteWise API Reference](#).

Then, stream manager exports the data based on the batch settings and priority defined in the stream's export configurations.

You can adjust your stream manager settings and Greengrass component logic to design your export strategy. For example:
For near real time exports, set low batch size and interval settings and append the data to the stream when it’s received.

To optimize batching, mitigate bandwidth constraints, or minimize cost, your Greengrass components can pool the timestamp-quality-value (TQV) data points received for a single asset property before appending the data to the stream. One strategy is to batch entries for up to 10 different property-asset combinations, or property aliases, in one message instead of sending more than one entry for the same property. This helps stream manager to remain within AWS IoT SiteWise quotas.

**Amazon S3 objects**

Stream manager supports automatic exports to Amazon S3. You can use Amazon S3 to store and retrieve large amounts of data. For more information, see **What is Amazon S3?** in the Amazon Simple Storage Service Developer Guide.

In the Stream Manager SDK, your Greengrass components use the `S3ExportTaskExecutorConfig` to define the export configuration for this destination type. For more information, see the SDK reference for your target language:

- `S3ExportTaskExecutorConfig` in the Python SDK
- `S3ExportTaskExecutorConfig` in the Java SDK
- `S3ExportTaskExecutorConfig` in the Node.js SDK

**Requirements**

This export destination has the following requirements:

- Target Amazon S3 buckets must be in the same AWS account as the Greengrass core device.
- If a Lambda function that runs in **Greengrass container** mode writes input files to the input file directory, you must mount the directory as a volume in the container with write permissions. This ensures that the files are written to the root file system and visible outside the container.
- The **Authorize core devices to interact with AWS services (p. 627)** must allow the following permissions to the target buckets. For example:

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "s3:PutObject",
            "s3:AbortMultipartUpload",
            "s3:ListMultipartUploadParts"
         ],
         "Resource": [
            "arn:aws:s3:::bucket-1-name/*",
            "arn:aws:s3:::bucket-2-name/*"
         ]
      }
   ]
}
```

You can grant granular or conditional access to resources, for example, by using a wildcard * naming scheme. For more information, see **Adding and removing IAM policies** in the IAM User Guide.
Exporting to Amazon S3

To create a stream that exports to Amazon S3, your Greengrass components use the `S3ExportTaskExecutorConfig` object to configure the export policy. The policy defines export settings, such as the multipart upload threshold and priority. For Amazon S3 exports, stream manager uploads data that it reads from local files on the core device. To initiate an upload, your Greengrass components append an export task to the target stream. The export task contains information about the input file and target Amazon S3 object. Stream manager runs tasks in the sequence that they are appended to the stream.

**Note**
The target bucket must already exist in your AWS account. If an object for the specified key doesn't exist, stream manager creates the object for you.

Stream manager uses the multipart upload threshold property, minimum part size (p. 557) setting, and size of the input file to determine how to upload data. The multipart upload threshold must be greater or equal to the minimum part size. If you want to upload data in parallel, you can create multiple streams.

The keys that specify your target Amazon S3 objects can include valid Java `DateTimeFormatter` strings in `!{timestamp:value}` placeholders. You can use these timestamp placeholders to partition data in Amazon S3 based on the time that the input file data was uploaded. For example, the following key name resolves to a value such as `my-key/2020/12/31/data.txt`.

```
my-key/{timestamp:YYYY}/!{timestamp:MM}/!{timestamp:dd}/data.txt
```

**Note**
If you want to monitor the export status for a stream, first create a status stream and then configure the export stream to use it. For more information, see the section called "Monitor export tasks" (p. 551).

Manage input data

You can author code that IoT applications use to manage the lifecycle of the input data. The following example workflow shows how you might use Greengrass components to manage this data.

1. A local process receives data from devices or peripherals, and then writes the data to files in a directory on the core device. These are the input files for stream manager.
2. A Greengrass component scans the directory and appends an export task (p. 536) to the target stream when a new file is created. The task is a JSON-serialized `S3ExportTaskDefinition` object that specifies the URL of the input file, the target Amazon S3 bucket and key, and optional user metadata.
3. Stream manager reads the input file and exports the data to Amazon S3 in the order of appended tasks. The target bucket must already exist in your AWS account. If an object for the specified key doesn't exist, stream manager creates the object for you.
4. The Greengrass component reads messages (p. 537) from a status stream to monitor the export status. After export tasks are completed, the Greengrass component can delete the corresponding input files. For more information, see the section called "Monitor export tasks" (p. 551).

Monitor export tasks

You can author code that IoT applications use to monitor the status of your Amazon S3 exports. Your Greengrass components must create a status stream and then configure the export stream to write status updates to the status stream. A single status stream can receive status updates from multiple streams that export to Amazon S3.
First, create a stream (p. 530) to use as the status stream. You can configure the size and retention policies for the stream to control the lifespan of the status messages. For example:

- Set Persistence to Memory if you don't want to store the status messages.
- Set StrategyOnFull to OverwriteOldestData so that new status messages are not lost.

Then, create or update the export stream to use the status stream. Specifically, set the status configuration property of the stream's S3ExportTaskExecutorConfig export configuration. This setting tells stream manager to write status messages about the export tasks to the status stream. In the StatusConfig object, specify the name of the status stream and the level of verbosity. The following supported values range from least verbose (ERROR) to most verbose (TRACE). The default is INFO.

- ERROR
- WARN
- INFO
- DEBUG
- TRACE

The following example workflow shows how Greengrass components might use a status stream to monitor export status.

1. As described in the previous workflow, a Greengrass component appends an export task (p. 536) to a stream that's configured to write status messages about export tasks to a status stream. The append operation return a sequence number that represents the task ID.

2. A Greengrass component reads messages (p. 537) sequentially from the status stream, and then filters the messages based on the stream name and task ID or based on an export task property from the message context. For example, the Greengrass component can filter by the input file URL of the export task, which is represented by the S3ExportTaskDefinition object in the message context.

The following status codes indicate that an export task has reached a completed state:

- Success. The upload was completed successfully.
- Failure. Stream manager encountered an error, for example, the specified bucket does not exist. After resolving the issue, you can append the export task to the stream again.
- Canceled. The task was stopped because the stream or export definition was deleted, or the time-to-live (TTL) period of the task expired.

**Note**
The task might also have a status of InProgress or Warning. Stream manager issues warnings when an event returns an error that doesn't affect the execution of the task. For example, a failure to clean up a partial upload returns a warning.

3. After export tasks are completed, the Greengrass component can delete the corresponding input files.

The following example shows how a Greengrass component might read and process status messages.

**Python**

```python
import time
from stream_manager import (
    ReadMessagesOptions,
    Status,
    StatusConfig,
    StatusLevel,
    StatusMessage,
    StreamManagerClient,
)
```
Export configurations for supported cloud destinations

```python
from stream_manager.util import Util

client = StreamManagerClient()

try:
    # Read the statuses from the export status stream
    is_file_uploaded_to_s3 = False
    while not is_file_uploaded_to_s3:
        try:
            messages_list = client.read_messages(
                "StatusStreamName", ReadMessagesOptions(min_message_count=1,
                read_timeout_millis=1000)
            )
            for message in messages_list:
                # Deserialize the status message first.
                status_message = Util.deserialize_json_bytes_to_obj(message.payload,
                    StatusMessage)
                # Check the status of the status message. If the status is "Success",
                # the file was successfully uploaded to S3.
                # If the status was either "Failure" or "Cancelled", the server was
                # unable to upload the file to S3.
                # We will print the message for why the upload to S3 failed from the
                # status message.
                # If the status was "InProgress", the status indicates that the server
                # has started uploading
                # the S3 task.
                if status_message.status == Status.Success:
                    logger.info("Successfully uploaded file at path " + file_url + " to
                        S3.")
                    is_file_uploaded_to_s3 = True
                elif status_message.status == Status.Failure or status_message.status
                        == Status.Canceled:
                    logger.info("Unable to upload file at path " + file_url + " to S3. Message:
                        " + status_message.message)
                    is_file_uploaded_to_s3 = True
            time.sleep(5)
        except StreamManagerException:
            logger.exception("Exception while running")
        except StreamManagerException:
            pass
        # Properly handle errors.
    except ConnectionError or asyncio.TimeoutError:
        pass
        # Properly handle errors.

Python SDK reference: read_messages | StatusMessage

Java

```
while (!isS3UploadComplete) {
    try {
        // Read the statuses from the export status stream
        List<Message> messages = client.readMessages("StatusStreamName",
            new ReadMessagesOptions().withMinMessageCount(1L).withReadTimeoutMillis(1000L));
        for (Message message : messages) {
            // Deserialize the status message first.
            StatusMessage statusMessage = ValidateAndSerialize.deserializeJsonBytesToObj(message.getPayload(),
                StatusMessage.class);
            // Check the status of the status message. If the status is "Success", the file was successfully uploaded to S3.
            // If the status was either "Failure" or "Canceled", the server was unable to upload the file to S3.
            // We will print the message for why the upload to S3 failed from the status message.
            // If the status was "InProgress", the status indicates that the server has started uploading the S3 task.
            if (Status.Success.equals(statusMessage.getStatus())) {
                System.out.println("Successfully uploaded file at path " + FILE_URL + " to S3.");
                isS3UploadComplete = true;
            } else if (Status.Failure.equals(statusMessage.getStatus()) ||
                Status.Canceled.equals(statusMessage.getStatus())) {
                System.out.println(String.format("Unable to upload file at path %s to S3. Message %s",
                    statusMessage.getStatusContext().getS3ExportTaskDefinition().getInputUrl(),
                    statusMessage.getMessage()));
                isS3UploadComplete = true;
            } else if (Status.InProgress.equals(statusMessage.getStatus())) {
                System.out.println(String.format("Upload in progress. " + FILE_URL + " to S3.");
            }
        }
    } catch (StreamManagerException ignored) {
        // Properly handle errors.
    } finally {
        // Sleep for sometime for the S3 upload task to complete before trying to read the status message.
        Thread.sleep(5000);
    }
} catch (e) {
    // Properly handle exception.
}

Java SDK reference: readMessages | StatusMessage

Node.js

const {
    StreamManagerClient, ReadMessagesOptions,
    Status, StatusConfig, StatusLevel, StatusMessage,
    util,
} = require(*'aws-greengrass-stream-manager-sdk'*);

const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        let isS3UploadComplete = false;
        while (!isS3UploadComplete) {
            try {
                // Read the statuses from the export status stream
                const messages = await c.readMessages("StatusStreamName",
                    new ReadMessagesOptions()
Configure AWS IoT Greengrass stream manager

On Greengrass core devices, stream manager can store, process, and export IoT device data. Stream manager provides parameters that you use to configure runtime settings. These settings apply to all streams on the Greengrass core device. You can use the AWS IoT Greengrass console or API to configure stream manager settings when you deploy the component. Changes take effect after the deployment completes.

Stream manager parameters

Stream manager provides the following parameters that you can configure when you deploy the component to your core devices. All parameters are optional.

Storage directory

Parameter name: STREAM_MANAGER_STORE_ROOT_DIR

Configure stream manager

Configure stream manager withMinMessageCount(1) withReadTimeoutMillis(1000));

messages.forEach((message) => {
  // Deserialize the status message first.
  const statusMessage =
    util.deserializeJsonBytesToObject(message.payload, StatusMessage);
  
  // Check the status of the status message. If the status is 'Success', the file was successfully uploaded to S3.
  // If the status was either 'Failure' or 'Cancelled', the server was unable to upload the file to S3.
  // We will print the message for why the upload to S3 failed from the status message.
  // If the status was "InProgress", the status indicates that the server has started uploading the S3 task.
  if (statusMessage.status === Status.Success) {
    console.log('Successfully uploaded file at path ${FILE_URL} to S3.');
    isS3UploadComplete = true;
  } else if (statusMessage.status === Status.Failure ||
    statusMessage.status === Status.Canceled) {
    console.log('Unable to upload file at path ${FILE_URL} to S3. Message: ${statusMessage.message}');
    isS3UploadComplete = true;
  }

  // Sleep for sometime for the S3 upload task to complete before trying to read the status message.
  await new Promise((r) => setTimeout(r, 5000));
}

client.onError((err) => {
  // Properly handle connection errors.
});

Node.js SDK reference: readMessages | StatusMessage
The absolute path of the local directory used to store streams. This value must start with a forward slash (for example, /data).

For information about securing stream data, see the section called “Local data security” (p. 520).

Server port

Parameter name: STREAM_MANAGER_SERVER_PORT

The local port number used to communicate with stream manager. The default is 8088.

You can specify 0 to use a random available port.

Authenticate client

Parameter name: STREAM_MANAGER_AUTHENTICATE_CLIENT

Indicates whether clients must be authenticated to interact with stream manager. All interaction between clients and stream manager is controlled by the Stream Manager SDK. This parameter determines which clients can call the Stream Manager SDK to work with streams. For more information, see the section called “Client authentication” (p. 521).

Valid values are true or false. The default is true (recommended).

- true. Allows only Greengrass components as clients. Components use internal AWS IoT Greengrass Core protocols to authenticate with the Stream Manager SDK.
- false. Allows any process that runs on the AWS IoT Greengrass Core to be a client. Do not set the value to false unless your business case requires it. For example, use false only if non-component processes on the core device must communicate directly with stream manager.

Maximum bandwidth

Parameter name: STREAM_MANAGER_EXPORTER_MAX_BANDWIDTH

The average maximum bandwidth (in kilobits per second) that can be used to export data. The default allows unlimited use of available bandwidth.

Thread pool size

Parameter name: STREAM_MANAGER_EXPORTER_THREAD_POOL_SIZE

The maximum number of active threads that can be used to export data. The default is 5.

The optimal size depends on your hardware, stream volume, and planned number of export streams. If your export speed is slow, you can adjust this setting to find the optimal size for your hardware and business case. The CPU and memory of your core device hardware are limiting factors. To start, you might try setting this value equal to the number of processor cores on the device.

Be careful not to set a size that's higher than your hardware can support. Each stream consumes hardware resources, so try to limit the number of export streams on constrained devices.

JVM arguments

Parameter name: JVM_ARGS

Custom Java Virtual Machine arguments to pass to stream manager at startup. Multiple arguments should be separated by spaces.

Use this parameter only when you must override the default settings used by the JVM. For example, you might need to increase the default heap size if you plan to export a large number of streams.

Logging level

Parameter name: LOG_LEVEL
The logging configuration for the component. Choose from the following log levels, listed here in level order:

- **DEBUG**
- **INFO**
- **WARN**
- **ERROR**

Default: **INFO**

**Minimum size for multipart upload**

Parameter name: `STREAM_MANAGER_EXPORTER_S3_DESTINATION_MULTIPART_UPLOAD_MIN_PART_SIZE_BYTES`

The minimum size (in bytes) of a part in a multipart upload to Amazon S3. Stream manager uses this setting and the size of the input file to determine how to batch data in a multipart PUT request. The default and minimum value is **5242880** bytes (5 MB).

**Note**
Stream manager uses the stream's `sizeThresholdForMultipartUploadBytes` property to determine whether to export to Amazon S3 as a single or multipart upload. User-defined Greengrass components set this threshold when they create a stream that exports to Amazon S3. The default threshold is 5 MB.

**See also**

- Manage data streams on the AWS IoT Greengrass Core (p. 519)
- Use `StreamManagerClient` to work with streams (p. 529)
- Export configurations for supported AWS Cloud destinations (p. 545)
Perform machine learning inference

With AWS IoT Greengrass, you can perform machine learning (ML) inference on your edge devices on locally generated data using cloud-trained models. You benefit from the low latency and cost savings of running local inference, yet still take advantage of cloud computing power for training models and complex processing.

AWS IoT Greengrass simplifies the steps required to perform inference. You can train your inference models anywhere and deploy them locally as machine learning components. For example, you can build and train deep-learning models in Amazon SageMaker and store them in an Amazon S3 bucket. You can then use these models as artifacts in your components to perform inference on your core devices.

Topics
• How AWS IoT Greengrass ML inference works (p. 558)
• What’s different in Version 2? (p. 559)
• Requirements (p. 559)
• Supported model sources (p. 559)
• Supported machine learning runtimes (p. 560)
• AWS-provided machine learning components (p. 560)
• Tutorial: Perform sample image classification inference using TensorFlow Lite (p. 564)
• Perform sample image classification inference on images from a camera using TensorFlow Lite (p. 568)
• Use Amazon SageMaker Edge Manager on Greengrass core devices (p. 573)
• Customize your machine learning components (p. 581)
• Troubleshooting machine learning inference (p. 592)

How AWS IoT Greengrass ML inference works

AWS provides machine learning components (p. 560) that you can use as-is to create one-click deployments to perform machine learning inference on your device. You can also use these components as templates to create custom components to meet your specific requirements.

AWS-provided machine learning components are broadly categorized as follows:
• Model component—Contains machine learning models as Greengrass artifacts.
• Runtime component—Contains the script that installs the machine learning framework and its dependencies on the Greengrass core device.
• Inference component—Contains the inference code and includes component dependencies to install the machine learning framework and download pre-trained machine learning models.

Each deployment that you create to perform machine learning inference consists of at least one component that runs your inference application, installs the machine learning framework, and downloads your machine learning models. To perform sample inference with AWS-provided components, you deploy an inference component to your core device, which automatically includes the corresponding model and runtime components as dependencies. To customize your deployments, you can plug in or swap out the sample model components with custom model components, or you can use the component recipes for the AWS-provided components as templates to create your own custom inference, model, and runtime components.

At a high level, you can complete the following steps to perform machine learning inference using custom components:
1. Create a model component. This component contains the machine learning models that you want to use to perform inference. AWS provides sample pre-trained DLR and TensorFlow Lite models. To use custom model, create your own model component.

2. Create a runtime component. This component contains the scripts required to install the machine learning runtime for your models. AWS provides sample runtime components for Deep Learning Runtime (DLR) and TensorFlow Lite. To use other runtimes with your custom models and inference code, create your own runtime components.

3. Create an inference component. This component contains your inference code, and includes your model and runtime components as dependencies. AWS provides sample inference components for image classification and object detection using DLR and TensorFlow Lite. To perform other types of inference, or to use custom models and runtimes, create your own inference component.

4. Deploy the inference component. When you deploy this component, AWS IoT Greengrass also automatically deploys the model and runtime component dependencies.

To get started with AWS-provided components, see the section called “Perform sample image classification inference” (p. 564).

For information about creating custom machine learning components, see Customize your machine learning components (p. 581).

**What's different in Version 2?**

AWS IoT Greengrass consolidates functional units for machine learning—such as models, runtimes, and inference code—into components that enable you to use a one-click process to install the machine learning runtime, download your trained models, and perform inference on your device.

The AWS-provided machine learning components provide you with the flexibility to quickly get started performing machine learning inference with sample inference code and pre-trained models. You can plug in custom model components to use your own custom-trained models with the inference and runtime components that AWS provides. For a completely customized machine learning solution, you can use the public components as templates to create custom components and use any runtime, model, or inference type that you want.

**Requirements**

The following requirements apply for creating and using machine learning components:

- A Greengrass core device. If you don't have one, see Getting started with AWS IoT Greengrass V2 (p. 24).
- Minimum 500 MB local storage space to use AWS-provided sample machine learning components.

**Supported model sources**

AWS IoT Greengrass supports using custom-trained machine learning models that are stored in Amazon S3. You can also use Amazon SageMaker edge packaging jobs to directly create model components for your SageMaker Neo-compiled models. For information about using SageMaker Edge Manager with AWS IoT Greengrass, see Use Amazon SageMaker Edge Manager on Greengrass core devices (p. 573).

The following requirements apply to the S3 buckets that contain your models:

- S3 buckets must not be encrypted using SSE-C. For buckets that use server-side encryption, AWS IoT Greengrass machine learning inference currently supports the SSE-S3 or SSE-KMS encryption options.
only. For more information about server-side encryption options, see Protecting data using server-side encryption in the Amazon Simple Storage Service Developer Guide.

- The names of S3 buckets must not include periods (\.). For more information, see the rule about using virtual hosted-style buckets with SSL in Rules for bucket naming in the Amazon Simple Storage Service Developer Guide.
- The S3 buckets that store your model sources must be in the same AWS account and AWS Region as your machine learning components.
- AWS IoT Greengrass must have read permission to the model source. To enable AWS IoT Greengrass to access the S3 buckets, the Greengrass device role (p. 627) must allow the s3:GetObject action. For more information about the device role, see Authorize core devices to interact with AWS services (p. 627).

### Supported machine learning runtimes

AWS IoT Greengrass enables you to create custom components to use any machine learning runtime of your choice to perform machine learning inference with your custom-trained models. For information about creating custom machine learning components, see Customize your machine learning components (p. 581).

To simplify the process of getting started with machine learning, AWS IoT Greengrass provides sample inference, model, and runtime components that use the following machine learning runtimes:

- Deep Learning Runtime (DLR) v1.6.0 and v1.3.0
- TensorFlow Lite v2.5.0

### AWS-provided machine learning components

The following table lists the AWS-provided components used for machine learning.

**Note**

Several AWS-provided components depend on specific minor versions of the Greengrass nucleus. Because of this dependency, you need to update these components when you update the Greengrass nucleus to a new minor version. For information about the specific versions of the nucleus that each component depends on, see the corresponding component topic. For more information about updating the nucleus, see Update the AWS IoT Greengrass Core software (OTA) (p. 129).

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Depends on nucleus</th>
<th>Component type (p. 323)</th>
<th>Open source (p. 761)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SageMaker Edge Manager (p. 220)</td>
<td>Deploys the Amazon SageMaker Edge Manager agent on the Greengrass core device.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>DLR image classification (p. 226)</td>
<td>Inference component that uses the DLR image classification</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
<td>Depends on nucleus</td>
<td>Component type (p. 323)</td>
<td>Open source (p. 761)</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>--------------------</td>
<td>-------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>DLR object detection (p. 232)</td>
<td>Inference component that uses the DLR object detection model store and the DLR runtime component as dependencies to install DLR, download sample object detection models, and perform object detection inference on supported devices.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>DLR image classification model store (p. 238)</td>
<td>Model component that contains sample ResNet-50 image classification models as Greengrass artifacts.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>DLR object detection model store (p. 241)</td>
<td>Model component that contains sample YOLOv3 object detection models as Greengrass artifacts.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
</tbody>
</table>
Machine learning components

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Depends on nucleus</th>
<th>Component type (p. 323)</th>
<th>Open source (p. 761)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLR installer (p. 243)</td>
<td>Runtime component that contains an installation script that is used to install DLR and its dependencies on the Greengrass core device.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>TensorFlow Lite image classification (p. 247)</td>
<td>Inference component that uses the TensorFlow Lite image classification model store and the TensorFlow Lite runtime component as dependencies to install TensorFlow Lite, download sample image classification models, and perform image classification inference on supported devices.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
<td>Depends on nucleus</td>
<td>Component type (p. 323)</td>
<td>Open source (p. 761)</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------</td>
<td>--------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>TensorFlow Lite object detection (p. 252)</td>
<td>Inference component that uses the TensorFlow Lite object detection model store and the TensorFlow Lite runtime component as dependencies to install TensorFlow Lite, download sample object detection models, and perform object detection inference on supported devices.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>TensorFlow Lite image classification model store (p. 256)</td>
<td>Model component that contains a sample MobileNet v1 model as a Greengrass artifact.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>TensorFlow Lite object detection model store (p. 258)</td>
<td>Model component that contains a sample Single Shot Detection (SSD) MobileNet model as a Greengrass artifact.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
<tr>
<td>TensorFlow Lite installer (p. 260)</td>
<td>Runtime component that contains an installation script that is used to install TensorFlow Lite and its dependencies on the Greengrass core device.</td>
<td>Yes</td>
<td>Generic</td>
<td>No</td>
</tr>
</tbody>
</table>
Tutorial: Perform sample image classification inference using TensorFlow Lite

This tutorial shows you how to use the TensorFlow Lite image classification (p. 247) inference component to perform sample image classification inference on a Greengrass core device. This component includes the following component dependencies:

- TensorFlow Lite image classification model store component
- TensorFlow Lite component

When you deploy this component, it downloads a pre-trained MobileNet v1 model and installs the TensorFlow Lite runtime and its dependencies. This component publishes inference results on the `ml/tflite/image-classification` topic. To view these inference results, use the AWS IoT MQTT client in the AWS IoT console to subscribe to this topic.

In this tutorial you deploy the sample inference component to perform image classification on the sample image that is provided by AWS IoT Greengrass. After you complete this tutorial, you can complete Perform sample image classification inference on images from a camera using TensorFlow Lite (p. 568), which shows you how to modify the sample inference component to perform image classification on images from a camera locally on a Greengrass core device.

Topics

- Prerequisites (p. 564)
- Step 1: Subscribe to the default notifications topic (p. 564)
- Step 2: Deploy the TensorFlow Lite image classification component (p. 565)
- Step 3: View inference results (p. 566)
- Next steps (p. 567)

Prerequisites

To complete this tutorial, you need the following:

- A Greengrass core device. If you don't have one, see Getting started with AWS IoT Greengrass V2 (p. 24).
- If you are using an Armv7l device such as Raspberry Pi, dependencies for OpenCV Python installed on the device. Run the following command to install the dependencies:

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

Step 1: Subscribe to the default notifications topic

In this step, you configure the AWS IoT MQTT client in the AWS IoT console to watch MQTT messages published by the TensorFlow Lite image classification component. By default, the component publishes inference results on the `ml/tflite/image-classification` topic. Subscribe to this topic before you deploy the component to your Greengrass core device to see the inference results when the component runs for the first time.
To subscribe to the default notifications topic
1. In the AWS IoT console navigation menu, choose Test, MQTT test client.
2. Under Subscribe to a topic, in the Topic name box, enter ml/tflite/image-classification.
3. Choose Subscribe.

Step 2: Deploy the TensorFlow Lite image classification component

In this step, you deploy the TensorFlow Lite image classification component to your core device:

To deploy the TensorFlow Lite image classification component (console)
1. In the AWS IoT Greengrass console navigation menu, choose Components.
4. From Add to deployment, choose one of the following:
   a. To merge this component to an existing deployment on your target device, choose Add to existing deployment, and then select the deployment that you want to revise.
   b. To create a new deployment on your target device, choose Create new deployment. If you have an existing deployment on your device, choosing this step replaces the existing deployment.
5. On the Specify target page, do the following:
   a. Under Deployment information, enter or modify the friendly name for your deployment.
   b. Under Deployment targets, select a target for your deployment, and choose Next. You cannot change the deployment target if you are revising an existing deployment.
6. On the Select components page, under Public components, verify that the aws.greengrass.TensorFlowLiteImageClassification component is selected, and choose Next.
7. On the Configure components page, keep the default configuration settings, and choose Next.
8. On the Configure advanced settings page, keep the default configuration settings, and choose Next.
9. On the Review page, choose Deploy

To deploy the TensorFlow Lite image classification component (AWS CLI)
1. Create a deployment.json file to define the deployment configuration for the TensorFlow Lite image classification component. This file should look like the following:

```json
{
  "targetArn": "targetArn",
  "components": {
    "aws.greengrass.TensorFlowLiteImageClassification": {
      "componentVersion": "2.1.0",
      "configurationUpdate": {}
    }
  }
}
```
• In the targetArn field, replace targetArn with the Amazon Resource Name (ARN) of the thing or thing group to target for the deployment, in the following format:
  • Thing: arn:aws:iot:region:account-id:thing/thingName
  • Thing group: arn:aws:iot:region:account-id:thinggroup/thingGroupName

• This tutorial uses component version 2.1.0. In the aws.greengrass.TensorFlowLiteObjectDetection component object, replace 2.1.0 to use a different version of the TensorFlow Lite object detection component.

2. Run the following command to deploy the TensorFlow Lite image classification component on the device:

   $ aws greengrassv2 create-deployment
   --cli-input-json file://path/to/deployment.json

The deployment can take several minutes to complete. In the next step, check the component log to verify that the deployment completed successfully and to view the inference results.

Step 3: View inference results

After you deploy the component, you can view the inference results in the component log on your Greengrass core device and in the AWS IoT MQTT client in the AWS IoT console. To subscribe to the topic on which the component publishes inference results, see Step 1: Subscribe to the default notifications topic (p. 564).

• AWS IoT MQTT client—To view the results that the inference component publishes on the default notifications topic (p. 564), complete the following steps:

  1. In the AWS IoT console navigation menu, choose Test, MQTT test client.
  2. Under Subscriptions, choose ml/tflite/image-classification.

You should see messages similar to the following example.

```json
{
  "timestamp": "2021-01-01 00:00:00.000000",
  "inference-type": "image-classification",
  "inference-description": "Top 5 predictions with score 0.3 or above ",
  "inference-results": [
    {
      "Label": "cougar, puma, catamount, mountain lion, painter, panther, Felis concolor",
      "Score": "0.5882352941176471"
    },
    {
      "Label": "Persian cat",
      "Score": "0.5882352941176471"
    },
    {
      "Label": "tiger cat",
      "Score": "0.5882352941176471"
    },
    {
      "Label": "dalmatian, coach dog, carriage dog",
      "Score": "0.5607843137254902"
    },
    {
      "Label": "malamute, malemute, Alaskan malamute",
      "Score": "0.5450980392156862"
    }
  ]
}
```
Next steps

• **Component log**—To view the inference results in the component log, run the following command on your Greengrass core device.

```bash
sudo tail -f /greengrass/v2/logs/aws.greengrass.TensorFlowLiteImageClassification.log
```

You should see results similar to the following example.

```json
2021-01-01 00:00:00.000000 [INFO] (Copier) aws.greengrass.TensorFlowLiteImageClassification: stdout. Publishing results to the IoT core....
{scriptIdName=services.aws.greengrass.TensorFlowLiteImageClassification.lifecycle.Run.script, serviceName=aws.greengrass.TensorFlowLiteImageClassification, currentState=RUNNING}

2021-01-01 00:00:00.000000 [INFO] (Copier) aws.greengrass.TensorFlowLiteImageClassification: stdout. "2021-01-01 00:00:00.000000", "inference-type": "image-classification", "inference-description": "Top 5 predictions with score 0.3 or above ", "inference-results": [{"Label": "cougar, puma, catamount, mountain lion, painter, panther, Felis concolor", "Score": "0.5882352941176471"}, {"Label": "Persian cat", "Score": "0.5882352941176471"}, {"Label": "tiger cat", "Score": "0.5882352941176471"}, {"Label": "dalmatian, coach dog, carriage dog", "Score": "0.5607843137254902"}, {"Label": "malamute, malemute, Alaskan malamute", "Score": "0.5450980392156862"}].
{scriptIdName=services.aws.greengrass.TensorFlowLiteImageClassification.lifecycle.Run.script, serviceName=aws.greengrass.TensorFlowLiteImageClassification, currentState=RUNNING}
```

If you can't see inference results in the component log or in the MQTT client, the deployment failed or didn't reach the core device. This can occur if your core device isn't connected to the internet or doesn't have the right permissions to run the component. Run the following command on your core device to view the AWS IoT Greengrass Core software log file. This file includes logs from the Greengrass core device's deployment service.

```bash
sudo tail -f /greengrass/v2/logs/greengrass.log
```

For more information, see Troubleshooting machine learning inference (p. 592).

**Next steps**

If you have a Greengrass core device with a supported camera interface, you can complete Perform sample image classification inference on images from a camera using TensorFlow Lite (p. 568), which shows you how to modify the sample inference component to perform image classification on images from a camera.

To further explore the configuration of the sample TensorFlow Lite image classification (p. 247) inference component, try the following:

- Modify the `InferenceInterval` configuration parameter to change how often the inference code runs.
- Modify the `ImageName` and `ImageDirectory` configuration parameters in the inference component configuration to specify a custom image to use for inference.
Perform sample image classification inference on images from a camera using TensorFlow Lite

This tutorial shows you how to use the TensorFlow Lite image classification (p. 247) inference component to perform sample image classification inference on images from a camera locally on a Greengrass core device. This component includes the following component dependencies:

- TensorFlow Lite image classification model store component
- TensorFlow Lite component

**Note**
This tutorial accesses the camera module for Raspberry Pi, NVIDIA Jetson Nano, or AWS DeepLens devices, but AWS IoT Greengrass supports other devices on Armv7l, Armv8, or x86_64 platforms. To set up a camera for a different device, consult the relevant documentation for your device.

**Topics**
- Prerequisites (p. 568)
- Step 1: Configure the camera module on your device (p. 569)
- Step 2: Verify your subscription to the default notifications topic (p. 571)
- Step 3: Modify the TensorFlow Lite image classification component configuration and deploy it (p. 571)
- Step 4: View inference results (p. 572)
- Next steps (p. 573)

**Prerequisites**
To complete this tutorial, you must first complete Tutorial: Perform sample image classification inference using TensorFlow Lite (p. 564).

You also need the following:

- A Greengrass core device with a camera interface. This tutorial accesses the camera module on one the following supported devices:
  - Raspberry Pi running Raspberry Pi OS (previously called Raspbian)
  - NVIDIA Jetson Nano
  - AWS DeepLens

For information about setting up a Greengrass core device, see Getting started with AWS IoT Greengrass V2 (p. 24).

- For Raspberry Pi or NVIDIA Jetson Nano devices, Raspberry Pi Camera Module V2 - 8 megapixel, 1080p. To learn how to set up the camera, see Connecting the camera in the Raspberry Pi documentation.
- Dependencies for OpenCV Python installed on the device. This requirement is applicable only for Armv7l devices such as the Raspberry Pi that is used in this tutorial. You don't need to meet this requirement for other devices.
Step 1: Configure the camera module on your device

In this step, you install and enable the camera module for your device. Run the following commands on the device.

**Raspberry Pi (Armv7l)**

1. Install the `picamera` interface for the camera module. Run the following command to install the camera module and the other Python libraries that are required for this tutorial.

   ```bash
   sudo apt-get install -y python3-picamera
   ```

2. Verify that Picamera installed successfully.

   ```bash
   sudo -u ggc_user bash -c 'python3 -c "import picamera"'
   ```

   If the output doesn't contain errors, the validation is successful.

   **Note**

   If the Python executable file that is installed on your device is `python3.7`, use `python3.7` instead of `python3` for the commands in this tutorial. Make sure that your pip installation maps to the correct `python3.7` or `python3` version to avoid dependency errors.

3. Reboot the device.

   ```bash
   sudo reboot
   ```

4. Open the Raspberry Pi configuration tool.

   ```bash
   sudo raspi-config
   ```

5. Use the arrow keys to open **Interfacing Options** and enable the camera interface. If prompted, allow the device to reboot.

6. Run the following command to test the camera setup.

   ```bash
   raspistill -v -o test.jpg
   ```

   This opens a preview window on the Raspberry Pi, saves a picture named `test.jpg` to your current directory, and displays information about the camera in the Raspberry Pi terminal.

7. Run the following command to create a symlink to enable the inference component to access your camera from the virtual environment that is created by the runtime component.

   ```bash
   sudo ln -s /usr/lib/python3/dist-packages/picamera $MLRootPath/greengrass_ml_tflite_venv/lib/python3.7/site-packages
   ```

   The default value for `MLRootPath` for this tutorial is `/greengrass/v2/work/variant.TensorFlowLite/greengrass_ml`. The `greengrass_ml_tflite_venv` folder in
Step 1: Configure the camera module on your device

this location is created when you deploy the inference component for the first time in Tutorial: Perform sample image classification inference using TensorFlow Lite (p. 564).

Jetson Nano (Armv8)

1. Run the following command to test the camera setup.

```
gst-launch-1.0 nvarguscamerasrc num-buffers=1 ! "video/x-raw(memory:NVMM), width=1920, height=1080, format=NV12, framerate=30/1" ! nvjpegenc ! filesink location=test.jpg
```

This captures and saves an image named `test.jpg` to your current directory.

2. (Optional) Reboot the device. If you encounter issues when you run the `gst-launch` command in the previous step, rebooting your device might resolve those issues.

```
sudo reboot
```

**Note**
For Armv8 (AArch64) devices, such as a Jetson Nano, you don't need to create a symlink to enable the inference component to access the camera from the virtual environment that is created by the runtime component.

AWS DeepLens (x86_64)

1. Update the `awscam` APT package. Run the following command on the device.

```
sudo apt-get update && sudo apt-get install awscam
```

2. Reboot the device.

```
sudo reboot
```

3. Run each of the following commands to create a symlink to enable the inference component to access your camera from the virtual environment that is created by the runtime component.

```
sudo ln -s /usr/lib/python3/dist-packages/awscam "$MLRootPath/greengrass_ml_tflite_conda/envs/greengrass_ml_tflite_conda/lib/python3.7/site-packages/"
```

```
sudo ln -s /usr/lib/python3/dist-packages/awscamdldt.so "$MLRootPath/greengrass_ml_tflite_conda/envs/greengrass_ml_tflite_conda/lib/python3.7/site-packages/"
```

The default value for MLRootPath for this tutorial is `/greengrass/v2/work/variant.TensorFlowLite/greengrass_ml`. The `greengrass_ml_tflite_conda` folder in this location is created when you deploy the inference component for the first time in Tutorial: Perform sample image classification inference using TensorFlow Lite (p. 564).

For more information about using AWS DeepLens, see AWS DeepLens Developer Guide.
Step 2: Verify your subscription to the default notifications topic

In Tutorial: Perform sample image classification inference using TensorFlow Lite (p. 564), you configured the AWS IoT MQTT client is configured in the AWS IoT console to watch MQTT messages published by the TensorFlow Lite image classification component on the `ml/tflite/image-classification` topic. In the AWS IoT console, verify that this subscription exists. If it doesn’t, follow the steps in Step 1: Subscribe to the default notifications topic (p. 564) to subscribe to this topic before you deploy the component to your Greengrass core device.

Step 3: Modify the TensorFlow Lite image classification component configuration and deploy it

In this step, you configure and deploy the TensorFlow Lite image classification component to your core device:

To configure and deploy the TensorFlow Lite image classification component (console)

1. In the AWS IoT Greengrass console navigation menu, choose Components.
4. From Add to deployment, choose one of the following:
   a. To merge this component to an existing deployment on your target device, choose Add to existing deployment, and then select the deployment that you want to revise.
   b. To create a new deployment on your target device, choose Create new deployment. If you have an existing deployment on your device, choosing this step replaces the existing deployment.
5. On the Specify target page, do the following:
   a. Under Deployment information, enter or modify the friendly name for your deployment.
   b. Under Deployment targets, select a target for your deployment, and choose Next. You cannot change the deployment target if you are revising an existing deployment.
6. On the Select components page, under Public components, verify that the `aws.greengrass.TensorFlowLiteImageClassification` component is selected, and choose Next.
7. On the Configure components page, do the following:
   a. Select the inference component, and choose Configure component.
   b. Under Configuration update, enter the following configuration update in the Configuration to merge box.

   ```json
   {
   "InferenceInterval": "60",
   "UseCamera": "true"
   }
   ``

   With this configuration update, the component accesses the camera module on your device and performs inference on images taken by the camera. The inference code runs every 60 seconds.
   c. Choose Confirm, and then choose Next.
8. On the **Configure advanced settings** page, keep the default configuration settings, and choose **Next**.

9. On the **Review** page, choose **Deploy**

**To configure and deploy the TensorFlow Lite image classification component (AWS CLI)**

1. Create a `deployment.json` file to define the deployment configuration for the TensorFlow Lite image classification component. This file should look like the following:

   ```json
   {
   "targetArn": "targetArn",
   "components": {
      "aws.greengrass.TensorFlowLiteImageClassification": {
         "componentVersion": "2.1.0",
         "configurationUpdate": {
            "InferenceInterval": "60",
            "UseCamera": "true"
         }
      }
   }
   }
   ```

   - In the `targetArn` field, replace `targetArn` with the Amazon Resource Name (ARN) of the thing or thing group to target for the deployment, in the following format:
     - **Thing**: `arn:aws:iot:region:account-id:thing/thingName`
     - **Thing group**: `arn:aws:iot:region:account-id:thinggroup/thingGroupName`
   - This tutorial uses component version 2.1.0. In the `aws.greengrass.TensorFlowLiteImageClassification` component object, replace `2.1.0` to use a different version of the TensorFlow Lite image classification component.

   With this configuration update, the component accesses the camera module on your device and performs inference on images taken by the camera. The inference code runs every 60 seconds.

   Replace the following values

2. Run the following command to deploy the TensorFlow Lite image classification component on the device:

   ```bash
   $ aws greengrassv2 create-deployment \
   --cli-input-json file://path/to/deployment.json
   ```

The deployment can take several minutes to complete. In the next step, check the component log to verify that the deployment completed successfully and to view the inference results.

**Step 4: View inference results**

After you deploy the component, you can view the inference results in the component log on your Greengrass core device and in the AWS IoT MQTT client in the AWS IoT console. To subscribe to the topic on which the component publishes inference results, see Step 2: Verify your subscription to the default notifications topic (p. 571).

- **AWS IoT MQTT client**—To view the results that the inference component publishes on the default notifications topic (p. 564), complete the following steps:
1. In the AWS IoT console navigation menu, choose Test, MQTT test client.
2. Under Subscriptions, choose ml/tflite/image-classification.

- **Component log**—To view the inference results in the component log, run the following command on your Greengrass core device.

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.TensorFlowLiteImageClassification.log
  ```

If you can't see inference results in the component log or in the MQTT client, the deployment failed or didn't reach the core device. This can occur if your core device isn't connected to the internet or doesn't have the required permissions to run the component. Run the following command on your core device to view the AWS IoT Greengrass Core software log file. This file includes logs from the Greengrass core device's deployment service.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

For more information, see Troubleshooting machine learning inference (p. 592).

**Next steps**

This tutorial shows you how to use the TensorFlow Lite image classification component, with custom configuration options to perform sample image classification on images taken by a camera.

For more information about customizing the configuration of public components or creating custom machine learning components, see Customize your machine learning components (p. 581).

**Use Amazon SageMaker Edge Manager on Greengrass core devices**

Amazon SageMaker Edge Manager is a software agent that runs on edge devices. SageMaker Edge Manager provides model management for edge devices so that you can package and use Amazon SageMaker Neo-compiled models directly on Greengrass core devices. By using SageMaker Edge Manager, you can also sample model input and output data from your core devices, and send that data to the AWS Cloud for monitoring and analysis. Because SageMaker Edge Manager uses SageMaker Neo to optimize your models for your target hardware, you don't need to install the DLR runtime directly on your device. On Greengrass devices, SageMaker Edge Manager doesn't load local AWS IoT certificates or call the AWS IoT credential provider endpoint directly. Instead, SageMaker Edge Manager uses the token exchange service (p. 311) to fetch temporary credential from a TES endpoint.

This section describes how SageMaker Edge Manager works on Greengrass devices. It shows you how to get started using SageMaker Edge Manager with AWS-provided sample components on an existing core device. These sample components use the SageMaker Edge Manager component as a dependency to deploy the Edge Manager agent, and perform inference using pre-trained models that were compiled using SageMaker Neo. For more information about the SageMaker Edge Manager agent, see SageMaker Edge Manager in the Amazon SageMaker Developer Guide.
How SageMaker Edge Manager works on Greengrass devices

To deploy the SageMaker Edge Manager agent to your core devices, create a deployment that includes the `aws.greengrass.SageMakerEdgeManager` component. AWS IoT Greengrass manages the installation and lifecycle of the Edge Manager agent on your devices. When a new version of the agent binary is available, deploy the updated version of the `aws.greengrass.SageMakerEdgeManager` component to upgrade the version of the agent that is installed on your device.

When you use SageMaker Edge Manager with AWS IoT Greengrass, your workflow includes the following high-level steps:

1. Compile models with SageMaker Neo.
2. Package your SageMaker Neo-compiled models using SageMaker edge packaging jobs. When you run an edge packaging job for your model, you can choose to create a model component with the packaged model as an artifact that can be deployed to your Greengrass core device.
3. Create a custom inference component. You use this inference component to interact with the Edge Manager agent to perform inference on the core device. These operations include loading models, invoke prediction requests to run inference, and unloading models when the component shuts down.
4. Deploy the SageMaker Edge Manager component, the packaged model component, and the inference component to run your model on the SageMaker inference engine (Edge Manager agent) on your device.

For more information about creating edge packaging jobs and inference components that work with SageMaker Edge Manager, see Create AWS IoT Greengrass V2 Components in the Amazon SageMaker Developer Guide.

The Get started with SageMaker Edge Manager (p. 575) tutorial shows you how to set up and use the SageMaker Edge Manager agent on an existing Greengrass core device, using AWS-provided example code that you can use to create sample inference and model components.

When you use SageMaker Edge Manager on Greengrass core devices, you can also use the capture data feature to upload sample data to the AWS Cloud. Capture data is a SageMaker feature that you use to upload inference input, inference results, and additional inference data to an S3 bucket or a local directory for future analysis. For more information about using capture data with SageMaker Edge Manager, see Manage Model in the Amazon SageMaker Developer Guide.

Requirements

You must meet the following requirements to use the SageMaker Edge Manager agent on Greengrass core devices.

- A Greengrass core device running on a Debian-based Linux platform (x86_64 or Armv8). If you don't have one, see Getting started with AWS IoT Greengrass V2 (p. 24).
- Python 3.6 or later, including pip for your version of Python, installed on your core device.
- The Greengrass device role (p. 627) configured with the following:
  - A trust relationship that allows `credentials.iot.amazonaws.com` and `sagemaker.amazonaws.com` to assume the role, as shown in the following IAM policy example.

```json
{
    "Version": "2012-10-17",
    "Statement": [
    {
        "Effect": "Allow",
        "Resource": "arn:aws:iam::\n```
To set up and use the SageMaker Edge Manager agent on an existing Greengrass core device, AWS provides example code that you can use to create the following sample inference and model components.

**Image classification**
- `com.greengrass.SageMakerEdgeManager.ImageClassification`
- `com.greengrass.SageMakerEdgeManager.ImageClassification.Model`

**Object detection**
- `com.greengrass.SageMakerEdgeManager.ObjectDetection`
- `com.greengrass.SageMakerEdgeManager.ObjectDetection.Model`
Topics

- Prerequisites (p. 576)
- Set up your Greengrass core device in SageMaker Edge Manager (p. 577)
- Create the sample components (p. 578)
- Run sample image classification inference (p. 578)

Prerequisites

To complete this tutorial, you must meet the following prerequisites:

- A Greengrass core device running on a Debian-based Linux platform (x86_64 or Armv8). If you don't have one, see Getting started with AWS IoT Greengrass V2 (p. 24).
- Python 3.6 or later, including pip for your version of Python, installed on your core device.
- The OpenGL API GLX runtime (libgl1-mesa-glx) installed on your core device.
- An AWS Identity and Access Management (IAM) user with administrator permissions.
- An internet-enabled Windows, Mac, or Unix-like development computer that meets the following requirements:
  - Python 3.6 or later installed.
  - AWS CLI installed and configured with your IAM administrator user credentials. For more information, see Installing the AWS CLI and Configuring the AWS CLI.
- The following S3 buckets created in the same AWS account and AWS Region as your Greengrass core device:
  - An S3 bucket to store the artifacts that are included in the sample inference and model components. This tutorial uses **DOC-EXAMPLE-BUCKET1** to refer to this bucket.
  - An S3 bucket that you associate with your SageMaker edge device fleet. SageMaker Edge Manager requires an S3 bucket to create the edge device fleet, and to store sample data from running inference on your device. This tutorial uses **DOC-EXAMPLE-BUCKET2** to refer to this bucket.

For information about creating S3 buckets, see Getting started with Amazon S3.

- The Greengrass device role (p. 627) configured with the following:
  - A trust relationship that allows credentials.iot.amazonaws.com and sagemaker.amazonaws.com to assume the role, as shown in the following IAM policy example.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "credentials.iot.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        },
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "sagemaker.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

- The AmazonSageMakerEdgeDeviceFleetPolicy IAM managed policy.
Get started with SageMaker Edge Manager

- The AmazonSageMakerFullAccess IAM managed policy.
- The s3:GetObject action for the S3 bucket that contains your component artifacts, as shown in the following IAM policy example.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET1/**"
            ],
            "Effect": "Allow"
        }
    ]
}
```

Set up your Greengrass core device in SageMaker Edge Manager

Edge device fleets in SageMaker Edge Manager are collections of logically grouped devices. To use SageMaker Edge Manager with AWS IoT Greengrass, you must create an edge device fleet that uses the same AWS IoT role alias as the Greengrass core device to which you deploy the SageMaker Edge Manager agent. Then, you must register the core device as part of that fleet.

**Topics**
- Create an edge device fleet (p. 577)
- Register your Greengrass core device (p. 577)

Create an edge device fleet

**To create an edge device fleet (console)**

1. In the Amazon SageMaker console, choose Edge Manager, and then choose Edge device fleets.
2. On the Device fleets page, choose Create device fleet.
3. Under Device fleet properties, do the following:
   - For Device fleet name, enter a name for your device fleet.
   - For IAM role, enter the Amazon Resource Name (ARN) of the AWS IoT role alias that you specified when setting up your Greengrass core device.
   - Disable the Create IAM role alias toggle.
4. Choose Next.
5. Under Output configuration, for S3 bucket URI, enter the URI of the S3 bucket that you want to associate with the device fleet.
6. Choose Submit.

Register your Greengrass core device

**To register your Greengrass core device as an edge device (console)**

1. In the Amazon SageMaker console, choose Edge Manager, and then choose Edge devices.
2. On the Devices page, choose Register devices.
3. Under **Device properties**, for **Device fleet name**, enter the name of the device fleet that you created, and then choose **Next**.

4. Choose **Next**.

5. Under **Device source**, for **Device name**, enter the AWS IoT thing name of your Greengrass core device.

6. Choose **Submit**.

### Create the sample components

To help you get started using the SageMaker Edge Manager component, AWS provides a Python script on GitHub that creates the sample inference and model components and uploads them to the AWS Cloud for you. Complete the following steps on a development computer.

**To create the sample components**

1. Download the [AWS IoT Greengrass component examples](https://github.com/aws-iot/greengrass-component-examples) repository on GitHub to your development computer.


   ```
   cd download-directory/machine-learning/sagemaker-edge-manager
   ```

3. Run the following command to create and upload the sample components to the AWS Cloud.

   ```python
   python3 create_components.py -r region -b DOC-EXAMPLE-BUCKET
   ```

   Replace `region` with the AWS Region where you created your Greengrass core device, and replace `DOC-EXAMPLE-BUCKET` with the name of the S3 bucket to store your component artifacts.

   **Note**

   By default, the script creates sample components for both image classification and object detection inference. To create components for only a specific type of inference, specify the `-i ImageClassification | ObjectDetection` argument.

Sample inference and model components for use with SageMaker Edge Manager are now created in your AWS account. To see the sample components in the [AWS IoT Greengrass console](https://console.aws.amazon.com/greengrass/), choose **Components**, and then under **My components**, search for the following components:

- `com.greengrass.SageMakerEdgeManager.ImageClassification`
- `com.greengrass.SageMakerEdgeManager.ImageClassification.Model`
- `com.greengrass.SageMakerEdgeManager.ObjectDetection`
- `com.greengrass.SageMakerEdgeManager.ObjectDetection.Model`

### Run sample image classification inference

To run image classification inference using the AWS-provided sample components and the SageMaker Edge Manager agent, you must deploy these components to your core device. Deploying these components downloads a SageMaker Neo-compiled pre-trained Resnet-50 model and installs the SageMaker Edge Manager agent on your device. The SageMaker Edge Manager agent loads the model and publishes inference results on the `gg/sageMakerEdgeManager/image-classification` topic. To view these inference results, use the AWS IoT MQTT client in the AWS IoT console to subscribe to this topic.

**Topics**
• Subscribe to the notifications topic (p. 579)
• Deploy the sample components (p. 579)
• View inference results (p. 581)

Subscribe to the notifications topic

In this step, you configure the AWS IoT MQTT client in the AWS IoT console to watch MQTT messages published by the sample inference component. By default, the component publishes inference results on the gg/sageMakerEdgeManager/image-classification topic. Subscribe to this topic before you deploy the component to your Greengrass core device to see the inference results when the component runs for the first time.

To subscribe to the default notifications topic

1. In the AWS IoT console navigation menu, choose Test, MQTT test client.
2. Under Subscribe to a topic, in the Topic name box, enter gg/sageMakerEdgeManager/image-classification.
3. Choose Subscribe.

Deploy the sample components

In this step, you configure and deploy the following components to your core device:

• aws.greengrass.SageMakerEdgeManager
• com.greengrass.SageMakerEdgeManager.ImageClassification
• com.greengrass.SageMakerEdgeManager.ImageClassification.Model

To deploy your components (console)

1. In the AWS IoT Greengrass console navigation menu, choose Deployments, and then choose the deployment for your target device that you want to revise.
2. On the deployment page, choose Revise, and then choose Revise deployment.
3. On the Specify target page, choose Next.
4. On the Select components page, do the following:
   a. Under My components, select the following components:
      • com.greengrass.SageMakerEdgeManager.ImageClassification
      • com.greengrass.SageMakerEdgeManager.ImageClassification.Model
   b. Under Public components, turn off the Show only selected components toggle, and then select the aws.greengrass.SageMakerEdgeManager component.
   c. Choose Next.
5. On the Configure components page, select the aws.greengrass.SageMakerEdgeManager component and do the following.
   a. Choose Configure component.
   b. Under Configuration update, in Configuration to merge, enter the following configuration.

```json
{
  "DeviceFleetName": "device-fleet-name",
  "BucketName": "DOC-EXAMPLE-BUCKET"
}
```
Replace `device-fleet-name` with the name of the edge device fleet that you created, and replace `DOC-EXAMPLE-BUCKET` with the name of the S3 bucket that is associated with your device fleet.

c. Choose Confirm, and then choose Next.

6. On the Configure advanced settings page, keep the default configuration settings, and choose Next.

7. On the Review page, choose Deploy

To deploy your components (AWS CLI)

1. On your development computer, create a `deployment.json` file to define the deployment configuration for your SageMaker Edge Manager components. This file should look like the following example.

   ```json
   {
   "targetArn": "targetArn",
   "components": {
   "aws.greengrass.SageMakerEdgeManager": {
   "componentVersion": "1.0.x",
   "configurationUpdate": {
   "merge": {
   "DeviceFleetName": "device-fleet-name",
   "BucketName": "DOC-EXAMPLE-BUCKET"
   }
   }
   },
   "com.greengrass.SageMakerEdgeManager.ImageClassification": {
   "componentVersion": "1.0.x",
   "configurationUpdate": {
   }
   },
   "com.greengrass.SageMakerEdgeManager.ImageClassification.Model": {
   "componentVersion": "1.0.x",
   "configurationUpdate": {
   }
   }
   },
   }
   }
   ```

   - In the `targetArn` field, replace `targetArn` with the Amazon Resource Name (ARN) of the thing or thing group to target for the deployment, in the following format:
     - Thing: `arn:aws:iot:region:account-id:thing/thingName`
     - Thing group: `arn:aws:iot:region:account-id:thinggroup/thingGroupName`
   - In the `merge` field, replace `device-fleet-name` with the name of the edge device fleet that you created. Then, replace `DOC-EXAMPLE-BUCKET` with the name of the S3 bucket that is associated with your device fleet.
   - Replace the component versions for each component with the latest available version.

2. Run the following command to deploy the components on the device:

   ```bash
   $ aws greengrassv2 create-deployment \
   --cli-input-json file:///path/to/deployment.json
   ```

   The deployment can take several minutes to complete. In the next step, check the component log to verify that the deployment completed successfully and to view the inference results.
View inference results

After you deploy the components, you can view the inference results in the component log on your Greengrass core device and in the AWS IoT MQTT client in the AWS IoT console. To subscribe to the topic on which the component publishes inference results, see Subscribe to the notifications topic (p. 579).

- **AWS IoT MQTT client**—To view the results that the inference component publishes on the default notifications topic (p. 579), complete the following steps:
  1. In the AWS IoT console navigation menu, choose Test, MQTT test client.
  2. Under Subscriptions, choose gg/sageMakerEdgeManager/image-classification.

- **Component log**—To view the inference results in the component log, run the following command on your Greengrass core device.

  ```
sudo tail -f /greengrass/v2/logs/com.greengrass.SageMakerEdgeManager.ImageClassification.log
  ```

If you can't see inference results in the component log or in the MQTT client, the deployment failed or didn't reach the core device. This can occur if your core device isn't connected to the internet or doesn't have the right permissions to run the component. Run the following command on your core device to view the AWS IoT Greengrass Core software log file. This file includes logs from the Greengrass core device's deployment service.

  ```
sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

For more information, see Troubleshooting machine learning inference (p. 592).

Customize your machine learning components

In AWS IoT Greengrass, you can configure sample machine learning components (p. 560) to customize how you perform machine learning inference on your devices with the inference, model, and runtime components as the building blocks. AWS IoT Greengrass also provides you the flexibility to use the sample components as templates and create your own custom components as needed. You can mix and match this modular approach to customize your machine learning inference components in the following ways:

**Using sample inference components**

- Modify the configuration of inference components when you deploy them.
- Use a custom model with the sample inference component by replacing the sample model store component with a custom model component. Your custom model must be trained using the same runtime as the sample model.

**Using custom inference components**

- Use custom inference code with the sample models and runtimes by adding public model components and runtime components as dependencies of custom inference components.
- Create and add custom model components or runtime components as dependencies of custom inference components. You must use custom components if you want to use custom inference code or a runtime for which AWS IoT Greengrass doesn't provide a sample component.

Topics
Modify the configuration of a public inference component

In the AWS IoT Greengrass console, the component page displays the default configuration of that component. For example, the default configuration of the TensorFlow Lite image classification component looks like the following:

```json
{
    "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
            "aws.greengrass.TensorFlowLiteImageClassification:mqttproxy:1": {
                "policyDescription": "Allows access to publish via topic ml/tflite/image-classification.",
                "operations": ["aws.greengrass#PublishToIoTCore"],
                "resources": ["ml/tflite/image-classification"]
            }
        }
    },
    "PublishResultsOnTopic": "ml/tflite/image-classification",
    "ImageName": "cat.jpeg",
    "InferenceInterval": 3600,
    "ModelResourceKey": {
        "model": "TensorFlowLite-Mobilenet"
    }
}
```

When you deploy a public inference component, you can modify the default configuration to customize your deployment. For information about the available configuration parameters for each public inference component, see the component topic in AWS-provided machine learning components (p. 560).

This section describes how to deploy a modified component from the AWS IoT Greengrass console. For information about deploying components using the AWS CLI, see Create deployments (p. 384).

To deploy a modified public inference component (console)

1. Sign in to the AWS IoT Greengrass console.
2. In the navigation menu, choose Components.
3. On the Components page, on the Public components tab, choose the component you want to deploy.
4. On the component page, choose Deploy.
5. From Add to deployment, choose one of the following:
   a. To merge this component to an existing deployment on your target device, choose Add to existing deployment, and then select the deployment that you want to revise.
   b. To create a new deployment on your target device, choose Create new deployment. If you have an existing deployment on your device, choosing this step replaces the existing deployment.
6. On the Specify target page, do the following:
Use a custom model with the sample inference component

If you want to use the sample inference component with your own machine learning models for a runtime for which AWS IoT Greengrass provides a sample runtime component, you must override the public model components with components that use those models as artifacts. At a high-level you complete the following steps to use a custom model with the sample inference component:

1. Create a model component that uses a custom model in an S3 bucket as an artifact. Your custom model must be trained using the same runtime as the model that you want to replace.
2. Modify the \texttt{ModelResourceKey} configuration parameter in the inference component to use the custom model. For information about updating the configuration of the inference component, see \textit{Modify the configuration of a public inference component} (p. 582)

When you deploy the inference component, AWS IoT Greengrass looks for the latest version of its component dependencies. It overrides the dependent public model component if a later custom version of the component exists in the same AWS account and AWS Region.

Create a custom model component (console)

1. Upload your model to an S3 bucket. For information about uploading your models to an S3 bucket, see \textit{Working with Amazon S3 Buckets} in the \textit{Amazon Simple Storage Service Developer Guide}.

\textbf{Note}

You must store your artifacts in S3 buckets that are in the same AWS account and AWS Region as the components. To enable AWS IoT Greengrass to access these artifacts, the \textit{Greengrass device role} (p. 627) must allow the \texttt{s3:GetObject} action. For more
information about the device role, see Authorize core devices to interact with AWS services (p. 627).

2. In the AWS IoT Greengrass console navigation menu, choose Components.

3. Retrieve the component recipe for the public model store component.

   a. On the Components page, on the Public components tab, look for and choose the public model component for which you want to create a new version. For example, variant.DLR.ImageClassification.ModelStore.

   b. On the component page, choose View recipe and copy the displayed JSON recipe.

4. On the Components page, on the My components tab, choose Create component.

5. On the Create component page, under Component information, select Enter recipe as JSON as your component source.

6. In the Recipe box, paste the component recipe that you previously copied.

7. In the recipe, update the following values:

   - ComponentVersion: Increment the minor version of the component.
     
     When you create a custom component to override a public model component, you must update only the minor version of the existing component version. For example, if the public component version is 2.1.0, you can create a custom component with version 2.1.1.

   - Manifests.Artifacts.Uri: Update each URI value to the Amazon S3 URI of the model that you want to use.

     Note
     Do not change the name of the component.

8. Choose Create component.

Create a custom model component (AWS CLI)

1. Upload your model to an S3 bucket. For information about uploading your models to an S3 bucket, see Working with Amazon S3 Buckets in the Amazon Simple Storage Service Developer Guide.

   Note
   You must store your artifacts in S3 buckets that are in the same AWS account and AWS Region as the components. To enable AWS IoT Greengrass to access these artifacts, the Greengrass device role (p. 627) must allow the s3:GetObject action. For more information about the device role, see Authorize core devices to interact with AWS services (p. 627).

2. Run the following command to retrieve the component recipe of the public component. This command writes the component recipe to the output file that you provide in your command. Convert the retrieved base64-encoded string to JSON or YAML, as needed.

   Linux, macOS, or Unix

   ```bash
   aws greengrassv2 get-component \\
   --arn <arn> \\
   --recipe-output-format <recipe-format> \\
   --query recipe \\
   --output text | base64 --decode > <recipe-file>
   ```

   Windows command prompt

   ```cmd
   aws greengrassv2 get-component ^
   --arn <arn> ^
   ```
3. Update the name of the recipe file to `<component-name>-<component-version>`, where component version is the target version of the new component. For example, `variant.DLR.ImageClassification.ModelStore-2.1.1.yaml`.

4. In the recipe, update the following values:
   - **ComponentVersion**: Increment the minor version of the component.
     
     When you create a custom component to override a public model component, you must update only the minor version of the existing component version. For example, if the public component version is `2.1.0`, you can create a custom component with version `2.1.1`.
   - **Manifests.Artifacts.Uri**: Update each URI value to the Amazon S3 URI of the model that you want to use.

   **Note**
   Do not change the name of the component.

5. Run the following command to create a new component using the recipe you retrieved and modified.

   ```sh
   aws greengrassv2 create-component-version \
   --inline-recipe fileb://path/to/component/recipe
   ```

   **Note**
   This step creates the component in the AWS IoT Greengrass service in the AWS Cloud. You can use the Greengrass CLI to develop, test, and deploy your component locally before you upload it to the cloud. For more information, see Develop AWS IoT Greengrass components (p. 322).

For more information about creating components, see Develop AWS IoT Greengrass components (p. 322).

---

### Create custom machine learning components

You must create custom components if you want to use custom inference code or a runtime for which AWS IoT Greengrass doesn’t provide a sample component. You can use your custom inference code with the AWS-provided sample machine learning models and runtimes, or you can develop a completely customized machine learning inference solution with your own models and runtime. If your models use a runtime for which AWS IoT Greengrass provides a sample runtime component, then you can use that runtime component, and you need to create custom components only for your inference code and the models you want to use.

**Topics**
- Retrieve the recipe for a public component (p. 586)
- Retrieve sample component artifacts (p. 586)
- Upload component artifacts to an S3 bucket (p. 586)
- Create custom components (p. 587)
Retrieve the recipe for a public component

You can use the recipe of an existing public machine learning component as a template to create a custom component. To view the component recipe for the latest version of a public component, use the console or the AWS CLI as follows:

- **Using the console**
  1. On the Components page, on the Public components tab, look for and choose the public component.
  2. On the component page, choose View recipe.
- **Using AWS CLI**

  Run the following command to retrieve the component recipe of the public variant component. This command writes the component recipe to the JSON or YAML recipe file that you provide in your command.

  **Linux, macOS, or Unix**

  ```bash
  aws greengrassv2 get-component \
  --arn <arn> \
  --recipe-output-format <recipe-format> \
  --query recipe \
  --output text | base64 --decode > <recipe-file>
  ```

  **Windows command prompt**

  ```bash
  aws greengrassv2 get-component ^
  --arn <arn> ^
  --recipe-output-format <recipe-format> ^
  --query recipe ^
  --output text > <recipe-file>.base64
  certutil -decode <recipe-file>.base64 <recipe-file>
  ```

  Replace the values in your command as follows:
  - `<arn>`: The Amazon Resource Name (ARN) of the public component.
  - `<recipe-format>`: The format in which you want to create the recipe file. Supported values are JSON and YAML.
  - `<recipe-file>`: The name of the recipe in the format `<component-name>-<component-version>`.

Retrieve sample component artifacts

You can use the artifacts used by the public machine learning components as templates to create your custom component artifacts, such as inference code or runtime installation scripts.

To view the sample artifacts that are included in the public machine learning components, deploy the public inference component and then view the artifacts on your device in the `/greengrass/v2/packages/artifacts-unarchived/component-name/component-version/` folder.

Upload component artifacts to an S3 bucket

Before you can create a custom component, you must upload the component artifacts to an S3 bucket and use the S3 URIs in your component recipe. For example, to use custom inference code in your
inference component, upload the code to an S3 bucket. You can then use the Amazon S3 URI of your inference code as an artifact in your component.

For information about uploading content to an S3 bucket, see Working with Amazon S3 Buckets in the Amazon Simple Storage Service Developer Guide.

Note
You must store your artifacts in S3 buckets that are in the same AWS account and AWS Region as the components. To enable AWS IoT Greengrass to access these artifacts, the Greengrass device role (p. 627) must allow the s3:GetObject action. For more information about the device role, see Authorize core devices to interact with AWS services (p. 627).

Create custom components

You can use the artifacts and recipes that you retrieved to create your custom machine learning components. For an example, see Create a custom inference component (p. 587).

For detailed information about creating and deploying components to Greengrass devices, see Develop AWS IoT Greengrass components (p. 322) and Deploy AWS IoT Greengrass components to devices (p. 383).

Create a custom inference component

This section shows you how to create a custom inference component using the DLR image classification component as a template.

Topics
- Upload your inference code to an Amazon S3 bucket (p. 587)
- Create a recipe for your inference component (p. 587)
- Create the inference component (p. 591)

Upload your inference code to an Amazon S3 bucket

Create your inference code and then upload it to an S3 bucket. For information about uploading content to an S3 bucket, see Working with Amazon S3 Buckets in the Amazon Simple Storage Service Developer Guide.

Note
You must store your artifacts in S3 buckets that are in the same AWS account and AWS Region as the components. To enable AWS IoT Greengrass to access these artifacts, the Greengrass device role (p. 627) must allow the s3:GetObject action. For more information about the device role, see Authorize core devices to interact with AWS services (p. 627).

Create a recipe for your inference component

1. Run the following command to retrieve the component recipe of the DLR image classification component. This command writes the component recipe to the JSON or YAML recipe file that you provide in your command.

```
aws greengrassv2 get-component \
  --arn

```

587
Create a custom inference component

Windows command prompt

```bash
aws greengrassv2 get-component
--arn
^
--recipe-output-format JSON | YAML ^
--query recipe ^
--output text > <recipe-file>.base64
```

Replace `<recipe-file>` with the name of the recipe in the format `<component-name>-<component-version>`.

2. In the `ComponentDependencies` object in your recipe, do one or more of the following depending on the model and runtime components that you want to use:

- Keep the DLR component dependency if you want to use DLR-compiled models. You can also replace it with a dependency on a custom runtime component, as shown in the following example.

  **Runtime component**

  **JSON**

  ```json
  {
    "<runtime-component>": {
      "VersionRequirement": "<version>",
      "DependencyType": "HARD"
    }
  }
  ```

  **YAML**

  ```yaml
  <runtime-component>:
    VersionRequirement: "<version>
    DependencyType: HARD
  ```

- Keep the DLR image classification model store dependency to use the pre-trained ResNet-50 models that AWS provides, or modify it to use a custom model component. When you include a dependency for a public model component, if a later custom version of the component exists in the same AWS account and AWS Region, then the inference component uses that custom component. Specify the model component dependency as shown in the following examples.

  **Public model component**

  **JSON**

  ```json
  {
    "variant.DLR.ImageClassification.ModelStore": {
      "VersionRequirement": "<version>",
      "DependencyType": "HARD"
    }
  }
  ```
Create a custom inference component

YAML

```
variant.DLR.ImageClassification.ModelStore:
  VersionRequirement: "<version>"
  DependencyType: HARD
```

Custom model component

JSON

```
{
  <custom-model-component>: {
    "VersionRequirement": "<version>",
    "DependencyType": "HARD"
  }
}
```

YAML

```
<custom-model-component>:
  VersionRequirement: "<version>"
  DependencyType: HARD
```

3. In the ComponentConfiguration object, add the default configuration for this component. You can later modify this configuration when you deploy the component. The following excerpt shows the component configuration for the DLR image classification component.

For example, if you use a custom model component as a dependency for your custom inference component, then modify ModelResourceKey to provide the names of the models that you are using.

JSON

```
{
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
      "aws.greengrass.ImageClassification:mqttproxy:1": {
        "policyDescription": "Allows access to publish via topic ml/dlr/image-classification.",
        "operations": [
          "aws.greengrass#PublishToIoTCore"
        ],
        "resources": [
          "ml/dlr/image-classification"
        ]
      }
    },
    "PublishResultsOnTopic": "ml/dlr/image-classification",
    "ImageName": "cat.jpeg",
    "InferenceInterval": 3600,
    "ModelResourceKey": {
      "armv7l": "DLR-resnet50-armv7l-cpu-ImageClassification"
      "x86_64": "DLR-resnet50-x86_64-cpu-ImageClassification"
      "aarch64": "DLR-resnet50-aarch64-cpu-ImageClassification"
    }
  }
}
```
Create a custom inference component

**YAML**

```yaml
accessControl:
  aws.greengrass.ipc.mqttproxy:
    'aws.greengrass.ImageClassification:mqttproxy:1':
      policyDescription: 'Allows access to publish via topic ml/dlr/image-classification.'
      operations:
        - 'aws.greengrass#PublishToIoTCore'
      resources:
        - ml/dlr/image-classification
    PublishResultsOnTopic: ml/dlr/image-classification
    ImageName: cat.jpeg
    InferenceInterval: 3600
    ModelResourceKey:
      armv7l: "DLR-resnet50-armv7l-cpu-ImageClassification"
      x86_64: "DLR-resnet50-x86_64-cpu-ImageClassification"
      aarch64: "DLR-resnet50-aarch64-cpu-ImageClassification"
```

4. In the **Manifests** object, provide information about the artifacts and the configuration of this component that are used when the component is deployed to different platforms and any other information required to successfully run the component. The following excerpt shows the configuration of the **Manifests** object for Linux platform in the DLR image classification component.

**JSON**

```json
{}
```

```json
{  
  "Manifests": [ 
    {  
      "Platform": {  
        "os": "linux",  
        "architecture": "arm"
      },  
      "Name": "32-bit armv7l - Linux (raspberry pi)",  
      "Artifacts": [ 
        {  
          "URI": "s3://SAMPLE-BUCKET/sample-artifacts-directory/image_classification.zip",  
          "Unarchive": "ZIP"
        }
      ],  
      "Lifecycle": {  
        "Setenv": {  
          "DLR_IC_MODEL_DIR": 
            "{variant.DLR.ImageClassification.ModelStore:artifacts:decompressedPath}/
            {configuration:/ModelResourceKey/armv7l}"},  
          "DEFAULT_DLR_IC_IMAGE_DIR": "{artifacts:decompressedPath}/
          image_classification/sample_images/"  
        },  
        "Run": {  
          "RequiresPrivilege": true,  
          "script": ". {variant.DLR:configuration:/MLRootPath}/
          greengrass_ml_dlr_venv/bin/activate\python3 {artifacts:decompressedPath}/
          image_classification/inference.py"
        }
      }
    }
  ]
}
```
Create the inference component

Use the AWS IoT Greengrass console or the AWS CLI to create a component using the recipe you just defined. After you create the component, you can deploy it to perform inference on your device. For an example of how to deploy an inference component, see Tutorial: Perform sample image classification inference using TensorFlow Lite (p. 564).

Create custom inference component (console)

1. Sign in to the AWS IoT Greengrass console.
2. In the navigation menu, choose Components.
3. On the Components page, on the My components tab, choose Create component.
4. On the Create component page, under Component information, select either Enter recipe as JSON or Enter recipe as YAML as your component source.
5. In the Recipe box, enter the custom recipe that you created.
6. Click Create component.

Create custom inference component (AWS CLI)

Run the following command to create a new custom component using the recipe that you created.

```
aws greengrassv2 create-component-version \
--inline-recipe fileb://path/to/recipe/file
```

Note

This step creates the component in the AWS IoT Greengrass service in the AWS Cloud. You can use the Greengrass CLI to develop, test, and deploy your component locally before you upload it to the cloud. For more information, see Develop AWS IoT Greengrass components (p. 322).
Troubleshooting machine learning inference

Use the troubleshooting information and solutions in this section to help resolve issues with your machine learning components. For the public machine learning inference components, you can see error messages in the following component logs:

- /greengrass/v2/logs/aws.greengrass.DLRImageClassification.log
- /greengrass/v2/logs/aws.greengrass.DLRObjectDetection.log
- /greengrass/v2/logs/aws.greengrass.TensorFlowLiteImageClassification.log
- /greengrass/v2/logs/aws.greengrass.TensorFlowLiteObjectDetection.log

If a component is installed correctly, then the component log contains the location of the library that it uses for inference.

Issues

- Failed to fetch library (p. 592)
- Cannot open shared object file (p. 592)
- <library> not found (p. 593)
- No CUDA-capable device is detected (p. 593)
- No such file or directory (p. 593)
- Memory errors (p. 594)
- Disk space errors (p. 594)
- Timeout errors (p. 594)

Failed to fetch library

The following error occurs when the installer script fails to download a required library during deployment on a Raspberry Pi device.

```
```

Run `sudo apt-get update` and deploy your component again.

Cannot open shared object file

You might see errors similar to the following when the installer script fails to download a required dependency for opencv-python during deployment on a Raspberry Pi device.

```
ImportError: libopenjp2.so.7: cannot open shared object file: No such file or directory
```

Run the following command to manually install the dependencies for opencv-python:

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```
<library> not found

The following errors indicate that the runtime component was unable to set up the virtual environment correctly:

- cv2 not found
- dlr not found
- numpy not found

Check the logs to make sure that all runtime dependencies were installed correctly. For more information about the libraries installed by the installer script, see the following topics:

- DLR installer (p. 243)
- TensorFlow Lite installer (p. 260)

No CUDA-capable device is detected

You might see the following error when you use GPU acceleration. Run the following command to enable GPU access for the Greengrass user.

```
sudo usermod -a -G video ggc_user
```

No such file or directory

The following errors indicate that the runtime component was unable to set up the virtual environment correctly:

- `MLRootPath/greengrass_ml_dlr_conda/bin/conda`: No such file or directory
- `MLRootPath/greengrass_ml_dlr_venv/bin/activate`: No such file or directory
- `MLRootPath/greengrass_ml_tflite_conda/bin/conda`: No such file or directory
- `MLRootPath/greengrass_ml_tflite_venv/bin/activate`: No such file or directory

Check the logs to make sure that all runtime dependencies were installed correctly. For more information about the libraries installed by the installer script, see the following topics:

- DLR installer (p. 243)
- TensorFlow Lite installer (p. 260)

By default `MLRootPath` is set to `/greengrass/v2/work/component-name/greengrass_ml`. To change this location, include the DLR installer (p. 243) or TensorFlow Lite installer (p. 260) runtime component directly in your deployment, and specify a modified value for the `MLRootPath` parameter in a configuration merge update. For more information about configuring component, see Update component configurations (p. 390).

**Note**

For the DLR component v1.3.x, you set the `MLRootPath` parameter in the configuration of the inference component, and the default value is `$HOME/greengrass_ml`. 
Memory errors

The following errors typically occur when the device does not have enough memory and the component process is interrupted.

- stderr. Killed.
- exitCode=137

We recommend a minimum of 500 MB of memory to deploy a public machine learning inference component.

Disk space errors

The no space left on device error typically occurs when a device does not have enough storage. Check the disk space that is available on your device and make sure you have enough space before you deploy the component again. We recommend a minimum of 500 MB of free disk space to deploy a public machine learning inference component.

Timeout errors

The public machine learning components download large machine learning model files that are larger than 200 MB. If the download times out during deployment, check your internet connection speed and retry the deployment.
Greengrass Command Line Interface

The Greengrass Command Line Interface (CLI) lets you interact with AWS IoT Greengrass Core on your device to locally develop components and debug issues. For example, you can use the Greengrass CLI to create a local deployment and restart a component on the core device.

Deploy the Greengrass CLI component (p. 166) (aws.greengrass.Cli) to install the Greengrass CLI on your core device.

**Important**
The Greengrass CLI is intended for use only in development environments. Don't deploy the Greengrass CLI on production devices.

**Topics**
- Install the Greengrass CLI (p. 595)
- Greengrass CLI commands (p. 597)

Install the Greengrass CLI

You can install the Greengrass CLI in one of the following ways:

- Use the `--deploy-dev-tools` argument when you first set up AWS IoT Greengrass Core software on your device. You must also specify `--provision true` to apply this argument.
- Deploy the Greengrass CLI component (aws.greengrass.Cli) on your device.

This section describes the steps to deploy the Greengrass CLI component. For information about installing the Greengrass CLI during initial setup, see Getting started with AWS IoT Greengrass V2 (p. 24).

**Prerequisites**

You must meet the following requirements to deploy the Greengrass CLI component:

- AWS IoT Greengrass Core software installed and configured on your core device. For more information, see Getting started with AWS IoT Greengrass V2 (p. 24).
- To use the AWS CLI to deploy the Greengrass CLI, you must have installed and configured the AWS CLI on your Linux device. For more information, see Configuring the AWS CLI in the AWS Command Line Interface User Guide.

After you deploy the Greengrass CLI component, you must run the Greengrass CLI as an authorized user to interact with AWS IoT Greengrass Core software. You can run it as one of the following users:

- The root user (`sudo`).
- A system user that runs AWS IoT Greengrass Core software.
- A system user in an authorized system group. You use the `AuthorizedPosixGroups` configuration parameter when you deploy the Greengrass CLI component to authorize the system group.

**Deploy the Greengrass CLI component**

Complete the following steps to deploy the Greengrass CLI component to your core device:
**To deploy the Greengrass CLI component (console)**

1. Sign in to the AWS IoT Greengrass console.
2. In the navigation menu, choose **Components**.
3. On the **Components** page, on the **Public components** tab, choose `aws.greengrass.Cli`.
4. On the `aws.greengrass.Cli` page, choose **Deploy**.
5. From **Add to deployment**, choose **Create new deployment**.
6. On the **Specify target** page, under **Deployment targets**, in the **Target name** list, choose the Greengrass group that you want to deploy to, and choose **Next**.
7. On the **Select components** page, verify that the `aws.greengrass.Cli` component is selected, and choose **Next**.
8. On the **Configure components** page, keep the default configuration settings, and choose **Next**.
9. On the **Configure advanced settings** page, keep the default configuration settings, and choose **Next**.
10. On the **Review** page, click **Deploy**

**To deploy the Greengrass CLI component (AWS CLI)**

1. On your device, create a `deployment.json` file to define the deployment configuration for the Greengrass CLI component. This file should look like the following:

   ```json
   {
     "targetArn": "targetArn",
     "components": {
       "aws.greengrass.Cli": {
         "componentVersion": "2.4.0",
         "configurationUpdate": {
           "AuthorizedPosixGroups": "<group1>,<group2>,...,<groupN>"
         }
       }
     }
   }
   
   • In the **target** field, replace **targetArn** with the Amazon Resource Name (ARN) of the thing or thing group to target for the deployment, in the following format:
     • Thing: `arn:aws:iot:region:account-id:thing/thingName`
     • Thing group: `arn:aws:iot:region:account-id:thinggroup/thingGroupName`
     • In the `aws.greengrass.Cli` component object, specify values as follows:
       version
       The version of the Greengrass CLI component.
       configurationUpdate.AuthorizedPosixGroups
       Optional. A string that contains a comma-separated list of system groups that you want to authorize to use the Greengrass CLI to interact with the AWS IoT Greengrass Core software. You can specify group names or group IDs. For example, specifying "group1,1002,group3" authorizes three system groups (group1, 1002, and group3) to use the Greengrass CLI.
       If you don't specify any groups to authorize, you can use the Greengrass CLI as the root user (`sudo`) or as the system user that runs the AWS IoT Greengrass Core software.
   2. Run the following command to deploy the Greengrass CLI component on the device:

   ```bash
   # aws greengrassv2 create-deployment \
   --cli-input-json file://path/to/deployment.json
   ```
During installation, the component adds a symbolic link to `greengrass-cli` in the `/greengrass/v2/bin` folder on your device, and you run the Greengrass CLI from this path. To run the Greengrass CLI without its absolute path, add your `/greengrass/v2/bin` folder to your PATH variable. To verify the Greengrass CLI installation, run the following command:

```
$ /greengrass/v2/bin/greengrass-cli help
```

You should see the following output:

```
Usage: greengrass-cli [-hV] [--ggcRootPath=<ggcRootPath>] [COMMAND]
Greengrass command line interface

--ggcRootPath=<ggcRootPath>  The AWS IoT Greengrass V2 root directory.
-h, --help                  Show this help message and exit.
-V, --version               Print version information and exit.

Commands:
  help                        Show help information for a command.
  component                   Retrieve component information and stop or restart
                              components.
  deployment                  Create local deployments and retrieve deployment status.
  logs                        Analyze Greengrass logs.
  get-debug-password          Generate a password for use with the HTTP debug view
                              component.
```

If the `greengrass-cli` isn't found, the deployment might have failed to install the Greengrass CLI. For more information, see Troubleshooting AWS IoT Greengrass V2 (p. 751).

## Greengrass CLI commands

The Greengrass CLI provides a command line interface to interact locally with your AWS IoT Greengrass core device. Greengrass CLI commands use the following format.

```
$ greengrass-cli <command> <subcommand> [arguments]
```

By default, the `greengrass-cli` executable in the `/greengrass/v2/bin/` folder interacts with the version of the AWS IoT Greengrass Core software running in the `/greengrass/v2` folder. If you call an executable that is not placed in this location, or if you want to interact with AWS IoT Greengrass Core software in a different location, then you must use one of the following methods to explicitly specify the root path of the AWS IoT Greengrass Core software that you want to interact with:

- Set the `GGC_ROOT_PATH` environment variable to `/greengrass/v2`.
- Add the `--ggcRootPath` `/greengrass/v2` argument to your command as shown here:

```
$ sudo greengrass-cli --ggcRootPath /greengrass/v2 <command> <subcommand> [arguments]
```

You can use the following arguments with any command:

- Use `--help` for information about a specific Greengrass CLI command.
- Use `--version` for information about the Greengrass CLI version.

This section describes the Greengrass CLI commands and provides examples for these commands. The synopsis for each command shows its arguments and their usage. Optional arguments are shown in square brackets.
Available commands

- component (p. 598)
- deployment (p. 600)
- logs (p. 602)
- get-debug-password (p. 606)

component

Use the component command to interact with local components on your core device.

Subcommands

- details (p. 598)
- list (p. 598)
- restart (p. 599)
- stop (p. 599)

details

Retrieve the version, status, and configuration of one component.

Synopsis

```
$ sudo greengrass-cli component details --name <component-name>
```

Arguments

```
--name, -n. The component name.
```

Output

The following example shows the output produced when you run this command.

```
$ sudo greengrass-cli component details --name MyComponent

Component Name: MyComponent
Version: 1.0.0
State: RUNNING
Configuration: null
```

list

Retrieve the names, component details, and runtime configuration for all root-level components.

Synopsis

```
$ sudo greengrass-cli component list
```

Arguments

None
output

The following example shows the output produced when you run this command.

```
$ sudo greengrass-cli component list
Components currently running in Greengrass:
Component Name: FleetStatusService
Version: 0.0.0
State: RUNNING
Configuration: {"periodicUpdateIntervalSec":86400.0}
Component Name: UpdateSystemPolicyService
Version: 0.0.0
State: RUNNING
Configuration: null
Component Name: aws.greengrass.Nucleus
Version: 2.0.0
State: FINISHED
Configuration: {"awsRegion":"region","runWithDefault":
{"posixUser":"ggc_user:ggc_group"},"telemetry":{}}
Component Name: DeploymentService
Version: 0.0.0
State: RUNNING
Configuration: null
Component Name: TelemetryAgent
Version: 0.0.0
State: RUNNING
Configuration: null
Component Name: aws.greengrass.Cli
Version: 2.0.0
State: RUNNING
Configuration: {"AuthorizedPosixGroups":"ggc_user"}
```

restart

Restart components.

Synopsis

```
$ sudo greengrass-cli component restart --names <component-name>,...
```

Arguments

--names, -n. The component name. At least one component name is required. You can specify additional component names, separating each name with a comma.

Output

None

stop

Stop running components.

Synopsis

```
$ sudo greengrass-cli component stop --names <component-name>,...
```
Arguments

--names, -n. The component name. At least one component name is required. You can specify additional component names if needed, separating each name with a comma.

Output

None

deployment

Use the deployment command to interact with local components on your core device.

Subcommands

• create (p. 600)
• list (p. 602)
• status (p. 602)

create

Create or update a local deployment using specified component recipes, artifacts, and runtime arguments.

Synopsis

```bash
$ sudo greengrass-cli deployment create
--recipeDir path/to/component/recipe
[--artifactDir path/to/artifact/folder]
[--update-config {component-configuration}]
[--groupId <thing-group>]
[--merge "<component-name>=<component-version>"...]
[--runWith "<component-name>:posixUser=<user-name>[:<group-name>]"...]
[--systemLimits "{component-system-resource-limits}"...]
[--remove <component-name>,...]
```

Arguments

• --recipeDir, -r. The full path to the folder that contains the component recipe files.
• --artifactDir, -a. The full path to the folder that contains the artifact files you want to include in your deployment. The artifacts folder must contain the following directory structure:

  /path/to/artifact/folder/<artifact-name>/<version>/<artifacts>

• --update-config, -c. The configuration arguments for the deployment, provided as a JSON string or a JSON file. The JSON string should be in the following format:

```json
{
  "componentName": {
    "MERGE": {"config-key": "config-value"},
    "RESET": ["path/to/reset/"
    }
  }
```

MERGE and RESET are case-sensitive and must be in upper case.
• --groupId, -g. The target thing group for the deployment.
• **--merge**, -m. The name and version of the target component that you want to add or update. You must provide the component information in the format `<component>=<version>`. Use a separate argument for each additional component to specify. If needed, use the **--runWith** argument to provide the `posixUser` and `posixGroup` information for running the component.

• **--runWith**. The `posixUser` and `posixGroup` information for running a generic or Lambda component. You must provide this information in the format `<component>:posixUser=<user>[:<group>]`. For example, `HelloWorld:posixUser=ggc_user:ggc_group`. Use a separate argument for each additional option to specify.

For more information, see Configure the user and group that run components (p. 122).

• **--systemLimits**. The system resource limits to apply to generic and non-containerized Lambda components' processes on the core device. You can configure the maximum amount of CPU and RAM usage that each component's processes can use on the core device. Specify a serialized JSON object or a file path to a JSON file. The JSON object must have the following format:

```json
{
   "componentName": {
      "cpus": cpuTimeLimit,
      "memory": memoryLimitInKb
   }
}
```

You can configure the following system resource limits for each component:

• **cpus** – The maximum amount of CPU time that this component's processes can use on the core device. A core device's total CPU time is equivalent to the device's number of CPU cores. For example, on a core device with 4 CPU cores, you can set this value to 2 to limit this component's processes to 50 percent usage of each CPU core. On a device with 1 CPU core, you can set this value to 0.25 to limit this component's processes to 25 percent usage of the CPU. If you set this value to a number greater than the number of CPU cores, the AWS IoT Greengrass Core software doesn't limit the component's CPU usage.

• **memory** – The maximum amount of RAM (in kilobytes) that this component's processes can use on the core device.

For more information, see Configure system resource limits for components (p. 124).

This feature is available for v2.4.0 and later of the Greengrass nucleus component (p. 136) and Greengrass CLI.

• **--remove**. The name of the target component that you want to remove from a local deployment. To remove a component that was merged from a cloud deployment, you must provide the group ID of the target thing group in the following format:

Greengrass nucleus v2.4.0 and later

```
--remove <component-name> --groupId <group-name>
```

Earlier than v2.4.0

```
--remove <component-name> --groupId thinggroup/<group-name>
```

**Output**

The following example shows the output produced when you run this command.

```
$ sudo greengrass-cli deployment create \
   --merge MyApp1=1.0.0 \
   --merge MyApp2=1.0.0 --runWith MyApp2:posixUser=ggc_user \
```
---remove MyApp3 \
--recipeDir recipes/ \
--artifactDir artifacts/

Local deployment has been submitted! Deployment Id: 44d89f46-1a29-4044-ad89-5151213dfcdbc

### list

Retrieve the status of the last 10 local deployments.

**Synopsis**

```
$ sudo greengrass-cli deployment list
```

**Arguments**

None

**Output**

The following example shows the output produced when you run this command. Depending on the status of your deployment, the output shows one of the following status values: IN_PROGRESS, SUCCEEDED, or FAILED.

```
$ sudo greengrass-cli deployment list
44d89f46-1a29-4044-ad89-5151213dfcbbc: SUCCEEDED
```

### status

Retrieve the status of a specific deployment.

**Synopsis**

```
$ sudo greengrass-cli deployment status -i <deployment-id>
```

**Arguments**

- `i`. The ID of the deployment.

**Output**

The following example shows the output produced when you run this command. Depending on the status of your deployment, the output shows one of the following status values: IN_PROGRESS, SUCCEEDED, or FAILED.

```
$ sudo greengrass-cli deployment status -i 44d89f46-1a29-4044-ad89-5151213dfcbbc
44d89f46-1a29-4044-ad89-5151213dfcbbc: FAILED
```

### logs

Use the `logs` command to analyze Greengrass logs on your core device.
Subcommands

- get (p. 603)
- list-keywords (p. 604)
- list-log-files (p. 605)

get

Collect, filter, and visualize Greengrass log files. This command supports only JSON-formatted log files. You can specify the logging format in the nucleus configuration.

Synopsis

```
$ sudo greengrass-cli logs get
[--log-dir path/to/a/log/folder]
[--log-file path/to/a/log/file]
[--follow true | false ]
[--filter <filter> ]
[--time-window <start-time>,<end-time> ]
[--verbose ]
[--no-color ]
[--before <value> ]
[--after <value> ]
[--syslog ]
[--max-long-queue-size <value> ]
```

Arguments

- --log-dir, -ld. The path to the directory to check for log files. Do not use with --syslog. Use a separate argument for each additional directory to specify. You must use at least one of --log-dir or --log-file. You can also use both arguments in a single command.

- --log-file, -lf. The paths to the log directories you want to use. Use a separate argument for each additional directory to specify. You must use at least one of --log-dir or --log-file. You can also use both arguments in a single command.

- --follow, -fol. Show log updates as they occur. Greengrass CLI continues to run and reads from the specified logs. If you specify a time window, then Greengrass CLI stops monitoring logs after all of the time windows end.

- --filter, -f. The keyword, regular expressions, or key-value pair to use as a filter. Provide this value as a string, a regular expression, or as a key-value pair. Use a separate argument for each additional filter to specify.

When evaluated, multiple filters specified in a single argument are separated by OR operators, and filters specified in additional arguments are combined with AND operators. For example, if your command includes --filter "installed" --filter "name=alpha,name=beta", then Greengrass CLI will filter and display log messages that contain both the keyword installed and a name key that has the values alpha or beta.

- --time-window, -t. The time window for which to show log information. You can use both exact timestamps and relative offsets. You must provide this information in the format <begin-time>,<end-time>. If you do not specify either the begin time or the end time, then the value for that option defaults to the current system date and time. Use a separate argument for each additional time window to specify.

Greengrass CLI supports the following formats for timestamps:

- yyyy-MM-DD, for example, 2020-06-30. The time defaults to 00:00:00 when you use this format.

- yyyyMMdd, for example, 20200630. The time defaults to 00:00:00 when you use this format.
HH:mm:ss, for example, 15:30:45. The date defaults to the current system date when you use this format.

HH:mm:ssSSS, for example, 15:30:45. The date defaults to the current system date when you use this format.

YYYY-MM-DD'T'HH:mm:ss'Z', for example, 2020-06-30T15:30:45Z.

YYYY-MM-DD'T'HH:mm:ss, for example, 2020-06-30T15:30:45.

yyyyMMdd'T'HH:mm:ss.SSS, for example, 2020-06-30T15:30:45.250.

Relative offsets specify a time period offset from the current system time. Greengrass CLI supports the following format for relative offsets: +|-[<value>h|hr|hours][value|m|min|minutes] [value]s|sec|seconds.

For example, the following argument to specify a time window between 1 hour and 2 hours 15 minutes before the current time is --time-window -2h15min,-1hr.

- --verbose. Show all fields from the log messages. Do not use with --syslog.
- --no-color, -nc. Remove color coding. The default color coding for log messages uses bold red text. Supports only UNIX-like terminals because it uses ANSI escape sequences.
- --before, -b. The number of lines to show preceding a matched log entry. Default is 0.
- --after, -a. The number of lines to show following a matched log entry. Default is 0.
- --syslog. Process all log files using the syslog protocol defined by RFC3164. Do not use with --log-dir and --verbose. The syslog protocol uses the following format: "<#Priority> #Timestamp $Host $Logger ($Class): $Message". If you do not specify a log file, then Greengrass CLI reads log messages from the following locations: /var/log/messages, /var/log/syslog, or the /var/log/system.log.
- --max-log-queue-size, -m. The maximum number of log entries to allocate to memory. Use this option to optimize memory usage. Default is 100.

Output

The following example shows the output produced when you run this command.

```
# sudo greengrass-cli logs get --verbose \
--log-file /greengrass/v2/logs/greengrass.log \
--filter deployment.serviceName=DeploymentService \
--filter level=INFO \
--time-window 2020-12-08T01:11:17,2020-12-08T01:11:22

2020-12-08T01:11:17.615Z [INFO] (pool-2-thread-14) com.aws.greengrass.deployment.DeploymentService: Current deployment finished. {DeploymentId=44d89f46-1a29-4044-ad89-5151213dfcbc, serviceName=DeploymentService, currentState=RUNNING}

2020-12-08T01:11:17.675Z [INFO] (pool-2-thread-14) com.aws.greengrass.deployment.IotJobsHelper: Updating status of persisted deployment. {Status=SUCCEEDED, StatusDetails={detailed-deployment-status=SUCCESSFUL}, ThingName=MyThing, JobId=22d89f46-1a29-4044-ad89-5151213dfcbc
```

list-keywords

Show suggested keywords that you can use to filter log files.
**Synopsis**

```
# sudo greengrass-cli logs list-keywords [arguments]
```

**Arguments**

None

**Output**

The following examples show the output produced when you run this command.

```
# sudo greengrass-cli logs list-keywords
Here is a list of suggested keywords for Greengrass log:
level=$str
thread=$str
loggerName=$str
eventType=$str
serviceName=$str
error=$str

# sudo greengrass-cli logs list-keywords --syslog
Here is a list of suggested keywords for syslog:
priority=$int
host=$str
logger=$str
class=$str
```

### list-log-files

Show log files located in a specified directory.

**Synopsis**

```
# sudo greengrass-cli logs list-log-files [arguments]
```

**Arguments**

```
--log-dir, -ld. The path to the directory to check for log files.
```

**Output**

The following examples show the output produced when you run this command.

```
# sudo greengrass-cli logs list-log-files -ld /greengrass/v2/logs/
/greengrass/v2/logs/aws.greengrass.Nucleus.log
/greengrass/v2/logs/main.log
/greengrass/v2/logs/greengrass.log
Total 3 files found.

# sudo greengrass-cli logs list-log-files
No log file found.
```
get-debug-password

Use the get-debug-password command to print a randomly generated password for the local debug console component (p. 199) (aws.greengrass.LocalDebugConsole). The password expires 8 hours after it is generated.

Synopsis

```bash
$ sudo greengrass-cli get-debug-password
```

Arguments

None

Output

The following example shows the output produced when you run this command.

```bash
$ sudo greengrass-cli get-debug-password
Username: debug
Password: KPr0lf9M4uMj6Arq2mxTZJ6mU7PpiczaCeg6y6Cmp5A
Password will expire at: 2020-11-17T16:46:59.772539-08:00
```
Data protection in AWS IoT Greengrass

The AWS shared responsibility model applies to data protection in AWS IoT Greengrass. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. This content includes the security configuration and management tasks for the AWS services that you use. For
Data encryption

AWS IoT Greengrass uses encryption to protect data while in-transit (over the internet or local network) and at rest (stored in the AWS Cloud).

Devices in a AWS IoT Greengrass environment often collect data that’s sent to AWS services for further processing. For more information about data encryption on other AWS services, see the security documentation for that service.

Encryption in transit

AWS IoT Greengrass has two modes of communication where data is in transit:
• the section called “Data in transit over the internet” (p. 609). Communication between a Greengrass core and AWS IoT Greengrass over the internet is encrypted.
• the section called “Data on the core device” (p. 609). Communication between components on the Greengrass core device is not encrypted.

Data in transit over the internet

AWS IoT Greengrass uses Transport Layer Security (TLS) to encrypt all communication over the internet. All data sent to the AWS Cloud is sent over a TLS connection using MQTT or HTTPS protocols, so it is secure by default. AWS IoT Greengrass uses the AWS IoT transport security model. For more information, see Transport security in the AWS IoT Core Developer Guide.

Data on the core device

AWS IoT Greengrass doesn’t encrypt data exchanged locally on the Greengrass core device because the data doesn’t leave the device. This includes communication between user-defined components, the AWS IoT device SDK, and public components, such as stream manager.

Encryption at rest

AWS IoT Greengrass stores your data:

• the section called “Data at rest in the AWS Cloud” (p. 609). This data is encrypted.
• the section called “Data at rest on the Greengrass core” (p. 609). This data is not encrypted (except local copies of your secrets).

Data at rest in the AWS Cloud

AWS IoT Greengrass encrypts customer data stored in the AWS Cloud. This data is protected using AWS KMS keys that are managed by AWS IoT Greengrass.

Data at rest on the Greengrass core

AWS IoT Greengrass relies on Unix file permissions and full-disk encryption (if enabled) to protect data at rest on the core. It is your responsibility to secure the file system and device.

However, AWS IoT Greengrass does encrypt local copies of your secrets retrieved from AWS Secrets Manager. For more information, see the secret manager (p. 285) component.

Key management for the Greengrass core device

It’s the responsibility of the customer to guarantee secure storage of cryptographic (public and private) keys on the Greengrass core device. AWS IoT Greengrass uses public and private keys for the following scenario:

• The IoT client key is used with the IoT certificate to authenticate the Transport Layer Security (TLS) handshake when a Greengrass core connects to AWS IoT Core. For more information, see the section called “Device authentication and authorization” (p. 610).

  Note
  The key and certificate are also referred to as the core private key and the core device certificate.

A Greengrass core supports private key storage using file system permissions. If you use file system-based private keys, you are responsible for their secure storage on the core device.
Device authentication and authorization for AWS IoT Greengrass

Devices in AWS IoT Greengrass environments use X.509 certificates for authentication and AWS IoT policies for authorization. Certificates and policies allow devices to securely connect with each other, AWS IoT Core, and AWS IoT Greengrass.

X.509 certificates are digital certificates that use the X.509 public key infrastructure standard to associate a public key with the identity contained in a certificate. X.509 certificates are issued by a trusted entity called a certificate authority (CA). The CA maintains one or more special certificates called CA certificates that it uses to issue X.509 certificates. Only the certificate authority has access to CA certificates.

AWS IoT policies define the set of operations allowed for AWS IoT devices. Specifically, they allow and deny access to AWS IoT Core and AWS IoT Greengrass data plane operations, such as publishing MQTT messages and retrieving device shadows.

All devices require an entry in the AWS IoT Core registry and an activated X.509 certificate with an attached AWS IoT policy. Devices fall into two categories:

- **Greengrass core devices**
  
  Greengrass core devices use certificates and AWS IoT policies to connect to AWS IoT Core and AWS IoT Greengrass. The certificates and policies also allow AWS IoT Greengrass to deploy components and configurations to core devices.

- **Client devices**
  
  MQTT client devices use certificates and policies to connect to AWS IoT Core and the AWS IoT Greengrass service. This enables client devices to use the AWS IoT Greengrass cloud discovery to find and connect to a Greengrass core device. A client device uses the same certificate to connect to the AWS IoT Core cloud service and core devices. Client devices also use discovery information for mutual authentication with the core device. For more information, see Interact with local IoT devices (p. 477).

**X.509 certificates**

Communication between core devices and client devices and between devices and AWS IoT Core or AWS IoT Greengrass must be authenticated. This mutual authentication is based on registered X.509 device certificates and cryptographic keys.

In an AWS IoT Greengrass environment, devices use certificates with public and private keys for the following Transport Layer Security (TLS) connections:

- The AWS IoT client component on the Greengrass core device that connects to AWS IoT Core and AWS IoT Greengrass over the internet.
- Client devices that connect to AWS IoT Greengrass over the internet to discover core devices.
- The MQTT broker component on the Greengrass core connecting to Greengrass devices in the group over the local network.

AWS IoT Greengrass core devices store certificates in the Greengrass root folder.

**Certificate authority (CA) certificates**

Greengrass core devices and client devices download a root CA certificate used for authentication with the AWS IoT Core and AWS IoT Greengrass services. We recommend that you use an Amazon Trust
AWS IoT Greengrass Developer Guide, Version 2

AWS IoT policies

Services (ATS) root CA certificate, such as Amazon Root CA 1. For more information, see CA certificates for server authentication in the AWS IoT Core Developer Guide.

Client devices also download a Greengrass core device CA certificate. They use this certificate to validate the MQTT server certificate on the core device during mutual authentication.

Certificate rotation on the local MQTT broker

Greengrass core devices generate a local MQTT server certificate that client devices use for mutual authentication. This certificate is signed by the core device CA certificate, which the core device stores in the AWS IoT Greengrass cloud. The MQTT server certificate expires every 7 days. This limited period is based on security best practices. This rotation helps mitigate the threat of an attacker stealing the MQTT server certificate and private key to impersonate the Greengrass core device.

When the MQTT server certificate expires, the Greengrass core device generates a new certificate and restarts the local MQTT broker. When this happens, all client devices connected to the Greengrass core device are disconnected. Client devices can reconnect to the Greengrass core device after a short period of time.

The core device CA certificate expires after 5 years.

AWS IoT policies for data plane operations

Use AWS IoT policies to authorize access to the AWS IoT Core and AWS IoT Greengrass data planes. The AWS IoT Core data plane provides operations for devices, users, and applications. These operations include the ability to connect to AWS IoT Core and subscribe to topics. The AWS IoT Greengrass data plane provides operations for Greengrass devices. These operations include the ability to resolve component dependencies and download public component artifacts.

An AWS IoT policy is a JSON document that's similar to an IAM policy. It contains one or more policy statements that specify the following properties:

- **Effect.** The access mode, which can be Allow or Deny.
- **Action.** The list of actions that are allowed or denied by the policy.
- **Resource.** The list of resources on which the action is allowed or denied.

AWS IoT policies support * as a wildcard character, and treat MQTT wildcard characters (+ and #) as literal strings. For more information about the * wildcard, see Using wildcard in resource ARNs in the AWS Identity and Access Management User Guide.

For more information, see AWS IoT policies and AWS IoT policy actions in the AWS IoT Core Developer Guide.

**Important**

Thing policy variables (iot:Connection.Thing.*) aren't supported for in AWS IoT policies for core devices or Greengrass data plane operations. Instead, you can use a wildcard that matches multiple devices that have similar names. For example, you can specify MyGreengrassDevice* to match MyGreengrassDevice1, MyGreengrassDevice2, and so on.

**Note**

AWS IoT Core enables you to attach AWS IoT policies to thing groups to define permissions for groups of devices. Thing group policies don't allow access to AWS IoT Greengrass data plane operations. To allow a thing access to an AWS IoT Greengrass data plane operation, add the permission to an AWS IoT policy that you attach to the thing's certificate.

AWS IoT Greengrass V2 policy actions

AWS IoT Greengrass V2 defines the following policy actions that Greengrass core devices and client devices can use in AWS IoT policies:
Core device actions

**greengrass:GetComponentVersionArtifact**

Grants permission to get a presigned URL to download a public component artifact.

This permission is evaluated when a core device receives a deployment that specifies a public component that has artifacts. If the core device already has the artifact, it doesn't download the artifact again.

**greengrass:ResolveComponentCandidates**

Grants permission to identify a list of components that meet the component, version, and platform requirements for a deployment. If the requirements conflict, or no components exist that meet the requirements, this operation returns an error and the deployment fails on the device.

This permission is evaluated when a core device receives a deployment that specifies components.

**greengrass:GetDeploymentConfiguration**

Grants permission to get a presigned URL to download a large deployment document.

This permission is evaluated when a core device receives a deployment that specifies a deployment document larger than 7 KB (if the deployment targets a thing) or 31 KB (if the deployment targets a thing group). The deployment document includes component configurations, deployment policies, and deployment metadata. For more information, see Deploy AWS IoT Greengrass components to devices (p. 383).

This feature is available for v2.3.0 and later of the Greengrass nucleus component (p. 136).

**greengrass:VerifyClientDeviceIdentity**

Grants permission to verify the identity of a client device that connects to a core device.

This permission is evaluated when a core device receives an MQTT connection from a client device. The client device presents its AWS IoT device certificate. Then, the core device sends the device certificate to the AWS IoT Greengrass cloud service to verify the client device's identity. For more information, see Interact with local IoT devices (p. 477).

**greengrass:VerifyClientDeviceIoTCertificateAssociation**

Grants permission to verify whether a client device is associated with an AWS IoT certificate.

This permission is evaluated when a core device authorizes a client device to connect over MQTT. For more information, see Interact with local IoT devices (p. 477).

**Note**

For a core device to use this operation, the Greengrass service role (p. 630) must be associated to your AWS account and allow the iot:DescribeCertificate permission.

**greengrass:PutCertificateAuthorities**

Grants permission to upload certificate authority (CA) certificates that client devices can download to verify the core device.

This permission is evaluated when a core device installs and runs the client device auth component (p. 146). This component creates a local certificate authority and uses this operation to upload its CA certificates. Client devices download these CA certificates when they use the Discover (p. 613) operation to find core devices where they can connect. When client devices connect to an MQTT broker on a core device, they use these CA certificates to verify the identity of the core device. For more information, see Interact with local IoT devices (p. 477).

**greengrass:GetConnectivityInfo**

Grants permission to get connectivity information for a core device. This information describes how client devices can connect to the core device.
Update a core device's AWS IoT policy

This permission is evaluated when a core device installs and runs the client device auth component (p. 146). This component uses the connectivity information to generate valid CA certificates to upload to the AWS IoT Greengrass cloud service with the PutCertificateAuthorities (p. 612) operation. Client devices use these CA certificates to verify the identity of the core device. For more information, see Interact with local IoT devices (p. 477).

You can also use this operation on the AWS IoT Greengrass control plane to view connectivity information for a core device. For more information, see GetConnectivityInfo in the AWS IoT Greengrass V1 API Reference.

greengrass:UpdateConnectivityInfo

Grants permission to update connectivity information for a core device. This information describes how client devices can connect to the core device.

This permission is evaluated when a core device runs the IP detector component (p. 177). This component identifies the information that client devices require to connect to the core device on the local network. Then, this component uses this operation to upload the connectivity information to the AWS IoT Greengrass cloud service, so client devices can retrieve this information with the Discover (p. 613) operation. For more information, see Interact with local IoT devices (p. 477).

You can also use this operation on the AWS IoT Greengrass control plane to manually update connectivity information for a core device. For more information, see UpdateConnectivityInfo in the AWS IoT Greengrass V1 API Reference.

Client device actions

greengrass:Discover

Grants permission to discover connectivity information for core devices where a client device can connect. This information describes how the client device can connect to the core devices. A client device can discover only the core devices that you have associated it with by using the BatchAssociateClientDeviceWithCoreDevice operation. For more information, see Interact with local IoT devices (p. 477).

Update a core device's AWS IoT policy

You can use the AWS IoT Greengrass and AWS IoT consoles to view and update a core device's AWS IoT policy.

Note
If you used the AWS IoT Greengrass Core software installer to provision resources (p. 46), your core device has an AWS IoT policy that allows access to all AWS IoT Greengrass actions (greengrass: *). You can follow these steps to restrict access to only the actions that a core device uses.

To review and update a core device's AWS IoT policy

1. In the AWS IoT Greengrass console navigation menu, choose Core devices.
2. On the Core devices page, choose the core device to update.
3. On the core device details page, choose the link to the core device's Thing. This link opens the thing details page in the AWS IoT console.
4. On the thing details page, choose Certificates.
5. In the Certificates tab, choose the thing's active certificate.
7. In the Policies tab, choose the AWS IoT policy to review and update. You can add the required permissions to any policy that is attached to the core device's active certificate.
Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices

The following example policy includes the minimum set of actions required to support basic Greengrass functionality for your core device.

- The policy includes the * wildcard after the core device thing name (For example, \texttt{core-device-thing-name*}). The core device uses the same device certificate to make multiple connections to AWS IoT Core, but the client ID in a connection might not be an exact match of the core device thing name. This wildcard allows the core device to connect when it uses a client ID with a suffix.
- The policy lists the MQTT topics and topic filters that the core device can publish messages to, subscribe to, and receive messages on, including topics used for shadow state. To support message exchange between AWS IoT Core, Greengrass components, and client devices, specify the topics and topic filters that you want to allow. For more information, see Publish/Subscribe policy examples in the \textit{AWS IoT Core Developer Guide}.
- The policy grants permission to publish to the following topic for telemetry data.

\texttt{$\textit{aws/things/core-device-thing-name*/greengrass/health/json}$}

You can remove this permission for core devices where you disable telemetry. For more information, see Gather system health telemetry data from AWS IoT Greengrass core devices (p. 652).

- The policy grants permission to assume an IAM role through an AWS IoT role alias. The core device uses this role, called the token exchange role, to acquire AWS credentials that it can use to authenticate AWS requests. For more information, see Authorize core devices to interact with AWS services (p. 627).

When you install the AWS IoT Greengrass Core software, you create and attach a second AWS IoT policy that includes only this permission. If you include this permission in your core device's primary AWS IoT policy, you can detach and delete the other AWS IoT policy.

- The policy includes a section that allows AWS IoT Core to get, update, and delete the core device's shadow. To allow shadow sync for connected devices in the Greengrass group, specify the target Amazon Resource Names (ARNs) in the Resource list (for example, \texttt{arn:aws:iot:region:account-id:thing/device-name}).

```json

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect"
      ]
    }
  ]
}
```
},
{
  "Effect": "Allow",
  "Action": [  
    "iot:Receive",
    "iot:Publish"
  ],
  "Resource": [  
  ]
},
{
  "Effect": "Allow",
  "Action": [  
    "iot:Subscribe"
  ],
  "Resource": [  
  ]
},
{
  "Effect": "Allow",
  "Action": [  
    "iot:GetThingShadow",
    "iot:UpdateThingShadow",
    "iot:DeleteThingShadow"
  ],
  "Resource": [  
    "arn:aws:iot:region:account-id:thing/core-device-thing-name*"
  ]
},
{
  "Effect": "Allow",
  "Action": [  
    "iot:AssumeRoleWithCertificate"
  ],
},
{
  "Effect": "Allow",
  "Action": [  
    "greengrass:GetComponentVersionArtifact",
    "greengrass:ResolveComponentCandidates",
    "greengrass:GetDeploymentConfiguration"
  ],
  "Resource": "*
  
}
Minimal AWS IoT policy to support client devices

The following example policy includes the minimum set of actions required to support interaction with client devices on a core device. To support client devices, a core device must have the permissions in this AWS IoT policy in addition to the Minimal AWS IoT policy for basic operation (p. 614).

- The policy allows the core device to get, update, and delete the shadow for any AWS IoT thing. These permissions allow the core device to manage and sync shadows for client devices. To follow best security practices, restrict the `iot:GetThingShadow`, `iot:UpdateThingShadow`, and `iot:DeleteThingShadow` operations to the minimal set of client devices that connect to the core device for your use case.
- The policy allows the core device to update its own connectivity information. This permission is required only if you deploy the IP detector component (p. 177) to the core device.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "greengrass:PutCertificateAuthorities",
        "greengrass:VerifyClientDeviceIdentity"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "greengrass:VerifyClientDeviceIoTCertificateAssociation"
      ],
    },
    {
      "Effect": "Allow",
      "Action": [
        "greengrass:GetConnectivityInfo",
        "greengrass:UpdateConnectivityInfo"
      ],
      "Resource": [
        "arn:aws:greengrass:region:account-id:greengrass/things/core-device-thing-name*/connectivityInfo"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:DeleteThingShadow"
      ],
      "Resource": [
        "arn:aws:iot:region:account-id:thing/*"
      ]
    }
  ]
}
```
Minimal AWS IoT policy for client devices

The following example policy includes the minimum set of actions required for a client device to discover core devices where they connect and communicate over MQTT. The client device's AWS IoT policy must include the `greengrass:Discover` action to allow the device to discover connectivity information for its associated Greengrass core devices. In the Resource section, specify the Amazon Resource Name (ARN) of the client device, not the ARN of the Greengrass core device.

- The policy allows communication on all MQTT topics. To follow best security practices, restrict the `iot:Publish`, `iot:Subscribe`, and `iot:Receive` permissions to the minimal set of topics that a client device requires for your use case.

- The policy allows the thing to discover core devices for all AWS IoT things. To follow best security practices, restrict the `greengrass:Discover` permission to the client device's AWS IoT thing or a wildcard that matches a set of AWS IoT things.

**Important**

**Thing policy variables** `{iot:Connection.Thing.*}` aren't supported for in AWS IoT policies for core devices or Greengrass data plane operations. Instead, you can use a wildcard that matches multiple devices that have similar names. For example, you can specify `MyGreengrassDevice*` to match `MyGreengrassDevice1`, `MyGreengrassDevice2`, and so on.

- A client device's AWS IoT policy doesn't typically require permissions for `iot:GetThingShadow`, `iot:UpdateThingShadow`, or `iot:DeleteThingShadow` actions, because the Greengrass core device handles shadow sync operations for client devices. To enable the core device to handle client device shadows, check that the core device's AWS IoT policy allows these actions, and that the Resource section includes the ARNs of the client devices.

```json
{
   "Version": "2012-10-17",
   "Statement": [
   {
      "Effect": "Allow",
      "Action": ["iot:Connect"],
      "Resource": "*"
   },
   {
      "Effect": "Allow",
      "Action": ["iot:Publish"],
   },
   {
      "Effect": "Allow",
      "Action": ["iot:Subscribe"],
   },
   {
      "Effect": "Allow",
      "Action": ["iot:Receive"
```
Identity and access management for AWS IoT Greengrass

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be authenticated (signed in) and authorized (have permissions) to use AWS IoT Greengrass resources. IAM is an AWS service that you can use with no additional charge.

Note
This topic describes IAM concepts and features. For information about IAM features supported by AWS IoT Greengrass, see the section called "How AWS IoT Greengrass works with IAM" (p. 622).

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in AWS IoT Greengrass.

Service user – If you use the AWS IoT Greengrass service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more AWS IoT Greengrass features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in AWS IoT Greengrass, see Troubleshooting identity and access issues for AWS IoT Greengrass (p. 639).

Service administrator – If you're in charge of AWS IoT Greengrass resources at your company, you probably have full access to AWS IoT Greengrass. It's your job to determine which AWS IoT Greengrass features and resources your employees should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with AWS IoT Greengrass, see How AWS IoT Greengrass works with IAM (p. 622).

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to AWS IoT Greengrass. To view example AWS IoT Greengrass identity-based policies that you can use in IAM, see Identity-based policy examples for AWS IoT Greengrass (p. 625).
Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see Signing in to the AWS Management Console as an IAM user or root user in the IAM User Guide.

You must be authenticated (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the AWS Management Console, use your password with your root user email address or your IAM user name. You can access AWS programmatically using your root user or IAM users access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don't use AWS tools, you must sign the request yourself. Do this using Signature Version 4, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 signing process in the AWS General Reference.

Regardless of the authentication method that you use, you might also be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Using multi-factor authentication (MFA) in AWS in the IAM User Guide.

AWS account root user

When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account root user and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

IAM users and groups

An IAM user is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see Managing access keys for IAM users in the IAM User Guide. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

An IAM group is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named IAMAdmins and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see When to create an IAM user (instead of a role) in the IAM User Guide.

IAM roles

An IAM role is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by switching roles. You can assume a role by calling an AWS CLI or AWS API
Managing access using policies

You control access in AWS by creating policies and attaching them to IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. You can sign in as the root user or an IAM user, or you can assume an IAM role. When you then make a request, AWS evaluates the related identity-based or resource-based policies. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the IAM User Guide.

IAM roles with temporary credentials are useful in the following situations:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.

- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as federated users. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated users and roles in the IAM User Guide.

- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the IAM User Guide.

- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.

- **Principal permissions** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform both actions. To see whether an action requires additional dependent actions in a policy, see Actions, Resources, and Condition Keys for AWS IoT Greengrass in the Service Authorization Reference.

- **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the IAM User Guide.

- **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Using an IAM role to grant permissions to applications running on Amazon EC2 instances in the IAM User Guide.

To learn whether to use IAM roles or IAM users, see When to create an IAM role (instead of a user) in the IAM User Guide.
Administrators can use AWS JSON policies to specify who has access to what. That is, which principal can perform actions on what resources, and under what conditions.

Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the \texttt{iam:GetRole} action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

**Identity-based policies**

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the IAM User Guide.

Identity-based policies can be further categorized as inline policies or managed policies. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choosing between managed policies and inline policies in the IAM User Guide.

**Resource-based policies**

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM role trust policies and Amazon S3 bucket policies. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

**Access control lists (ACLs)**

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the Amazon Simple Storage Service Developer Guide.

**Other policy types**

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of entity's identity-based policies and its permissions boundaries. Resource-based policies that specify
the user or role in the Principal field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the IAM User Guide.

- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see How SCPs work in the AWS Organizations User Guide.

- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the IAM User Guide.

### Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the IAM User Guide.

## See also

- the section called “How AWS IoT Greengrass works with IAM” (p. 622)
- the section called “Identity-based policy examples” (p. 625)
- the section called “Troubleshooting identity and access issues” (p. 639)

## How AWS IoT Greengrass works with IAM

Before you use IAM to manage access to AWS IoT Greengrass, you should understand the IAM features that you can use with AWS IoT Greengrass.

<table>
<thead>
<tr>
<th>IAM feature</th>
<th>Supported by Greengrass?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity-based policies with resource-level permissions (p. 623)</td>
<td>Yes</td>
</tr>
<tr>
<td>Resource-based policies (p. 624)</td>
<td>No</td>
</tr>
<tr>
<td>Access control lists (ACLs) (p. 624)</td>
<td>No</td>
</tr>
<tr>
<td>Tags-based authorization (p. 625)</td>
<td>Yes</td>
</tr>
<tr>
<td>Temporary credentials (p. 625)</td>
<td>Yes</td>
</tr>
<tr>
<td>Service-linked roles (p. 625)</td>
<td>No</td>
</tr>
<tr>
<td>Service roles (p. 625)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

For a high-level view of how other AWS services work with IAM, see AWS services that work with IAM in the IAM User Guide.
Identity-based policies for AWS IoT Greengrass

With IAM identity-based policies, you can specify allowed or denied actions and resources and the conditions under which actions are allowed or denied. AWS IoT Greengrass supports specific actions, resources, and condition keys. To learn about all of the elements that you use in a policy, see IAM JSON policy elements reference in the IAM User Guide.

**Actions**

Administrators can use AWS JSON policies to specify who has access to what. That is, which principal can perform actions on what resources, and under what conditions.

The Action element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as permission-only actions that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called dependent actions.

Include actions in a policy to grant permissions to perform the associated operation.

Policy actions for AWS IoT Greengrass use the greengrass: prefix before the action. For example, to allow someone to use the ListCoreDevices API operation to list the core devices in their AWS account, you include the greengrass:ListCoreDevices action in their policy. Policy statements must include either an Action or NotAction element. AWS IoT Greengrass defines its own set of actions that describe tasks that you can perform with this service.

To specify multiple actions in a single statement, list them between brackets ([ ]) and separate them with commas, as follows:

```
"Action": [  
  "greengrass:action1",  
  "greengrass:action2",  
  "greengrass:action3"  
]
```

You can use wildcards (*) to specify multiple actions. For example, to specify all actions that begin with the word List, include the following action:

```
"Action": "greengrass:List*"
```

**Note**

We recommend that you avoid the use of wildcards to specify all available actions for a service. As a best practice, you should grant least privilege and narrowly scope permissions in a policy. For more information, see the section called “Grant minimum possible permissions” (p. 648).

For the complete list of AWS IoT Greengrass actions, see Actions Defined by AWS IoT Greengrass in the IAM User Guide.

**Resources**

Administrators can use AWS JSON policies to specify who has access to what. That is, which principal can perform actions on what resources, and under what conditions.

The Resource JSON policy element specifies the object or objects to which the action applies. Statements must include either a Resource or a NotResource element. As a best practice, specify a resource using its Amazon Resource Name (ARN). You can do this for actions that support a specific resource type, known as resource-level permissions.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.
How AWS IoT Greengrass works with IAM

"Resource": "*"

The following table contains the AWS IoT Greengrass resource ARNs that can be used in the Resource element of a policy statement. For a mapping of supported resource-level permissions for AWS IoT Greengrass actions, see Actions Defined by AWS IoT Greengrass in the IAM User Guide.

Some AWS IoT Greengrass actions (for example, some list operations), cannot be performed on a specific resource. In those cases, you must use the wildcard alone.

"Resource": "*

To specify multiple resource ARNs in a statement, list them between brackets ([ )] and separate them with commas, as follows:

"Resource": [
    "resource-arn1",
    "resource-arn2",
    "resource-arn3"
]

For more information about ARN formats, see Amazon Resource Names (ARNs) and AWS service namespaces in the Amazon Web Services General Reference.

**Condition keys**

Administrators can use AWS JSON policies to specify who has access to what. That is, which principal can perform actions on what resources, and under what conditions.

The Condition element (or Condition block) lets you specify conditions in which a statement is in effect. The Condition element is optional. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple Condition elements in a statement, or multiple keys in a single Condition element, AWS evaluates them using a logical AND operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical OR operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM policy elements: variables and tags in the IAM User Guide.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the IAM User Guide.

**Examples**

To view examples of AWS IoT Greengrass identity-based policies, see the section called “Identity-based policy examples” (p. 625).

**Resource-based policies for AWS IoT Greengrass**

AWS IoT Greengrass does not support resource-based policies (p. 621).

**Access control lists (ACLs)**

AWS IoT Greengrass does not support ACLs (p. 621).
Authorization based on AWS IoT Greengrass tags

You can attach tags to supported AWS IoT Greengrass resources or pass tags in a request to AWS IoT Greengrass. To control access based on tags, you provide tag information in the **Condition element** of a policy using the `aws:ResourceTag/${TagKey}`, `aws:RequestTag/${TagKey}`, or `aws:TagKeys` condition keys. For more information, see [Tag your resources](p. 759).

IAM roles for AWS IoT Greengrass

An **IAM role** is an entity within your AWS account that has specific permissions.

Using temporary credentials with AWS IoT Greengrass

Temporary credentials are used to sign in with federation, assume an IAM role, or to assume a cross-account role. You obtain temporary security credentials by calling AWS STS API operations such as **AssumeRole** or **GetFederationToken**.

On the Greengrass core, temporary credentials for the **device role** (p. 627) are made available to Greengrass components. If your components use the AWS SDK, you don't need to add logic to obtain the credentials because the AWS SDK does this for you.

Service-linked roles

AWS IoT Greengrass does not support **service-linked roles**.

Service roles

This feature allows a service to assume a **service role** on your behalf. This role allows the service to access resources in other services to complete an action on your behalf. Service roles appear in your IAM account and are owned by the account. This means that an IAM administrator can change the permissions for this role. However, doing so might break the functionality of the service.

AWS IoT Greengrass core devices use a service role to allow Greengrass components and Lambda functions to access some of your AWS resources on your behalf. For more information, see the section called "Authorize core devices to interact with AWS services" (p. 627).

AWS IoT Greengrass uses a service role to access some of your AWS resources on your behalf. For more information, see Greengrass service role (p. 630).

Identity-based policy examples for AWS IoT Greengrass

By default, IAM users and roles don't have permission to create or modify AWS IoT Greengrass resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the IAM users or groups that require those permissions.

Policy best practices

Identity-based policies are very powerful. They determine whether someone can create, access, or delete AWS IoT Greengrass resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started using AWS managed policies** – To start using AWS IoT Greengrass quickly, use AWS managed policies to give your employees the permissions they need. These policies are already available in your account and are maintained and updated by AWS. For more information, see Get started using permissions with AWS managed policies in the [IAM User Guide](p. 625).
• **Grant least privilege** – When you create custom policies, grant only the permissions required to perform a task. Start with a minimum set of permissions and grant additional permissions as necessary. Doing so is more secure than starting with permissions that are too lenient and then trying to tighten them later. For more information, see Grant least privilege in the *IAM User Guide*.

• **Enable MFA for sensitive operations** – For extra security, require IAM users to use multi-factor authentication (MFA) to access sensitive resources or API operations. For more information, see Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.

• **Use policy conditions for extra security** – To the extent that it's practical, define the conditions under which your identity-based policies allow access to a resource. For example, you can write conditions to specify a range of allowable IP addresses that a request must come from. You can also write conditions to allow requests only within a specified date or time range, or to require the use of SSL or MFA. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.

### Policy examples

The following example customer-defined policies grant permissions for common scenarios.

#### Examples

- **Allow users to view their own permissions** (p. 626)

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating policies on the JSON tab in the *IAM User Guide*.

#### Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ViewOwnUserInfo",
      "Effect": "Allow",
      "Action": [
        "iam:GetUserPolicy",
        "iam:ListGroupsForUser",
        "iam:ListAttachedUserPolicies",
        "iam:ListUserPolicies",
        "iam:GetUser"
      ],
      "Resource": ["arn:aws:iam::*:user/${aws:username}"]
    },
    {
      "Sid": "NavigateInConsole",
      "Effect": "Allow",
      "Action": [
        "iam:GetGroupPolicy",
        "iam:GetPolicyVersion",
        "iam:GetPolicy",
        "iam:ListAttachedGroupPolicies",
        "iam:ListGroupPolicies",
        "iam:ListPolicyVersions",
        "iam:ListPolicies",
        "iam:ListUsers"
      ],
      "Resource": "*"
    }
  ]
}
```
Authorize core devices to interact with AWS services

AWS IoT Greengrass core devices use the AWS IoT Core credentials provider to authorize calls to AWS services. The AWS IoT Core credentials provider enables devices to use their X.509 certificates as the unique device identity to authenticate AWS requests. This eliminates the need to store an AWS access key ID and secret access key on your AWS IoT Greengrass core devices. For more information, see Authorizing direct calls to AWS services in the AWS IoT Core Developer Guide.

When you run the AWS IoT Greengrass Core software, you can choose to provision the AWS resources that the core device requires. This includes the AWS Identity and Access Management (IAM) role that your core device assumes through the AWS IoT Core credentials provider. Use the `--provision true` argument to configure a role and policies that allow the core device to get temporary AWS credentials. This argument also configures an AWS IoT role alias that points to this IAM role. You can specify the name of the IAM role and AWS IoT role alias to use. If you specify `--provision true` without these other name parameters, the Greengrass core device creates and uses the following default resources:

- **IAM role**: GreengrassV2TokenExchangeRole
  
  This role has a policy named GreengrassV2TokenExchangeRoleAccess and a trust relationship that allows credentials.iot.amazonaws.com to assume the role. The policy includes the minimum permissions for the core device.

  **Important**
  This policy doesn’t include access to files in S3 buckets. You must add permissions to the role to allow core devices to retrieve component artifacts from S3 buckets. For more information, see Allow access to S3 buckets for component artifacts (p. 628).

- **AWS IoT role alias**: GreengrassV2TokenExchangeRoleAlias
  
  This role alias refers to the IAM role.

For more information, see Install the AWS IoT Greengrass Core software (p. 26).

You can also set the role alias for an existing core device. To do so, configure the `iotRoleAlias` configuration parameter of the Greengrass nucleus component (p. 136).

You can acquire temporary AWS credentials for this IAM role to perform AWS operations in your custom components. For more information, see Interact with AWS services (p. 329).

**Topics**
- Service role permissions for core devices (p. 627)
- Allow access to S3 buckets for component artifacts (p. 628)

**Service role permissions for core devices**

The role allows the following service to assume the role:

- credentials.iot.amazonaws.com

If you use the AWS IoT Greengrass Core software to create this role, it uses the following permissions policy to allow core devices to connect and send logs to AWS. The policy’s name defaults to the name of the IAM role ending with Access. For example, if you use the default IAM role name, then this policy’s name is GreengrassV2TokenExchangeRoleAccess.
Authorize core devices to interact with AWS services

Greengrass nucleus v2.4.0 and later

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "iot:DescribeCertificate",
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents",
            "logs:DescribeLogStreams",
            "s3:GetBucketLocation"
         ],
         "Resource": "*
      }
   ]
}
```

Earlier than v2.4.0

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "iot:DescribeCertificate",
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents",
            "logs:DescribeLogStreams",
            "iot:Connect",
            "iot:Publish",
            "iot:Subscribe",
            "iot:Receive",
            "s3:GetBucketLocation"
         ],
         "Resource": "*
      }
   ]
}
```

Allow access to S3 buckets for component artifacts

The default core device role doesn't allow core devices to access S3 buckets. To deploy components that have artifacts in S3 buckets, you must add the s3:GetObject permission to allow core devices to download component artifacts. You can add a new policy to the core device role to grant this permission.

**To add a policy that allows access to component artifacts in Amazon S3**

1. Create a file called `component-artifact-policy.json` and copy the following JSON into the file. This policy allows access to all files in an S3 bucket. Replace `DOC-EXAMPLE-BUCKET` with the name of the S3 bucket to allow the core device to access.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "s3:GetObject"
         ],
         "Resource": "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*"
      }
   ]
}
```
Minimal IAM policy for installer to provision resources

2. Run the following command to create the policy from the policy document in `component-artifact-policy.json`.

```bash
aws iam create-policy \
  --policy-name MyGreengrassV2ComponentArtifactPolicy \
  --policy-document file://component-artifact-policy.json
```

Copy the policy Amazon Resource Name (ARN) from the policy metadata in the output. You use this ARN to attach this policy to the core device role in the next step.

3. Run the following command to attach the policy to the core device role. Replace `GreengrassV2TokenExchangeRole` with the name of the role that you specified when you ran the AWS IoT Greengrass Core software. Then, replace the policy ARN with the ARN from the previous step.

```bash
aws iam attach-role-policy \
  --role-name GreengrassV2TokenExchangeRole \
  --policy-arn arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
```

If the command has no output, it succeeded, and your core device can access artifacts that you upload to this S3 bucket.

**Minimal IAM policy for installer to provision resources**

When you install the AWS IoT Greengrass Core software, you can provision required AWS resources, such as an AWS IoT thing and an IAM role for your device. You can also deploy local development tools to the device. The installer requires AWS credentials so that it can perform these actions in your AWS account. For more information, see Install the AWS IoT Greengrass Core software (p. 45).

The following example policy includes the minimum set of actions that the installer requires to provision these resources. These permissions are required if you specify the `--provision` argument for the installer.

**Note**

The `DeployDevTools` policy statement is required only if you specify the `--deploy-dev-tools` argument for the installer.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:AddThingToThingGroup",
        "iot:AttachPolicy",
        "iot:AttachThingPrincipal",
        "iot:CreateKeysAndCertificate",
        "iot:CreatePolicy",
        "iot:CreateRoleAlias",
        "iot:CreateThing",
        "iot:CreateThingGroup",
        "s3:GetObject"
      ],
      "Resource": "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*
    }
  ]
}
```
Greengrass service role

The Greengrass service role is an AWS Identity and Access Management (IAM) service role that authorizes AWS IoT Greengrass to access resources from AWS services on your behalf. This role makes it possible for AWS IoT Greengrass to verify the identity of client devices and manage core device connectivity information.

**Note**
AWS IoT Greengrass V1 also uses this role to perform essential tasks. For more information, see Greengrass service role in the AWS IoT Greengrass V1 Developer Guide.

To allow AWS IoT Greengrass to access your resources, the Greengrass service role must be associated with your AWS account and specify AWS IoT Greengrass as a trusted entity. The role must include the AWSGreengrassResourceAccessRolePolicy managed policy or a custom policy that defines equivalent permissions for the AWS IoT Greengrass features that you use. AWS maintains this policy, which defines the set of permissions that AWS IoT Greengrass uses to access your AWS resources.

You can reuse the same Greengrass service role across AWS Regions, but you must associate it with your account in every AWS Region where you use AWS IoT Greengrass. If the service role isn't configured in the current AWS Region, core devices fail to verify client devices and fail to update connectivity information.

The following sections describe how to create and manage the Greengrass service role with the AWS Management Console or AWS CLI.

**Topics**
- Manage the Greengrass service role (console) (p. 631)
- Manage the Greengrass service role (CLI) (p. 633)
- See also (p. 635)

Note
In addition to the service role that authorizes service-level access, you assign a *token exchange role* to Greengrass core devices. The token exchange role is a separate IAM role that controls how Greengrass components and Lambda functions on the core device can access AWS services. For more information, see Authorize core devices to interact with AWS services (p. 627).

Manage the Greengrass service role (console)

The AWS IoT console makes it easy to manage your Greengrass service role. For example, when you configure client device discovery for a core device, the console checks whether your AWS account is attached to a Greengrass service role in the current AWS Region. If not, the console can create and configure a service role for you. For more information, see the section called “Create the Greengrass service role” (p. 631).

You can use the AWS IoT console for the following role management tasks:

- Find your Greengrass service role (p. 631)
- Create the Greengrass service role (p. 631)
- Change the Greengrass service role (p. 632)
- Detach the Greengrass service role (p. 632)

Note
The user who is signed in to the console must have permissions to view, create, or change the service role.

Find your Greengrass service role (console)

Use the following steps to find the service role that AWS IoT Greengrass uses in the current AWS Region.

1. Navigate to the AWS IoT console.
2. In the navigation pane, choose **Settings**.
3. Scroll to the **Greengrass service role** section to see your service role and its policies.

   If you don't see a service role, the console can create or configure one for you. For more information, see Create the Greengrass service role (p. 631).

Create the Greengrass service role (console)

The console can create and configure a default Greengrass service role for you. This role has the following properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Greengrass_ServiceRole</td>
</tr>
<tr>
<td>Trusted entity</td>
<td>AWS service: greengrass</td>
</tr>
<tr>
<td>Policy</td>
<td>AWSGreengrassResourceAccessRolePolicy</td>
</tr>
</tbody>
</table>

Note
If you create this role with the **AWS IoT Greengrass V1 device setup script**, the role name is GreengrassServiceRole_random-string.
When you configure client device discovery for a core device, the console checks whether a Greengrass service role is associated with your AWS account in the current AWS Region. If not, the console prompts you to allow AWS IoT Greengrass to read and write to AWS services on your behalf.

If you grant permission, the console checks whether a role named `Greengrass_ServiceRole` exists in your AWS account.

- If the role exists, the console attaches the service role to your AWS account in the current AWS Region.
- If the role doesn’t exist, the console creates a default Greengrass service role and attaches it to your AWS account in the current AWS Region.

**Note**

If you want to create a service role with custom role policies, use the IAM console to create or modify the role. For more information, see [Creating a role to delegate permissions to an AWS service](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_for_service.html) or [Modifying a role](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_for_service.html) in the IAM User Guide. Make sure that the role grants permissions that are equivalent to the `AWSGreengrassResourceAccessRolePolicy` managed policy for the features and resources that you use.

If you create a service role, return to the AWS IoT console and attach the role to your AWS account. You can do this under **Greengrass service role** on the **Settings** page.

### Change the Greengrass service role (console)

Use the following procedure to choose a different Greengrass service role to attach to your AWS account in the AWS Region currently selected in the console.

1. Navigate to the AWS IoT console.
2. In the navigation pane, choose **Settings**.
3. Under **Greengrass service role**, choose **Change role**.
   
   The Update Greengrass service role dialog box opens and shows the IAM roles in your AWS account that define AWS IoT Greengrass as a trusted entity.

4. Choose the Greengrass service role to attach.
5. Choose **Attach role**.

### Detach the Greengrass service role (console)

Use the following procedure to detach the Greengrass service role from your AWS account in the current AWS Region. This revokes permissions for AWS IoT Greengrass to access AWS services in the current AWS Region.

**Important**

Detaching the service role might interrupt active operations.

1. Navigate to the AWS IoT console.
2. In the navigation pane, choose **Settings**.
3. Under **Greengrass service role**, choose **Detach role**.
4. In the confirmation dialog box, choose **Detach**.

**Note**

If you no longer need the role, you can delete it in the IAM console. For more information, see [Deleting roles or instance profiles](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_for_service.html) in the IAM User Guide.

Other roles might allow AWS IoT Greengrass to access your resources. To find all roles that allow AWS IoT Greengrass to assume permissions on your behalf, in the IAM console, on the **Roles** page, look for roles that include `AWS service: greengrass` in the Trusted entities column.
Manage the Greengrass service role (CLI)

In the following procedures, we assume that the AWS Command Line Interface is installed and configured to use your AWS account. For more information, see Installing, updating, and uninstalling the AWS CLI and Configuring the AWS CLI in the AWS Command Line Interface User Guide.

You can use the AWS CLI for the following role management tasks:

Topics

- Get the Greengrass service role (CLI) (p. 633)
- Create the Greengrass service role (CLI) (p. 633)
- Remove the Greengrass service role (CLI) (p. 634)

Note
AWS IoT Greengrass V2 uses AWS IoT Greengrass V1 API operations to manage the Greengrass service role.

Get the Greengrass service role (CLI)

Use the following procedure to find out if a Greengrass service role is associated with your AWS account in an AWS Region.

1. Get the service role. Replace `region` with your AWS Region (for example, us-west-2).

   ```bash
   aws greengrass get-service-role-for-account --region region
   ```

   If a Greengrass service role is already associated with your account, the request returns the following role metadata.

   ```json
   {
     "AssociatedAt": "timestamp",
     "RoleArn": "arn:aws:iam::account-id:role/path/role-name"
   }
   ```

   If the request doesn't return role metadata, then you must create the service role (if it doesn't exist) and associate it with your account in the AWS Region.

Create the Greengrass service role (CLI)

Use the following steps to create a role and associate it with your AWS account.

To create the service role using IAM

1. Create the role with a trust policy that allows AWS IoT Greengrass to assume the role. This example creates a role named `Greengrass_ServiceRole`, but you can use a different name.

   ```bash
   aws iam create-role --role-name Greengrass_ServiceRole --assume-role-policy-document '{
     "Version": "2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Principal": {
           "Service": "greengrass.amazonaws.com"
         }
       }
     ]
   }'
   ```
Windows command prompt

```
aws iam create-role --role-name Greengrass_ServiceRole --assume-role-policy-document "{"Version":"2012-10-17","Statement":[{"Effect":"Allow","Principal":{"Service":"greengrass.amazonaws.com"},"Action":"sts:AssumeRole"}]}
```

2. Copy the role ARN from the role metadata in the output. You use the ARN to associate the role with your account.

3. Attach the **AWSGreengrassResourceAccessRolePolicy** policy to the role.

```
aws iam attach-role-policy --role-name Greengrass_ServiceRole --policy-arn arn:aws:iam::aws:policy/service-role/AWSGreengrassResourceAccessRolePolicy
```

**To associate the service role with your AWS account**

- Associate the role with your account. Replace role-arn with the service role ARN and region with your AWS Region (for example, us-west-2).

```
aws greengrass associate-service-role-to-account --role-arn role-arn --region region
```

If successful, the request returns the following response.

```
{
  "AssociatedAt": "timestamp"
}
```

**Remove the Greengrass service role (CLI)**

Use the following steps to disassociate the Greengrass service role from your AWS account.

- Disassociate the service role from your account. Replace region with your AWS Region (for example, us-west-2).

```
aws greengrass disassociate-service-role-from-account --region region
```

If successful, the following response is returned.

```
{
  "DisassociatedAt": "timestamp"
}
```

**Note**

You should delete the service role if you're not using it in any AWS Region. First use `delete-role-policy` to detach the AWSGreengrassResourceAccessRolePolicy managed policy from the role, and then use `delete-role` to delete the role. For more information, see Deleting roles or instance profiles in the **IAM User Guide**.
See also

- Creating a role to delegate permissions to an AWS service in the IAM User Guide
- Modifying a role in the IAM User Guide
- Deleting roles or instance profiles in the IAM User Guide
- AWS IoT Greengrass commands in the AWS CLI Command Reference
  - associate-service-role-to-account
  - disassociate-service-role-from-account
  - get-service-role-for-account
- IAM commands in the AWS CLI Command Reference
  - attach-role-policy
  - create-role
  - delete-role
  - delete-role-policy

AWS managed policies for AWS IoT Greengrass

To add permissions to users, groups, and roles, it is easier to use AWS managed policies than to write policies yourself. It takes time and expertise to create IAM customer managed policies that provide your team with only the permissions they need. To get started quickly, you can use our AWS managed policies. These policies cover common use cases and are available in your AWS account. For more information about AWS managed policies, see AWS managed policies in the IAM User Guide.

AWS services maintain and update AWS managed policies. You can't change the permissions in AWS managed policies. Services occasionally add additional permissions to an AWS managed policy to support new features. This type of update affects all identities (users, groups, and roles) where the policy is attached. Services are most likely to update an AWS managed policy when a new feature is launched or when new operations become available. Services do not remove permissions from an AWS managed policy, so policy updates won't break your existing permissions.

Additionally, AWS supports managed policies for job functions that span multiple services. For example, the ViewOnlyAccess AWS managed policy provides read-only access to many AWS services and resources. When a service launches a new feature, AWS adds read-only permissions for new operations and resources. For a list and descriptions of job function policies, see AWS managed policies for job functions in the IAM User Guide.

Topics

- AWS managed policy: AWSGreengrassFullAccess (p. 635)
- AWS managed policy: AWSGreengrassReadOnlyAccess (p. 636)
- AWS managed policy: AWSGreengrassResourceAccessRolePolicy (p. 636)
- AWS IoT Greengrass updates to AWS managed policies (p. 638)

AWS managed policy: AWSGreengrassFullAccess

You can attach the AWSGreengrassFullAccess policy to your IAM identities.

This policy grants administrative permissions that allow a principal full access to all AWS IoT Greengrass actions.

Permissions details

This policy includes the following permissions:
• **greengrass** – Allows principals full access to all AWS IoT Greengrass actions.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "greengrass:*"
      ],
      "Resource": "*"
    }
  ]
}
```

**AWS managed policy: AWSGreengrassReadOnlyAccess**

You can attach the AWSGreengrassReadOnlyAccess policy to your IAM identities.

This policy grants read-only permissions that allow a principal to view, but not modify, information in AWS IoT Greengrass. For example, principals with these permissions can view the list of components deployed to a Greengrass core device, but can't create a deployment to change the components that run on that device.

**Permissions details**

This policy includes the following permissions:

• **greengrass** – Allows principals to perform actions that return either a list of items or details about an item. This includes API operations that start with *List* or *Get*.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "greengrass:List*",
        "greengrass:Get*"
      ],
      "Resource": "*"
    }
  ]
}
```

**AWS managed policy: AWSGreengrassResourceAccessRolePolicy**

You can attach the AWSGreengrassResourceAccessRolePolicy policy to your IAM entities. AWS IoT Greengrass also attaches this policy to a service role that allows AWS IoT Greengrass to perform actions on your behalf. For more information, see [Greengrass service role](p. 630).

This policy grants administrative permissions that allow AWS IoT Greengrass to perform essential tasks, such as retrieving your Lambda functions, managing AWS IoT device shadows, and verifying Greengrass client devices.

**Permissions details**

This policy includes the following permissions.
• **greengrass** – Manage Greengrass resources.

• **iot (Shadow)** – Manage AWS IoT shadows that have special identifiers in their names. These permissions are required so that AWS IoT Greengrass can communicate with core devices.

• **iot (DescribeThing and DescribeCertificate)** – Retrieve information about AWS IoT things and certificates. These permissions are required so that AWS IoT Greengrass can verify client devices that connect to a core device. For more information, see [Interact with local IoT devices](#) (p. 477).

• **lambda** – Retrieve information about AWS Lambda functions. This permission is required so that AWS IoT Greengrass V1 can deploy Lambda functions to Greengrass cores. For more information, see [Run Lambda function on the AWS IoT Greengrass core](#) in the **AWS IoT Greengrass V1 Developer Guide**.

• **secretsmanager** – Retrieve the value of AWS Secrets Manager secrets whose names start with `greengrass-`. This permission is required so that AWS IoT Greengrass V1 can deploy Secrets Manager secrets to Greengrass cores. For more information, see [Deploy secrets to the AWS IoT Greengrass core](#) in the **AWS IoT Greengrass V1 Developer Guide**.

• **s3** – Retrieve files objects from S3 buckets whose names contain `greengrass` or `sagemaker`. These permissions are required so that AWS IoT Greengrass V1 can deploy machine learning resources that you store in S3 buckets. For more information, see [Machine learning resources](#) in the **AWS IoT Greengrass V1 Developer Guide**.

• **sagemaker** – Retrieve information about Amazon SageMaker machine learning inference models. This permission is required so that AWS IoT Greengrass V1 can deploy ML models to Greengrass cores. For more information, see [Perform machine learning inference](#) in the **AWS IoT Greengrass V1 Developer Guide**.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "AllowGreengrassAccessToShadows",
         "Action": [
            "iot:DeleteThingShadow",
            "iot:GetThingShadow",
            "iot:UpdateThingShadow"
         ],
         "Effect": "Allow",
         "Resource": [
            "arn:aws:iot:*/*:thing/GG_*",
            "arn:aws:iot:*/*:thing/*-gcm",
            "arn:aws:iot:*/*:thing/*-gda",
            "arn:aws:iot:*/*:thing/*-gci"
         ]
      },
      {
         "Sid": "AllowGreengrassToDescribeThings",
         "Action": [
            "iot:DescribeThing"
         ],
         "Effect": "Allow",
         "Resource": "arn:aws:iot:*/*:thing/*"
      },
      {
         "Sid": "AllowGreengrassToDescribeCertificates",
         "Action": [
            "iot:DescribeCertificate"
         ],
         "Effect": "Allow",
         "Resource": "arn:aws:iot:*/*:cert/*"
      },
      {
         "Sid": "AllowGreengrassToCallGreengrassServices",
         "Action": [
```
AWS IoT Greengrass Developer Guide, Version 2
AWS managed policies

"greengrass:*"

"Sid": "AllowGreengrassToGetLambdaFunctions",
"Action": [
   "lambda:GetFunction",
   "lambda:GetFunctionConfiguration"
]

"Sid": "AllowGreengrassToGetGreengrassSecrets",
"Action": [
   "secretsmanager:GetSecretValue"
]

"Sid": "AllowGreengrassAccessToS3Objects",
"Action": [
   "s3:GetObject"
]

"Sid": "AllowGreengrassAccessToS3BucketLocation",
"Action": [
   "s3:GetBucketLocation"
]

"Sid": "AllowGreengrassAccessToSageMakerTrainingJobs",
"Action": [
   "sagemaker:DescribeTrainingJob"
]

AWS IoT Greengrass updates to AWS managed policies

You can view details about updates to AWS managed policies for AWS IoT Greengrass from the time this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the AWS IoT Greengrass V2 document history page (p. 762).
Troubleshooting identity and access issues for AWS IoT Greengrass

Use the following information to help you diagnose and fix common issues that you might encounter when working with AWS IoT Greengrass and IAM.

Issues
- I'm not authorized to perform an action in AWS IoT Greengrass (p. 639)
- I'm not authorized to perform iam:PassRole (p. 639)
- I'm an administrator and want to allow others to access AWS IoT Greengrass (p. 640)
- I want to allow people outside of my AWS account to access my AWS IoT Greengrass resources (p. 640)

For general troubleshooting help, see Troubleshooting (p. 751).

I'm not authorized to perform an action in AWS IoT Greengrass

If you receive an error that states you're not authorized to perform an action, you must contact your administrator for assistance. Your administrator is the person who provided you with your user name and password.

The following example error occurs when the mateojackson IAM user tries to view details about a core device, but does not have greengrass:GetCoreDevice permissions.


In this case, Mateo asks his administrator to update his policies to allow him to access the arn:aws:greengrass:us-west-2:123456789012:coreDevices/MyGreengrassCore resource using the greengrass:GetCoreDevice action.

The following are general IAM issues that you might encounter when working with AWS IoT Greengrass.

I'm not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the iam:PassRole action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password. Ask that person to update your policies to allow you to pass a role to AWS IoT Greengrass.

Some AWS services allow you to pass an existing role to that service, instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named marymajor tries to use the console to perform an action in AWS IoT Greengrass. However, the action requires the service to have permissions granted by a service role. Mary does not have permissions to pass the role to the service.
Allow device traffic through a proxy or firewall

Greengrass core devices and Greengrass components perform outbound requests to AWS services and other websites. As a security measure, you might limit outbound traffic to a small range of endpoints and ports. You can use the following information about endpoints and ports to limit device traffic through a proxy or firewall. For more information about how to configure a core device to use a proxy, see Connect on port 443 or through a network proxy (p. 125).

Topics

- Endpoints for basic operation (p. 640)
- Endpoints for installation with automatic provisioning (p. 642)
- Endpoints for AWS-provided components (p. 643)

Endpoints for basic operation

Greengrass core devices use the following endpoints and ports for basic operation.

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>greengrass-ats.iot.region.amazonaws.com</td>
<td>8443 or 443</td>
<td>Yes</td>
<td>Used for data plane operations, such as installing deployments and working</td>
</tr>
</tbody>
</table>
## Endpoints for basic operation

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix-ats.iot.region.amazonaws.com</td>
<td>MQTT: 8883 or 443 HTTPS: 8443 or 443</td>
<td>Yes</td>
<td>Used for data plane operations for device management, such as MQTT communication and shadow sync with AWS IoT Core.</td>
</tr>
<tr>
<td>prefix.credentials.iot.region.amazonaws.com</td>
<td></td>
<td>Yes</td>
<td>Used to acquire AWS credentials, which the core device uses to download component artifacts from Amazon S3 and perform other operations. For more information, see [Authorize core devices to interact with AWS services](p. 627).</td>
</tr>
<tr>
<td>*.s3.amazonaws.com</td>
<td>443</td>
<td>Yes</td>
<td>Used for deployments. This format includes the * character, because endpoint prefixes are controlled internally and might change at any time.</td>
</tr>
</tbody>
</table>
Endpoints for installation with automatic provisioning

Greengrass core devices use the following endpoints and ports when you install the AWS IoT Greengrass Core software with automatic resource provisioning (p. 46).

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data.iot.<em>region</em>.amazonaws.com</td>
<td>443</td>
<td>No</td>
<td>Required if the core device runs a version of the Greengrass nucleus (p. 136) earlier than v2.4.0 and is configured to use a network proxy. The core device uses this endpoint for MQTT communication with AWS IoT Core when behind a proxy. For more information, see Configure a network proxy (p. 127).</td>
</tr>
<tr>
<td>iam.amazonaws.com</td>
<td>443</td>
<td>Yes</td>
<td>Used to create AWS IoT resources and retrieve information about existing AWS IoT resources.</td>
</tr>
<tr>
<td>iot.<em>region</em>.amazonaws.com</td>
<td>443</td>
<td>Yes</td>
<td>Used to create AWS IoT resources and retrieve information about existing AWS IoT resources.</td>
</tr>
</tbody>
</table>
Endpoints for AWS-provided components

Greengrass core devices use additional endpoints depending on which software components they run. You can find the endpoints that each AWS-provided component requires in the Requirements section on each component's page in this developer guide. For more information, see AWS-provided components (p. 132).

### Endpoints for AWS-provided components

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sts.region.amazonaws.com</td>
<td>443</td>
<td>Yes</td>
<td>Used to get the ID of your AWS account.</td>
</tr>
<tr>
<td>greengrass.region.amazonaws.com</td>
<td>443</td>
<td>No</td>
<td>Required if you use the --deploy-dev-tools argument to deploy the Greengrass CLI component to the core device.</td>
</tr>
</tbody>
</table>

## Compliance validation for AWS IoT Greengrass

To learn whether AWS IoT Greengrass or other AWS services are in scope of specific compliance programs, see AWS Services in Scope by Compliance Program. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- **Security and Compliance Quick Start Guides** – These deployment guides discuss architectural considerations and provide steps for deploying baseline environments on AWS that are security and compliance focused.
- **Architecting for HIPAA Security and Compliance Whitepaper** – This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.

  **Note**
  Not all services are compliant with HIPAA.

- **AWS Compliance Resources** – This collection of workbooks and guides might apply to your industry and location.
Resilience in AWS IoT Greengrass

The AWS global infrastructure is built around Amazon Web Services Regions and Availability Zones. Each AWS Region provides multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information, see AWS Global Infrastructure.

In addition to the AWS global infrastructure, AWS IoT Greengrass offers several features to help support your data resiliency and backup needs.

- If the core loses internet connectivity, Greengrass devices can continue to communicate over the local network.
- You can configure a Greengrass core device to write logs to the local file system and to CloudWatch Logs. If the core loses connectivity, local logging can continue, but CloudWatch logs are sent with a limited number of retries. After the retries are exhausted, the event is dropped. For more information, see the log manager component (p. 206).
- You can author Greengrass components that read stream manager (p. 519) streams and send the data to local storage destinations.

Infrastructure security in AWS IoT Greengrass

As a managed service, AWS IoT Greengrass is protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access AWS IoT Greengrass through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS), such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

In an AWS IoT Greengrass environment, devices use X.509 certificates and cryptographic keys to connect and authenticate to the AWS Cloud. For more information, see the section called “Device authentication and authorization” (p. 610).

- Evaluating Resources with Rules in the AWS Config Developer Guide – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.
Configuration and vulnerability analysis in AWS IoT Greengrass

IoT environments can consist of large numbers of devices that have diverse capabilities, are long-lived, and are geographically distributed. These characteristics make device setup complex and error-prone. And because devices are often constrained in computational power, memory, and storage capabilities, this limits the use of encryption and other forms of security on the devices themselves. Also, devices often use software with known vulnerabilities. These factors make IoT devices an attractive target for hackers and make it difficult to secure them on an ongoing basis.

AWS IoT Device Defender addresses these challenges by providing tools to identify security issues and deviations from best practices. You can use AWS IoT Device Defender to analyze, audit, and monitor connected devices to detect abnormal behavior, and mitigate security risks. AWS IoT Device Defender can audit devices to ensure they adhere to security best practices and detect abnormal behavior on devices. This makes it possible to enforce consistent security policies across your devices and respond quickly when devices are compromised. For more information, see the following topics:

- The Device Defender component (p. 170)
- AWS IoT Device Defender in the AWS IoT Core Developer Guide.

In AWS IoT Greengrass environments, you should be aware of the following considerations:

- It's your responsibility to secure your physical devices, the file system on your devices, and the local network.
- AWS IoT Greengrass doesn't enforce network isolation for user-defined Greengrass components, whether or not they run in a Greengrass container. Therefore, it's possible for Greengrass components to communicate with any other process running in the system or outside over network.

Code integrity in AWS IoT Greengrass V2

AWS IoT Greengrass deploys software components from the AWS Cloud to devices that run the AWS IoT Greengrass Core software. These software components include AWS-provided components (p. 132) and custom components (p. 324) that you upload to your AWS account. Every component is composed of a recipe. The recipe defines the component's metadata, and any number of artifacts, which are component binaries, such as compiled code and static resources. Component artifacts are stored in Amazon S3.

As you develop and deploy Greengrass components, you follow these basic steps that work with component artifacts in your AWS account and on your devices:

1. Create and upload artifacts to S3 buckets.
2. Create a component from a recipe and artifacts in the AWS IoT Greengrass service, which calculates a cryptographic hash of each artifact.
3. Deploy a component to Greengrass core devices, which download and verify the integrity of each artifact.

AWS is responsible for maintaining the integrity of artifacts after you upload artifacts to S3 buckets, including when you deploy components to Greengrass core devices. You are responsible for securing software artifacts before you upload the artifacts to S3 buckets. You are also responsible for securing access to resources in your AWS account, including the S3 buckets where you upload component artifacts.
Amazon S3 provides a feature called S3 Object Lock that you can use to protect against changes to component artifacts in S3 buckets your AWS account. You can use S3 Object Lock to prevent component artifacts from being deleted or overwritten. For more information, see Using S3 Object Lock in the Amazon Simple Storage Service User Guide.

When AWS publishes a public component, and when you upload a custom component, AWS IoT Greengrass calculates a cryptographic digest for each component artifact. AWS IoT Greengrass updates the component recipe to include each artifact's digest and the hash algorithm used to calculate that digest. This digest guarantees the integrity of the artifact, because if the artifact changes in the AWS Cloud or during download, its file digest won't match the digest that AWS IoT Greengrass stores in the component recipe. For more information, see Artifacts in the component recipe reference (p. 351).

When you deploy a component to a core device, the AWS IoT Greengrass Core software downloads the component recipe and each component artifact that the recipe defines. The AWS IoT Greengrass Core software calculates the digest of each downloaded artifact file and compares it with that artifact's digest in the recipe. If the digests don't match, the deployment fails, and the AWS IoT Greengrass Core software deletes the downloaded artifacts from the device's file system. For more information about how connections between core devices and AWS IoT Greengrass are secured, see Encryption in transit (p. 608).

You are responsible for securing component artifact files on your core devices' file systems. The AWS IoT Greengrass Core software saves artifacts to the packages folder in the Greengrass root folder. You can use AWS IoT Device Defender to analyze, audit, and monitor core devices. For more information, see Configuration and vulnerability analysis in AWS IoT Greengrass (p. 645).

AWS IoT Greengrass and interface VPC endpoints (AWS PrivateLink)

You can establish a private connection between your VPC and the AWS IoT Greengrass control plane by creating an interface VPC endpoint. You can use this endpoint to manage components, deployments, and core devices in the AWS IoT Greengrass service. Interface endpoints are powered by AWS PrivateLink, a technology that enables you to access AWS IoT Greengrass APIs privately without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to communicate with AWS IoT Greengrass APIs. Traffic between your VPC and AWS IoT Greengrass does not leave the Amazon network.

Note
Currently, you can't configure Greengrass core devices to operate completely within your VPC.

Each interface endpoint is represented by one or more Elastic Network Interfaces in your subnets.

For more information, see Interface VPC endpoints (AWS PrivateLink) in the Amazon VPC User Guide.

Topics
• Considerations for AWS IoT Greengrass VPC endpoints (p. 647)
• Create an interface VPC endpoint for AWS IoT Greengrass control plane operations (p. 647)
• Creating a VPC endpoint policy for AWS IoT Greengrass (p. 647)
Considerations for AWS IoT Greengrass VPC endpoints

Before you set up an interface VPC endpoint for AWS IoT Greengrass, review Interface endpoint properties and limitations in the Amazon VPC User Guide. Additionally, be aware of the following considerations:

- AWS IoT Greengrass supports making calls to all of its control plane API actions from your VPC. The control plane includes operations such as CreateDeployment and ListEffectiveDeployments. The control plane does not include operations such as ResolveComponentCandidates (p. 611) and Discover (p. 502), which are data plane operations.
- VPC endpoints for AWS IoT Greengrass are currently not supported in AWS China Regions.

Create an interface VPC endpoint for AWS IoT Greengrass control plane operations

You can create a VPC endpoint for the AWS IoT Greengrass control plane using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see Creating an interface endpoint in the Amazon VPC User Guide.

Create a VPC endpoint for AWS IoT Greengrass using the following service name:

- com.amazonaws.region.greengrass

If you enable private DNS for the endpoint, you can make API requests to AWS IoT Greengrass using its default DNS name for the Region, for example, greengrass.us-east-1.amazonaws.com. Private DNS is enabled by default.

For more information, see Accessing a service through an interface endpoint in the Amazon VPC User Guide.

Creating a VPC endpoint policy for AWS IoT Greengrass

You can attach an endpoint policy to your VPC endpoint that controls access to AWS IoT Greengrass control plane operations. The policy specifies the following information:

- The principal that can perform actions.
- The actions that the principal can perform.
- The resources that the principal can perform actions on.

For more information, see Controlling access to services with VPC endpoints in the Amazon VPC User Guide.

Example Example: VPC endpoint policy for AWS IoT Greengrass actions

The following is an example of an endpoint policy for AWS IoT Greengrass. When attached to an endpoint, this policy grants access to the listed AWS IoT Greengrass actions for all principals on all resources.

```json
{
}
```
Security best practices for AWS IoT Greengrass

This topic contains security best practices for AWS IoT Greengrass.

Grant minimum possible permissions

Follow the principle of least privilege by using the minimum set of permissions in IAM roles. Limit the use of the * wildcard for the Action and Resource properties in your IAM policies. Instead, declare a finite set of actions and resources when possible. For more information about least privilege and other policy best practices, see the section called “Policy best practices” (p. 625).

The least privilege best practice also applies to AWS IoT policies you attach to your Greengrass core.

Don't hardcode credentials in Greengrass components

Don't hardcode credentials in your user-defined Greengrass components. To better protect your credentials:

- To interact with AWS services, define permissions for specific actions and resources in the Greengrass core device service role (p. 627).
- Use the secret manager component (p. 285) to store your credentials. Or, if the function uses the AWS SDK, use credentials from the default credential provider chain.

Don't log sensitive information

You should prevent the logging of credentials and other personally identifiable information (PII). We recommend that you implement the following safeguards even though access to local logs on a core device requires root privileges and access to CloudWatch Logs requires IAM permissions.

- Don't use sensitive information in MQTT topic paths.
- Don't use sensitive information in device (thing) names, types, and attributes in the AWS IoT Core registry.
- Don't log sensitive information in your user-defined Greengrass components or Lambda functions.
- Don't use sensitive information in the names and IDs of Greengrass resources:
  - Core devices
  - Components
  - Deployments

"Statement": [
  {
    "Principal": "*",
    "Effect": "Allow",
    "Action": [
      "greengrass:CreateDeployment",
      "greengrass:ListEffectiveDeployments"
    ],
    "Resource": "*"
  }
]
Keep your device clock in sync

It's important to have an accurate time on your device. X.509 certificates have an expiry date and time. The clock on your device is used to verify that a server certificate is still valid. Device clocks can drift over time or batteries can get discharged.

For more information, see the Keep your device's clock in sync best practice in the AWS IoT Core Developer Guide.

See also

- Security best practices in AWS IoT Core in the AWS IoT Developer Guide
- Ten security golden rules for IoT solutions on the Internet of Things on AWS Official Blog
Logging and monitoring in AWS IoT Greengrass

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS IoT Greengrass and your AWS solutions. You should collect monitoring data from all parts of your AWS solution so that you can more easily debug a multi-point failure, if one occurs. Before you start monitoring AWS IoT Greengrass, you should create a monitoring plan that includes answers to the following questions:

• What are your monitoring goals?
• Which resources will you monitor?
• How often will you monitor these resources?
• Which monitoring tools will you use?
• Who will perform the monitoring tasks?
• Who should be notified when something goes wrong?

Monitoring tools

AWS provides tools that you can use to monitor AWS IoT Greengrass. You can configure some of these tools to do the monitoring for you. Some of the tools require manual intervention. We recommend that you automate monitoring tasks as much as possible.

You can use the following automated monitoring tools to monitor AWS IoT Greengrass and report issues:

• **Amazon CloudWatch Logs** – Monitor, store, and access your log files from AWS CloudTrail or other sources. For more information, see Monitoring log files in the Amazon CloudWatch User Guide.
• **AWS CloudTrail Log Monitoring** – Share log files between accounts, monitor CloudTrail log files in real time by sending them to CloudWatch Logs, write log processing applications in Java, and validate that your log files have not changed after delivery by CloudTrail. For more information, see Working with CloudTrail log files in the AWS CloudTrail User Guide.
• **Greengrass system health telemetry** – Subscribe to receive telemetry data sent from the Greengrass core. For more information, see the section called “Gather system health telemetry data” (p. 652).

Log AWS IoT Greengrass V2 API calls with AWS CloudTrail

AWS IoT Greengrass V2 is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in AWS IoT Greengrass Version 2. CloudTrail captures all API calls for AWS IoT Greengrass as events. The calls that are captured include calls from the AWS IoT Greengrass console and code calls to the AWS IoT Greengrass API operations.

If you create a trail, you can enable continuous delivery of CloudTrail events to an S3 bucket, including events for AWS IoT Greengrass. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in Event history. Using the information collected by CloudTrail, you can determine the request that was made to AWS IoT Greengrass, the IP address from which the request was made, who made the request, when it was made, and additional details.
For more information about CloudTrail, see the AWS CloudTrail User Guide.

AWS IoT Greengrass V2 information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in AWS IoT Greengrass, that activity is recorded in a CloudTrail event along with other AWS service events in Event history. You can view, search, and download recent events in your AWS account. For more information, see Viewing events with CloudTrail Event history.

For an ongoing record of events in your AWS account, including events for AWS IoT Greengrass, create a trail. A trail enables CloudTrail to deliver log files to an S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple regions

All AWS IoT Greengrass V2 actions are logged by CloudTrail and are documented in the AWS IoT Greengrass V2 API Reference. For example, calls to the CreateComponentVersion, CreateDeployment and CancelDeployment actions generate entries in the CloudTrail log files.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity element.

Understanding AWS IoT Greengrass V2 log file entries

A trail is a configuration that enables delivery of events as log files to an S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source. It includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the CreateDeployment action.

```json
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/Administrator",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "Administrator"
    }
}
```
Gather system health telemetry data from AWS IoT Greengrass core devices

System health telemetry data is diagnostic data that can help you monitor the performance of critical operations on your Greengrass core devices. The telemetry agent on Greengrass core devices collects local telemetry data and publishes it to Amazon EventBridge without requiring any customer interaction. Core devices publish telemetry data to EventBridge on a best effort basis. For example, core devices might fail to deliver telemetry data while offline.

**Note**
Amazon EventBridge is an event bus service that you can use to connect your applications with data from a variety of sources, such as Greengrass core devices. For more information, see What is Amazon EventBridge? in the Amazon EventBridge User Guide.

You can create projects and applications to retrieve, analyze, transform, and report telemetry data from your edge devices. Domain experts, such as process engineers, can use these applications to gain insights into fleet health.

```json
{
  "eventTime": "2021-01-06T02:38:05Z",
  "eventSource": "greengrass.amazonaws.com",
  "eventName": "CreateDeployment",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-cli/2.1.9 Python/3.7.9 Windows/10 exe/AMD64 prompt/off command/greengrassv2.create-deployment",
  "requestParameters": {
    "deploymentPolicies": {
      "failureHandlingPolicy": "DO NOTHING",
      "componentUpdatePolicy": {
        "timeoutInSeconds": 60,
        "action": "NOTIFY_COMPONENTS"
      },
      "configurationValidationPolicy": {
        "timeoutInSeconds": 60
      }
    },
    "deploymentName": "Deployment for MyGreengrassCoreGroup",
    "components": {
      "aws.greengrass.Cli": {
        "componentVersion": "2.0.3"
      }
    },
    "iotJobConfiguration": {},
    "targetArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/MyGreengrassCoreGroup"
  },
  "responseElements": {
    "iotJobArn": "arn:aws:iot:us-west-2:123456789012:job/fdfeba1d-ac6d-44ef-ab28-54f684e578d",
    "iotJobId": "fdfeba1d-ac6d-44ef-ab28-54f684e578d",
    "deploymentId": "4196dddc-0a21-4c54-a985-66a525f6946e"
  },
  "requestID": "311b9529-4aad-42ac-8408-c06c6fed79a9",
  "eventID": "c0f3aa2c-af22-48c1-8161-bad4a2ab1841",
  "readOnly": false,
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "eventCategory": "Management",
  "recipientAccountId": "123456789012"
}
```
To ensure that the Greengrass components function properly, AWS IoT Greengrass uses the data for development and quality improvement purposes. This feature also helps inform new and enhanced edge capabilities. AWS IoT Greengrass retains telemetry data for only up to seven days.

This feature is enabled by default for all Greengrass core devices. You automatically start to receive data as soon as you set up a Greengrass core device.

AWS IoT Greengrass telemetry uses the following default settings:

- The telemetry agent aggregates telemetry data every hour.
- The telemetry agent publishes a telemetry message every 24 hours.

The telemetry agent publishes data using the MQTT protocol with a quality of service (QoS) level of 0, which means that it doesn't confirm delivery or retry publishing attempts. Telemetry messages share an MQTT connection with other messages for subscriptions destined for AWS IoT Core.

Aside from your data link costs, the data transfer from the core to AWS IoT Core is without charge. This is because the agent publishes to an AWS reserved topic. However, depending on your use case, you might incur costs when you receive or process the data.

### Topics

- Telemetry metrics (p. 653)
- Configure telemetry settings (p. 654)
- Subscribe to telemetry data (p. 654)

## Telemetry metrics

The telemetry agent collects and publishes the following system metrics.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemMemUsage</td>
<td>The amount of memory currently in use by all applications on the Greengrass core device, including the operating system.</td>
<td>System</td>
</tr>
<tr>
<td>CpuUsage</td>
<td>The amount of CPU currently in use by all applications on the Greengrass core device, including the operating system.</td>
<td>System</td>
</tr>
<tr>
<td>TotalNumberOfFDs</td>
<td>The number of file descriptors stored by the operating system of the Greengrass core device. One file descriptor uniquely identifies one open file.</td>
<td>System</td>
</tr>
<tr>
<td>NumberOfComponentsRunning</td>
<td>The number of components that are running on the Greengrass core device.</td>
<td>AWS IoT Greengrass Core</td>
</tr>
<tr>
<td>NumberOfComponentsErrored</td>
<td>The number of components that are in error state on the Greengrass core device.</td>
<td>AWS IoT Greengrass Core</td>
</tr>
</tbody>
</table>
Configure telemetry settings

You can enable or disable the telemetry feature for each Greengrass core device. You can also configure the intervals over which the core device aggregates and publishes data. To configure telemetry, customize the telemetry configuration parameter (p. 142) when you deploy the Greengrass nucleus component (p. 136).

Subscribe to telemetry data

You can create rules in Amazon EventBridge that define how to process telemetry data published from the Greengrass core device. When EventBridge receives the data, it invokes the target actions defined in your rules. For example, you can create event rules that send notifications, store event information, take corrective action, or invoke other events.

Telemetry events

Telemetry events use the following format.

```
{
    "version": "0",
    "id": "a09d303e-2f6e-3d3c-a693-8e33f4fe3955",
    "detail-type": "Greengrass Telemetry Data",
    "source": "aws.greengrass",
    "account": "123456789012",
    "time": "2020-11-30T20:45:53Z",
    "region": "us-east-1",
    "resources": []
}
```
"detail": {
"ThingName": "MyGreengrassCore",
"Schema": "2020-07-30",
"ADP": [
{ "TS": 1602186483234, 
"NS": "SystemMetrics", 
"M": [
{ "N": "TotalNumberOfFDs", 
"Sum": 6447.0, 
"U": "Count"
},
{ "N": "CpuUsage", 
"Sum": 15.458333333333332, 
"U": "Percent"
},
{ "N": "SystemMemUsage", 
"Sum": 10201.0, 
"U": "Megabytes"
]
]
},
{ "TS": 1602186483234, 
"NS": "GreengrassComponents", 
"M": [
{ "N": "NumberOfComponentsStopping", 
"Sum": 0.0, 
"U": "Count"
},
{ "N": "NumberOfComponentsStarting", 
"Sum": 0.0, 
"U": "Count"
},
{ "N": "NumberOfComponentsBroken", 
"Sum": 0.0, 
"U": "Count"
},
{ "N": "NumberOfComponentsFinished", 
"Sum": 1.0, 
"U": "Count"
},
{ "N": "NumberOfComponentsInstalled", 
"Sum": 0.0, 
"U": "Count"
},
{ "N": "NumberOfComponentsRunning", 
"Sum": 7.0, 
"U": "Count"
},
{ "N": "NumberOfComponentsNew", 
"Sum": 0.0, 
"U": "Count"
},
{ "N": "NumberOfComponentsErrored", 
"Sum": 655
}]}
The ADP array contains a list of aggregated data points that have the following properties:

**TS**
- The timestamp of when the data was aggregated.

**NS**
- The metric namespace.

**M**
- The list of metrics. A metric contains the following properties:
  **N**
  - The name of the metric (p. 653).
  **Sum**
  - The aggregated metric value. The telemetry agent adds new values to the previous total, so the sum is a value that increases constantly. You can use the timestamp to find the value of a specific aggregation. For example, to find the latest aggregated value, subtract the previous timestamped value from the latest timestamped value.
  **U**
  - The unit of the metric value.

### Prerequisites to create EventBridge rules

Before you create an EventBridge rule for AWS IoT Greengrass, you should do the following:

- Familiarize yourself with events, rules, and targets in EventBridge.
- Create and configure the targets invoked by your EventBridge rules. Rules can invoke many types of targets, such as Amazon Kinesis streams, AWS Lambda functions, Amazon SNS topics, and Amazon SQS queues.

Your EventBridge rule, and the associated targets must be in the AWS Region where you created your Greengrass resources. For more information, see Service endpoints and quotas in the AWS General Reference.

For more information, see What is Amazon EventBridge? and Getting started with Amazon EventBridge in the Amazon EventBridge User Guide.

### Create an event rule to get telemetry data (console)

Use the following steps to use the AWS Management Console to create an EventBridge rule that receives telemetry data published by the Greengrass core device. This allows web servers, email addresses, and
Subscribe to telemetry data

other topic subscribers to respond to the event. For more information, see Creating a EventBridge rule that triggers on an event from an AWS resource in the Amazon EventBridge User Guide.

1. Open the Amazon EventBridge console, and choose Create rule.
2. Under Name and description, enter a name and description for the rule.
3. Under Define pattern, configure the rule pattern.
   a. Choose Event pattern.
   b. Choose Pre-defined pattern by service.
   c. For Service provider, choose AWS.
   d. For Service name, choose Greengrass.
   e. For Event type, select Greengrass Telemetry Data.
4. Under Select event bus, keep the default event bus options.
5. Under Select targets, configure your target. The following example uses an Amazon SQS queue, but you can configure other target types.
   a. For Target, choose SQS queue.
   b. For Queue*, choose your target queue.
6. Under Tags - optional, define tags for the rule or leave the fields empty.
7. Choose Create.

Create an event rule to get telemetry data (CLI)

Use the following steps to use the AWS CLI to create an EventBridge rule that receives telemetry data published by Greengrass core devices. This allows web servers, email addresses, and other topic subscribers to respond to the event.

1. Create the rule.
   - Replace thing-name with the thing name of the core device.

   ```bash
   aws events put-rule
   --name MyGreengrassTelemetryEventRule
   --event-pattern "{"source": ["aws.greengrass"], "detail": {"ThingName": ["thing-name"]}}"
   ```

   Properties that are omitted from the pattern are ignored.

2. Add the topic as a rule target. The following example uses Amazon SQS but you can configure other target types.
   - Replace queue-arn with the ARN of your Amazon SQS queue.

   ```bash
   aws events put-targets
   --rule MyGreengrassTelemetryEventRule
   --targets "Id"="1","Arn"="queue-arn"
   ```

   **Note**
   To allow Amazon EventBridge to invoke your target queue, you must add a resource-based policy to your topic. For more information, see Amazon SQS permissions in the Amazon EventBridge User Guide.
Check Greengrass core device status

Greengrass core devices report the status of their software components to AWS IoT Greengrass. You can check the health summary of each device, and you can check the status of each component on each device.

Core devices have the following health statuses:

- **HEALTHY** – The AWS IoT Greengrass Core software and all components run without issue on the core device.
- **UNHEALTHY** – The AWS IoT Greengrass Core software or a component is in an error state on the core device.

**Note**

AWS IoT Greengrass relies on individual devices to send status updates to the AWS Cloud. If the AWS IoT Greengrass Core software isn't running on the device, or if device isn't connected to the AWS Cloud, then the reported status of that device might not reflect its current status. The status timestamp indicates when the device status was last updated.

**Topics**

- Check health of a core device (p. 658)
- Check health of a core device group (p. 658)
- Check core device component status (p. 659)

Check health of a core device

You can check the status of individual core devices.

**To check the status of a core device (AWS CLI)**

- Run the following command to retrieve the status of a device. Replace `coreDeviceName` with the name of the core device to query.

```
aws greengrassv2 get-core-device \\
--core-device-thing-name coreDeviceName
```

The response contains information about the core device, including its status.

Check health of a core device group

You can check the status of a group of core devices (a thing group).

**To check the status of a group of devices (AWS CLI)**

- Run the following command to retrieve the status of multiple core devices. Replace the ARN in the command with the ARN of the thing group to query.

```
aws greengrassv2 list-core-devices \\
--thing-group ARN
```
Check core device component status

You can check the status, such as lifecycle state, of the software components on a core device. For more information about component lifecycle states, see Develop AWS IoT Greengrass components (p. 322).

To check the status of components on a core device (AWS CLI)

- Run the following command to retrieve the status of the components on a core device. Replace `coreDeviceName` with the name of the core device to query.

```bash
aws greengrassv2 list-installed-components \
  --core-device-thing-name coreDeviceName
```

The response contains the list of components that run on the core device. Each entry in the list contains the lifecycle state of the component.
Using AWS IoT Device Tester for AWS IoT Greengrass V2

AWS IoT Device Tester (IDT) is a downloadable testing framework that lets you validate IoT devices. You can use IDT for AWS IoT Greengrass to run the AWS IoT Greengrass qualification suite, and create and run custom test suites for your devices.

IDT for AWS IoT Greengrass runs on your host computer (Windows, macOS, or Linux) connected to the device to be tested. It runs tests and aggregates results. It also provides a command line interface to manage the testing process.

AWS IoT Greengrass qualification suite

Use AWS IoT Device Tester for AWS IoT Greengrass V2 to verify that the AWS IoT Greengrass Core software runs on your hardware and can communicate with the AWS Cloud. It also performs end-to-end tests with AWS IoT Core. For example, it verifies that your device can deploy components and upgrade them.

If you want to add your hardware to the AWS Partner Device Catalog, run the AWS IoT Greengrass qualification suite to generate test reports that you can submit to AWS IoT. For more information, see AWS Device Qualification Program.

IDT for AWS IoT Greengrass V2 organizes tests using the concepts of test suites and test groups.

- A test suite is the set of test groups used to verify that a device works with particular versions of AWS IoT Greengrass.
- A test group is the set of individual tests related to a particular feature, such as component deployments.

For more information, see Use IDT to run the AWS IoT Greengrass qualification suite (p. 663).
Custom test suites

Starting in IDT v4.0.1, IDT for AWS IoT Greengrass V2 combines a standardized configuration setup and result format with a test suite environment that enables you to develop custom test suites for your devices and device software. You can add custom tests for your own internal validation or provide them to your customers for device verification.

How a test writer configures a custom test suite determines the settings configurations that are required to run custom test suites. For more information, see Use IDT to develop and run your own test suites (p. 687).

Supported versions of AWS IoT Device Tester for AWS IoT Greengrass V2

This topic lists supported versions of IDT for AWS IoT Greengrass V2. As a best practice, we recommend that you use the latest version of IDT for AWS IoT Greengrass V2 that supports your target version of AWS IoT Greengrass V2. New releases of AWS IoT Greengrass might require you to download a new version of IDT for AWS IoT Greengrass V2. You receive a notification when you start a test run if IDT for AWS IoT Greengrass V2 is not compatible with the version of AWS IoT Greengrass you are using.

By downloading the software, you agree to the AWS IoT Device Tester License Agreement.

**Note**
IDT does not support being run by multiple users from a shared location, such as an NFS directory or a Windows network shared folder. We recommend that you extract the IDT package to a local drive and run the IDT binary on your local workstation.

**Latest IDT version for AWS IoT Greengrass V2**

You can use this version of IDT for AWS IoT Greengrass V2 with the AWS IoT Greengrass version listed here.

**IDT v4.2.0 for AWS IoT Greengrass**

Supported AWS IoT Greengrass versions:

- Greengrass nucleus (p. 136) v2.4.0, v2.3.0, v2.2.0, and v2.1.0

IDT software downloads:

- IDT v4.2.0 with test suite GGV2Q_2.0.1 for Linux
- IDT v4.2.0 with test suite GGV2Q_2.0.1 for macOS
- IDT v4.2.0 with test suite GGV2Q_2.0.1 for Windows

Release notes:

- Includes support for qualification of the following features on devices running AWS IoT Greengrass Core software v2.2.0 and later versions:
  - Docker—Validates that devices can download a Docker container image from Amazon Elastic Container Registry (Amazon ECR).
  - Machine learning—Validates that devices can perform machine learning (ML) inference using the Deep Learning Runtime or TensorFlow Lite ML frameworks.
  - Stream manager—Validates that devices can download, install, and run the AWS IoT Greengrass stream manager.
  - Enables you to validate and qualify devices running AWS IoT Greengrass Core software v2.4.0, v2.3.0, v2.2.0, and v2.1.0.
• Groups the test logs for each test case in a separate `<test-case-id>` folder within the `<device-tester-extract-location>/results/<execution-id>/logs/<test-group-id>` directory.

• Additional minor bug fixes.

Test suite version:
GGV2Q_2.0.1
• Released 2021.08.31

Unsupported versions of AWS IoT Device Tester for AWS IoT Greengrass V2

This topic lists unsupported versions of IDT for AWS IoT Greengrass V2. Unsupported versions do not receive bug fixes or updates. For more information, see the section called “Support policy for AWS IoT Device Tester for AWS IoT Greengrass” (p. 749).

IDT v4.1.0 for AWS IoT Greengrass

Release notes:
• Enables you to validate and qualify devices running AWS IoT Greengrass Core software v2.3.0, v2.2.0, v2.1.0, and v2.0.5.
• Improves the `userdata.json` configuration by removing the requirement to specify the `GreengrassNucleusVersion` and `GreengrassCLIVersion` properties.
• Includes support for Lambda and MQTT feature qualification for AWS IoT Greengrass Core software v2.1.0 and later versions. You can now use IDT for AWS IoT Greengrass V2 to validate that your core device can run Lambda functions and that the device can publish and subscribe to AWS IoT Core MQTT topics.
• Improves logging capabilities.
• Additional minor bug fixes.

Test suite version:
GGV2Q_1.1.1
• Released 2021.06.18

IDT v4.0.2 for AWS IoT Greengrass

Release notes:
• Enables you to validate and qualify devices running AWS IoT Greengrass Core software v2.1.0.
• Adds support for Lambda and MQTT feature qualification for AWS IoT Greengrass Core software v2.1.0 and later versions. You can now use IDT for AWS IoT Greengrass V2 to validate that your core device can run Lambda functions and that the device can publish and subscribe to AWS IoT Core MQTT topics.
• Improves logging capabilities.
• Additional minor bug fixes.

Test suite version:
GGV2Q_1.1.1
• Released 2021.05.05

IDT v4.0.1 for AWS IoT Greengrass

Release notes:
• Enables you to validate and qualify devices running AWS IoT Greengrass Version 2 software.
Use IDT to run the AWS IoT Greengrass qualification suite

You can use AWS IoT Device Tester for AWS IoT Greengrass V2 to verify that the AWS IoT Greengrass Core software runs on your hardware and can communicate with the AWS Cloud. It also performs end-to-end tests with AWS IoT Core. For example, it verifies that your device can deploy components and upgrade them.

In addition to testing devices, IDT for AWS IoT Greengrass V2 creates resources (for example, AWS IoT things, groups, and so on) in your AWS account to facilitate the qualification process.

To create these resources, IDT for AWS IoT Greengrass V2 uses the AWS credentials configured in the config.json file to make API calls on your behalf. These resources are provisioned at various times during a test.

When you use IDT for AWS IoT Greengrass V2 to run the AWS IoT Greengrass qualification suite, it performs the following steps:

1. Loads and validates your device and credentials configuration.
2. Performs selected tests with the required local and cloud resources.
3. Cleans up local and cloud resources.
4. Generates tests reports that indicate if your board passed the tests required for qualification.

Test suite versions

IDT for AWS IoT Greengrass V2 organizes tests into test suites and test groups.

- A test suite is the set of test groups used to verify that a device works with particular versions of AWS IoT Greengrass.
- A test group is the set of individual tests related to a particular feature, such as component deployments.

Test suites are versioned using a major.minor.patch format, for example GGV2Q_1.0.0. When you download IDT, the package includes the latest Greengrass qualification suite version.

Important
IDT supports the four latest major.minor versions of the Greengrass qualification suite versions to generate qualification reports that you can submit to AWS Partner Network to
Test group descriptions

IDT v4.2.0 and later

**Required Test Groups for Core Qualification**

These test groups are required to qualify your AWS IoT Greengrass V2 device for the AWS Partner Device Catalog.

**Core Dependencies**

Validates that the device meets all software and hardware requirements for the AWS IoT Greengrass Core software. This test group includes the following test case:

**Java Version**

Checks that the required Java version is installed on the device under test. AWS IoT Greengrass requires Java 8 or later.

**PreTest Validation**

Checks that the device can run the following Linux commands:

- chmod
- cp
- echo
- grep
- kill
- ln
- mkinfo
- ps
- rm
- sh
- uname

**Version Checker**

Checks that the version of AWS IoT Greengrass provided is compatible with the AWS IoT Device Tester version you are using.

**Component**

Validates that the device can deploy components and upgrade them. This test group includes the following tests:

- **Cloud Component**
  
  Validates device capability for cloud components.

- **Local Component**
  
  Validates device capability for local components.

**Lambda**

Validates that the device can deploy Lambda function components that use the Java runtime, and that the Lambda functions can use AWS IoT Core MQTT topics as event sources for work messages.

**MQTT**

Validates that the device can subscribe and publish to AWS IoT Core MQTT topics.

**Optional Test Groups**

These test groups are optional. If you choose to qualify for optional tests, your device is listed with additional capabilities in the AWS Partner Device Catalog.
Docker dependencies

Validates that the device meets all required technical dependencies to use the AWS-provided Docker application manager (aws.greengrass.DockerApplicationManager) component.

Docker Application Manager Qualification

Validates that the device can download a Docker container image from Amazon Elastic Container Registry (Amazon ECR).

Machine Learning Dependencies

Validates that the device meets all of the required technical dependencies to use the AWS-provided machine learning (ML) components.

Machine Learning Inference Tests

Validates that the device can perform ML inference using the Deep Learning Runtime and TensorFlow Lite ML frameworks.

Stream Manager Dependencies

Validates that the device can download, install, and run the AWS IoT Greengrass stream manager (p. 519).

IDT v4.0.2 and v4.1.x

**Required Test Groups for Core Qualification**

These test groups are required to qualify your AWS IoT Greengrass V2 device for the AWS Partner Device Catalog.

Core Dependencies

Validates that your device meets all software and hardware requirements for the AWS IoT Greengrass Core software. This test group includes the following test case:

Java Version

Checks that the required Java version is installed on your device. AWS IoT Greengrass requires Java 8 or later.

PreTest Validation

Checks that your device can run the following Linux commands:

```
chmod, cp, echo, grep, kill, ln, mkinfo, ps, rm, sh, uname
```

Version Checker

Checks that the version of AWS IoT Greengrass provided is compatible with the AWS IoT Device Tester version you are using.

Component

Validates that your device can deploy components and upgrade them. This test group includes the following tests:

Cloud Component

Validates device capability for cloud components.

Local Component

Validates device capability for local components.
Prerequisites for running the AWS IoT Greengrass qualification suite

This section describes the prerequisites for using AWS IoT Device Tester (IDT) for AWS IoT Greengrass.

Download the latest version of AWS IoT Device Tester for AWS IoT Greengrass

Download the latest version (p. 661) of IDT and extract the software into a location (device-tester-extract-location) on your file system where you have read/write permissions.

Note
IDT does not support being run by multiple users from a shared location, such as an NFS directory or a Windows network shared folder. We recommend that you extract the IDT package to a local drive and run the IDT binary on your local workstation.

Windows has a path length limitation of 260 characters. If you are using Windows, extract IDT to a root directory like C:\ or D:\ to keep your paths under the 260 character limit.

Download the AWS IoT Greengrass software

IDT for AWS IoT Greengrass V2 tests your device for compatibility with a specific version of AWS IoT Greengrass. Run the following command to download the AWS IoT Greengrass Core software to a file named aws.greengrass.nucleus.zip. Replace version with a supported nucleus component version (p. 661) for your IDT version.

```
curl -s https://d2s8p8vqu9w66.cloudfront.net/releases/greengrass-version.zip > aws.greengrass.nucleus.zip
```

Place the downloaded aws.greengrass.nucleus.zip file in the device-tester-extract-location/products/ folder.

Note
Do not place multiple files in this directory for the same operating system and architecture.

Create and configure an AWS account

Before you can use AWS IoT Device Tester for AWS IoT Greengrass V2, you must perform the following steps:

1. Set up an AWS account. (p. 667) If you already have an AWS account, skip to step 2.
2. Configure permissions for IDT. (p. 668)

These account permissions allow IDT to access AWS services and create AWS resources, such as AWS IoT things and AWS IoT Greengrass components, on your behalf.
Prerequisites

To create these resources, IDT for AWS IoT Greengrass V2 uses the AWS credentials configured in the config.json file to make API calls on your behalf. These resources are provisioned at various times during a test.

**Note**
Although most tests qualify for AWS Free Tier, you must provide a credit card when you sign up for an AWS account. For more information, see Why do I need a payment method if my account is covered by the Free Tier?

**Step 1: Set up an AWS account**

In this step, create and configure an AWS account. If you already have an AWS account, skip to the section called “Step 2: Configure permissions for IDT” (p. 668).

If you do not have an AWS account, complete the following steps to create one.

**To sign up for an AWS account**

2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

**To create an administrator user for yourself and add the user to an administrators group (console)**

1. Sign in to the IAM console as the account owner by choosing Root user and entering your AWS account email address. On the next page, enter your password.

   **Note**
   We strongly recommend that you adhere to the best practice of using the Administrator IAM user that follows and securely lock away the root user credentials. Sign in as the root user only to perform a few account and service management tasks.

2. In the navigation pane, choose Users and then choose Add user.
3. For User name, enter Administrator.
4. Select the check box next to AWS Management Console access. Then select Custom password, and then enter your new password in the text box.
5. (Optional) By default, AWS requires the new user to create a new password when first signing in. You can clear the check box next to User must create a new password at next sign-in to allow the new user to reset their password after they sign in.
6. Choose Next: Permissions.
7. Under Set permissions, choose Add user to group.
8. Choose Create group.
9. In the Create group dialog box, for Group name enter Administrators.
10. Choose Filter policies, and then select AWS managed - job function to filter the table contents.
11. In the policy list, select the check box for AdministratorAccess. Then choose Create group.

   **Note**
   You must activate IAM user and role access to Billing before you can use the AdministratorAccess permissions to access the AWS Billing and Cost Management console. To do this, follow the instructions in step 1 of the tutorial about delegating access to the billing console.
12. Back in the list of groups, select the check box for your new group. Choose Refresh if necessary to see the group in the list.
13. Choose Next: Tags.
14. (Optional) Add metadata to the user by attaching tags as key-value pairs. For more information about using tags in IAM, see Tagging IAM entities in the IAM User Guide.
15. Choose Next: Review to see the list of group memberships to be added to the new user. When you are ready to proceed, choose Create user.

You can use this same process to create more groups and users and to give your users access to your AWS account resources. To learn about using policies that restrict user permissions to specific AWS resources, see Access management and Example policies.

Step 2: Configure permissions for IDT

In this step, configure the permissions that IDT for AWS IoT Greengrass V2 uses to run tests and collect IDT usage data. You can use the AWS Management Console (p. 668) or AWS Command Line Interface (AWS CLI) (p. 671) to create an IAM policy and a test user for IDT, and then attach policies to the user. If you already created a test user for IDT, skip to Configure your device to run IDT tests (p. 675).

To configure permissions for IDT (console)

1. Sign in to the IAM console.
2. Create a customer managed policy that grants permissions to create roles with specific permissions.
   a. In the navigation pane, choose Policies, and then choose Create policy.
   b. On the JSON tab, replace the placeholder content with the following policy.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor1",
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": "arn:aws:iam::*:role/idt-*",
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": [
            "iot.amazonaws.com",
            "lambda.amazonaws.com",
            "greengrass.amazonaws.com"
          ]
        }
      }
    },
    {
      "Sid": "VisualEditor2",
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:PublishVersion",
        "iot:DeleteCertificate",
        "lambda:DeleteFunction",
        "lambda:GetFunction",
        "execute-api:Invoke",
        "iot:UpdateCertificate"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:098862408343:9xpvmvs5h4/prod/POST/metrics",
        "arn:aws:lambda:*:*:function:idt-*",
        "arn:aws:iot:*:*:cert/*"
      ]
    }
  ]
}
```
**Prerequisites**

```
"iot:ListAttachedPolicies",
"iot:CreateThing",
"iot:DescribeRoleAlias",
"iot:DescribeEndpoint",
"iot:DeleteThing",
"iot:DeleteThingGroup",
"iot:DeleteRoleAlias",
"iot:DeletePolicy",
"iot:DetachPolicy",
"iot:DetachThingPrincipal",
"iot:RemoveThingFromThingGroup",
"iot:UpdateCertificate",
"iot:DescribeThing",
"iot:DeleteCertificate",
"iot:CreatePolicy",
"iot:GetPolicy",
"iot:GetThingShadow",
"iot:CreateKeysAndCertificate",
"iot:ListThings",
"iot:UpdateThingShadow",
"iot:Publish",
"iot:CreateCertificateFromCsr",
"iot:ListTagsForResource",
"iot:TagResource",
"iot:ListThingPrincipals",
"iot-device-tester:SendMetrics",
"iot-device-tester:SupportedVersion",
"iot-device-tester:LatestIdt",
"iot-device-tester:CheckVersion",
"iot-device-tester:DownloadTestSuite"
]
"Resource": "*"
}
{
"Sid": "VisualEditor7",
"Effect": "Allow",
"Action": [  
  "iot:DetachThingPrincipal",
  "iot:AttachThingPrincipal"
],
"Resource": [  
  "arn:aws:iot::*:*:thing/idt-*",
  "arn:aws:iot::*:cert/*"
]
},
{
"Sid": "VisualEditor8",
"Effect": "Allow",
"Action": [  
  "s3:GetObject",
  "s3:PutObject",
  "s3:AbortMultipartUpload",
  "s3:ListMultipartUploadParts",
  "s3:DeleteObjectVersion",
  "s3:PutObjectTagging",
  "s3:ListBucketVersions",
  "s3:CreateBucket",
  "s3:DeleteObject",
  "s3:DeleteBucket",
  "s3:ListBucket",
  "s3:PutBucketTagging"
],
"Resource": "arn:aws:s3:::idt*"
}
{
"Sid": "VisualEditor9",
"Effect": "Allow",
"Action": [  
  "iot:CreateCertificateFromCsr",
  "iot:ListTagsForResource",
  "iot:TagResource",
  "iot:ListThingPrincipals",
  "iot-device-tester:SendMetrics",
  "iot-device-tester:SupportedVersion",
  "iot-device-tester:LatestIdt",
  "iot-device-tester:CheckVersion",
  "iot-device-tester:DownloadTestSuite"
],
"Resource": "*"
}
```
Prerequisites

"Effect": "Allow",
"Action": [
  "iot:CreateThingGroup",
  "iot:AddThingToThingGroup",
  "iot:DescribeThingGroup"
],
"Resource": "arn:aws:iot:*:*:thinggroup/*"
},

{ "Sid": "VisualEditor10",
  "Effect": "Allow",
  "Action": [
    "iot:AddThingToThingGroup"
  ],
  "Resource": "arn:aws:iot:*:*:thing/idt-*"
},

{ "Sid": "VisualEditor11",
  "Effect": "Allow",
  "Action": [
    "iot:DescribeRoleAlias"
  ],
  "Resource": "arn:aws:iot:*:*:rolealias/*"
}

```json
```


c. Choose Review policy.

d. For Name, enter IDTGreengrassIAMPPermissions. Under Summary, review the permissions granted by your policy.

e. Choose Create policy.

3. Create an IAM user and attach the permissions required by IDT for AWS IoT Greengrass.

a. Create an IAM user. Follow steps 1 through 5 in Creating IAM users (console) in the IAM User Guide.

b. Attach the permissions to your IAM user:

i. On the Set permissions page, choose Attach existing policies to user directly.

ii. Search for the IDTGreengrassIAMPPermissions policy that you created in the previous step. Select the check box.

c. Choose Next: Tags.

d. Choose Next: Review to view a summary of your choices.

e. Choose Create user.

f. To view the user's access keys (access key IDs and secret access keys), choose Show next to the password and access key. To save the access keys, choose Download.csv and save the file to a secure location. You use this information later to configure your AWS credentials file.

4. Next step: Configure your physical device (p. 675).

To configure permissions for IDT (AWS CLI)

1. On your computer, install and configure the AWS CLI if it's not already installed. Follow the steps in Installing the AWS CLI in the AWS Command Line Interface User Guide.

   Note
   The AWS CLI is an open source tool that you can use to interact with AWS services from your command-line shell.

2. Create a customer managed policy that grants permissions to manage IDT and AWS IoT Greengrass roles.

```json
```
a. Open a text editor and save the following policy contents in a JSON file.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::*:role/idt-*",
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": [
                        "iot.amazonaws.com",
                        "lambda.amazonaws.com",
                        "greengrass.amazonaws.com"
                    ]
                }
            }
        },
        {
            "Sid": "VisualEditor2",
            "Effect": "Allow",
            "Action": [
                "lambda:CreateFunction",
                "lambda:PublishVersion",
                "iot:DeleteCertificate",
                "lambda:DeleteFunction",
                "lambda:GetFunction",
                "execute-api:Invoke",
                "iot:UpdateCertificate"
            ],
            "Resource": [
                "arn:aws:execute-api:us-east-1:098862408343:9xpmnvs5h4/prod/POST/metrics",
                "arn:aws:lambda:*:*:function:idt-*",
                "arn:aws:iot:*:*:cert/*"
            ]
        },
        {
            "Sid": "VisualEditor3",
            "Effect": "Allow",
            "Action": [
                "iot:CreateThing",
                "iot:DeleteThing"
            ],
            "Resource": [
                "arn:aws:iot:*:*:thing/idt-*",
                "arn:aws:iot:*:*:cert/*"
            ]
        },
        {
            "Sid": "VisualEditor4",
            "Effect": "Allow",
            "Action": [
                "iot:AttachPolicy",
                "iot:DetachPolicy",
                "iot:DeletePolicy"
            ],
            "Resource": [
                "arn:aws:iot:*:*:policy/idt-*",
                "arn:aws:iot:*:*:cert/*"
            ]
        }
    ]
}
```
"Sid": "VisualEditor5",
"Effect": "Allow",
"Action": [
  "iot:CreateJob",
  "iot:DescribeJob",
  "iot:DescribeJobExecution",
  "iot:ListJobs*",
  "iot:DeleteJob"
],
"Resource": [
  "arn:aws:iot:*:*:thing/idt-*",
  "arn:aws:iot:*:*:job/*",
  "arn:aws:iot:*:*:thinggroup/*"
],
{
"Sid": "VisualEditor6",
"Effect": "Allow",
"Action": [
  "iot:DescribeEndpoint",
  "greengrass:*",
  "iam:ListAttachedRolePolicies",
  "iam:CreateRole",
  "iam:CreatePolicy",
  "iam:ListPolicies",
  "iam:ListRolePolicies",
  "iam:AttachRolePolicy",
  "iam:ListEntitiesForPolicy",
  "iam:GetRole",
  "iam:PassRole",
  "iam:DeleteRole",
  "iam:DeletePolicy",
  "iam:DetachRolePolicy",
  "iam:TagRole",
  "iam:TagPolicy",
  "iot:CreateRoleAlias",
  "iot:DescribeThingGroup",
  "iot:CreateThingGroup",
  "iot:AddThingToThingGroup",
  "iot:AttachPolicy",
  "iot:ListAttachedPolicies",
  "iot:CreateThing",
  "iot:DescribeRoleAlias",
  "iot:DescribeEndpoint",
  "iot:DeleteThing",
  "iot:DeleteThingGroup",
  "iot:DeleteRoleAlias",
  "iot:DeletePolicy",
  "iot:DetachPolicy",
  "iot:DetachThingPrincipal",
  "iot:RemoveThingFromThingGroup",
  "iot:UpdateCertificate",
  "iot:DescribeThing",
  "iot:DeleteCertificate",
  "iot:CreatePolicy",
  "iot:GetPolicy",
  "iot:GetThingShadow",
  "iot:CreateKeysAndCertificate",
  "iot:ListThings",
  "iot:UpdateThingShadow",
  "iot:Publish",
  "iot:CreateCertificateFromCsr",
  "iot:ListTagsForResource",
  "iot:TagResource",
  "iot:ListThingPrincipals",
  "iot-device-tester:SendMetrics",
]
AWS IoT Greengrass Developer Guide, Version 2
Prerequisites

]

"iot-device-tester:SupportedVersion",
"iot-device-tester:LatestIdt",
"iot-device-tester:CheckVersion",
"iot-device-tester:DownloadTestSuite"
],
"Resource": "*"
},
{
"Sid": "VisualEditor7",
"Effect": "Allow",
"Action": [
"iot:DetachThingPrincipal",
"iot:AttachThingPrincipal"
],
"Resource": [
"arn:aws:iot:*:*:thing/idt-*",
"arn:aws:iot:*:*:cert/*"
]
},
{
"Sid": "VisualEditor8",
"Effect": "Allow",
"Action": [
"s3:GetObject",
"s3:PutObject",
"s3:AbortMultipartUpload",
"s3:ListMultipartUploadParts",
"s3:DeleteObjectVersion",
"s3:PutObjectTagging",
"s3:ListBucketVersions",
"s3:CreateBucket",
"s3:DeleteObject",
"s3:DeleteBucket",
"s3:ListBucket",
"s3:PutBucketTagging"
],
"Resource": "arn:aws:s3:::idt*"
},
{
"Sid": "VisualEditor9",
"Effect": "Allow",
"Action": [
"iot:CreateThingGroup",
"iot:AddThingToThingGroup",
"iot:DescribeThingGroup"
],
"Resource": "arn:aws:iot:*:*:thinggroup/*"
},
{
"Sid": "VisualEditor10",
"Effect": "Allow",
"Action": [
"iot:AddThingToThingGroup"
],
"Resource": "arn:aws:iot:*:*:thing/idt-*"
},
{
"Sid": "VisualEditor11",
"Effect": "Allow",
"Action": [
"iot:DescribeRoleAlias"
],
"Resource": "arn:aws:iot:*:*:rolealias/*"
}

674


Configure your device to run IDT tests

To enable IDT to run tests for device qualification, you must configure your host computer to access your device, and configure user permissions on your device.

Install Java on the host computer

Starting with IDT v4.2.0, the optional qualification tests for AWS IoT Greengrass require Java to run. You can use Java version 8 or greater. We recommend Amazon Corretto 11 or OpenJDK 11.

Configure your host computer to access your device under test

IDT runs on your host computer and must be able to use SSH to connect to your device. There are two options to allow IDT to gain SSH access to your devices under test:

1. Follow the instructions here to create an SSH key pair and authorize your key to sign in to your device under test without specifying a password.
2. Provide a user name and password for each device in the device.json file. For more information, see Configure device.json (p. 679).

You can use any SSL implementation to create an SSH key. The following instructions show you how to use SSH-KEYGEN or PuTTYgen (for Windows). If you are using another SSL implementation, refer to the documentation for that implementation.

IDT uses SSH keys to authenticate with your device under test.
To create an SSH key with SSH-KEYGEN

1. Create an SSH key.

You can use the Open SSH ssh-keygen command to create an SSH key pair. If you already have an SSH key pair on your host computer, it is a best practice to create a SSH key pair specifically for IDT. This way, after you have completed testing, your host computer can no longer connect to your device without entering a password. It also allows you to restrict access to the remote device to only those who need it.

Note
Windows does not have an installed SSH client. For information about installing an SSH client on Windows, see Download SSH Client Software.

The ssh-keygen command prompts you for a name and path to store the key pair. By default, the key pair files are named id_rsa (private key) and id_rsa.pub (public key). On macOS and Linux, the default location of these files is ~/.ssh/. On Windows, the default location is C:\Users <user-name>\.ssh.

When prompted, enter a key phrase to protect your SSH key. For more information, see Generate a New SSH key.

2. Add authorized SSH keys to your device under test.

IDT must use your SSH private key to sign in to your device under test. To authorize your SSH private key to sign in to your device under test, use the ssh-copy-id command from your host computer. This command adds your public key into the ~/.ssh/authorized_keys file on your device under test. For example:

$ ssh-copy-id <remote-ssh-user>@<remote-device-ip>

Where remote-ssh-user is the user name used to sign in to your device under test and remote-device-ip is the IP address of the device under test to run tests against. For example:

ssh-copy-id pi@192.168.1.5

When prompted, enter the password for the user name you specified in the ssh-copy-id command.

ssh-copy-id assumes the public key is named id_rsa.pub and is stored the default location (on macOS and Linux, ~/.ssh/ and on Windows, C:\Users\<user-name>\.ssh). If you gave the public key a different name or stored it in a different location, you must specify the fully qualified path to your SSH public key using the -i option to ssh-copy-id (for example, ssh-copy-id -i ~/my/path/myKey.pub). For more information about creating SSH keys and copying public keys, see SSH-COPY-ID.

To create an SSH key using PuTTYgen (Windows only)

1. Make sure you have the OpenSSH server and client installed on your device under test. For more information, see OpenSSH.
2. Install PuTTYgen on your device under test.
3. Open PuTTYgen.
4. Choose Generate and move your mouse cursor inside the box to generate a private key.
5. From the Conversions menu, choose Export OpenSSH key, and save the private key with a .pem file extension.
6. Add the public key to the /home/<user>/.ssh/authorized_keys file on device under test.

   a. Copy the public key text from the PuTTYgen window.
   b. Use PuTTY to create a session on your device under test.
Configure your device to run IDT tests

i. From a command prompt or Windows Powershell window, run the following command:

```
C:/<path-to-putty>/putty.exe -ssh <user>@<dut-ip-address>
```

ii. When prompted, enter your device's password.

iii. Use vi or another text editor to append the public key to the `/home/<user>/.ssh/authorized_keys` file on your device under test.

7. Update your `device.json` file with your user name, the IP address, and path to the private key file that you just saved on your host computer for each device under test. For more information, see the section called "Configure device.json" (p. 679). Make sure you provide the full path and file name to the private key and use forward slashes (`/`). For example, for the Windows path `C:\DT\privatekey.pem`, use `C:/DT/privatekey.pem` in the `device.json` file.

Configure user permissions on your device

IDT performs operations on various directories and files in a device under test. Some of these operations require elevated permissions (using `sudo`). To automate these operations, IDT for AWS IoT Greengrass V2 must be able to run commands with sudo without being prompted for a password.

Follow these steps on the device under test to allow sudo access without being prompted for a password.

**Note**
username refers to the SSH user used by IDT to access the device under test.

**To add the user to the sudo group**

1. On the device under test, run `sudo usermod -aG sudo <username>`.
2. Sign out and then sign back in for changes to take effect.
3. To verify your user name was added successfully, run `sudo echo test`. If you are not prompted for a password, your user is configured correctly.
4. Open the `/etc/sudoers` file and add the following line to the end of the file:

```
<ssh-username> ALL=(ALL) NOPASSWD: ALL
```

Configure your device to test optional features

This section describes the device requirements to run IDT tests for optional Docker and machine learning (ML) features. You must make sure your device meets these requirements only if you want to test these features. Otherwise, continue to the section called “Configure IDT settings” (p. 678).

**Topics**
- Docker qualification requirements (p. 677)
- ML qualification requirements (p. 678)

**Docker qualification requirements**

IDT for AWS IoT Greengrass V2 provides Docker qualification tests to validate that your devices can use the AWS-provided Docker application manager (p. 162) component to download Docker container images that you can run using custom Docker container components. For information about creating custom Docker components, see Run a Docker container (p. 331).

To run Docker qualification tests, your devices under test must meet the following requirements to deploy the Docker application manager component.
Configure IDT settings

- Docker Engine 1.9.1 or later installed and running on your Greengrass core device. Version 20.10 is the latest version that is verified to work with the connector. You must install Docker directly on the core device before you deploy custom components that run Docker containers.
- The Docker daemon started and running on the core device before you deploy this component.
- Root user permissions or Docker configured for you to run it as a non-root user. Adding a user to the docker group enables you to call docker commands without sudo. To add ggc_user, or the non-root user that you use to run AWS IoT Greengrass, to the docker group that you configure, run `sudo usermod -aG docker user-name`.

ML qualification requirements

IDT for AWS IoT Greengrass V2 provides ML qualification tests to validate that your devices can use the AWS-provided machine learning components (p. 216) to perform ML inference locally using the Deep Learning Runtime or TensorFlow Lite ML frameworks. For more information about running ML inference on Greengrass devices, see Perform machine learning inference (p. 558).

To run ML qualification tests, your devices under test must meet the following requirements to deploy the machine learning components.

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies:

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

Configure IDT settings to run the AWS IoT Greengrass qualification suite

Before you run tests, you must configure settings for AWS credentials and devices on your host computer.

Configure AWS credentials in config.json

You must configure your IAM user credentials in the `<device_tester_extract_location>/configs/config.json` file. Use the credentials for the IDT for AWS IoT Greengrass V2 user created in the section called “Create and configure an AWS account” (p. 666). You can specify your credentials in one of two ways:

- In a credentials file
- As environment variables

Configure AWS credentials with a credentials file

IDT uses the same credentials file as the AWS CLI. For more information, see Configuration and credential files.

The location of the credentials file varies, depending on the operating system you are using:

- macOS, Linux: `~/.aws/credentials`
- Windows: C:\Users\UserName\.aws\credentials
Add your AWS credentials to the credentials file in the following format:

```plaintext
[default]
aws_access_key_id = <your_access_key_id>
aws_secret_access_key = <your_secret_access_key>
```

To configure IDT for AWS IoT Greengrass V2 to use AWS credentials from your credentials file, edit your config.json file as follows:

```json
{
  "awsRegion": "region",
  "auth": {
    "method": "file",
    "credentials": {
      "profile": "default"
    }
  }
}
```

**Note**

If you do not use the default AWS profile, be sure to change the profile name in your config.json file. For more information, see Named profiles.

**Configure AWS credentials with environment variables**

Environment variables are variables maintained by the operating system and used by system commands. They are not saved if you close the SSH session. IDT for AWS IoT Greengrass V2 can use the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables to store your AWS credentials.

To set these variables on Linux, macOS, or Unix, use `export`:

```bash
export AWS_ACCESS_KEY_ID=<your_access_key_id>
export AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

To set these variables on Windows, use `set`:

```bash
set AWS_ACCESS_KEY_ID=<your_access_key_id>
set AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

To configure IDT to use the environment variables, edit the auth section in your config.json file. Here is an example:

```json
{
  "awsRegion": "region",
  "auth": {
    "method": "environment"
  }
}
```

**Configure device.json**

In addition to AWS credentials, IDT for AWS IoT Greengrass V2 needs information about the devices that tests are run on. Example information would be IP address, login information, operating system, and CPU architecture.

You must provide this information using the device.json template located in `<device_tester_extract_location>/configs/device.json`:
[{
  "id": "<pool-id>",
  "sku": "<sku>",
  "features": [
  {
    "name": "arch",
    "value": "x86_64 | armv6l | armv7l | aarch64"
  },
  {
    "name": "ml",
    "value": "dlr | tensorflowlite | dlr,tensorflowlite | no"
  },
  {
    "name": "docker",
    "value": "yes | no"
  },
  {
    "name": "streamManagement",
    "value": "yes | no"
  }
  ],
  "devices": [
  {
    "id": "<device-id>",
    "operatingSystem": "linux",
    "connectivity": {
      "protocol": "ssh",
      "ip": "<ip-address>",
      "port": 22,
      "auth": {
        "method": "pki | password",
        "credentials": {
          "user": "<user-name>",
          "privKeyPath": "/path/to/private/key",
          "password": "<password>"
        }
      }
    }
  }
  ]
}]

**Note**
Specify **privKeyPath** only if method is set to **pki**.
Specify **password** only if method is set to **password**.

All properties that contain values are required, as described here:

**id**

A user-defined alphanumeric ID that uniquely identifies a collection of devices called a device pool. Devices that belong to a pool must have identical hardware. When you run a suite of tests, devices in the pool are used to parallelize the workload. Multiple devices are used to run different tests.

**sku**

An alphanumeric value that uniquely identifies the device under test. The SKU is used to track qualified boards.

**Note**
If you want to list your device in the AWS Partner Device Catalog, the SKU you specify here must match the SKU that you use in the listing process.
features

An array that contains the device's supported features. All features are required.

arch

The supported operating system architectures that the test run validates. Valid values are:
- x86_64
- armv6l
- armv7l
- aarch64

ml

Validates that the device meets all of the required technical dependencies to use the AWS-provided machine learning (ML) components.

Enabling this feature also validates that the device can perform ML inference using the Deep Learning Runtime and TensorFlow Lite ML frameworks.

Valid values are any combination of dlr and tensorflowlite, or no.

docker

Validates that the device meets all required technical dependencies to use the AWS-provided Docker application manager (aws.greengrass.DockerApplicationManager) component.

Enabling this feature also validates that the device can download a Docker container image from Amazon Elastic Container Registry (Amazon ECR).

Valid values are any combination of yes or no.

streamManagement

Validates that the device can download, install, and run the AWS IoT Greengrass stream manager (p. 519).

Valid values are any combination of yes or no.

Note

IDT v4.2.0 and later versions support testing the ml, docker, and streamManagement features. If you don't want to test these features, set the corresponding value to no.

devices.id

A user-defined unique identifier for the device being tested.

devices.operatingSystem

The device operating system. Currently, the only supported value is linux.

connectivity.protocol

The communication protocol used to communicate with this device. Currently, the only supported value is ssh for physical devices.

connectivity.ip

The IP address of the device being tested.

This property applies only if connectivity.protocol is set to ssh.

connectivity.port

Optional. The port number to use for SSH connections.

The default value is 22.
Configure IDT settings

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth`

Authentication information for the connection.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth.method`

The authentication method used to access a device over the given connectivity protocol.

Supported values are:
- `pki`
- `password`

`connectivity.auth.credentials`

The credentials used for authentication.

`connectivity.auth.credentials.password`

The password used for signing in to the device being tested.

This value applies only if `connectivity.auth.method` is set to `password`.

`connectivity.auth.credentials.privKeyPath`

The full path to the private key used to sign in to the device under test.

This value applies only if `connectivity.auth.method` is set to `pki`.

`connectivity.auth.credentials.user`

The user name for signing in to the device being tested.

Configure userdata.json

IDT for AWS IoT Greengrass V2 also needs additional information about the location of test artifacts and AWS IoT Greengrass software versions.

You must provide this information using the `userdata.json` template located in `<device_tester_extract_location>/configs/userdata.json`:

```
{
    "TempResourcesDirOnDevice": "/path/to/temp/folder",
    "InstallationDirRootOnDevice": "/path/to/installation/folder",
    "GreengrassNucleusZip": "/path/to/aws.greengrass.nucleus.zip"
}
```

All properties that contain values are required as described here:

`TempResourcesDirOnDevice`

The full path to a temporary folder on the device under test in which to store test artifacts. Make sure that sudo permissions are not required to write to this directory.

**Note**

The contents of this folder are deleted when IDT finishes running a test.

`InstallationDirRootOnDevice`

The full path to a folder on the device in which to install AWS IoT Greengrass.

You must set the required file permissions for this folder. Run the following command for each folder in the installation path.
Run the AWS IoT Greengrass qualification suite

After you set the required configuration (p. 678), you can start the tests. The runtime of the full test suite depends on your hardware. For reference, it takes approximately 30 minutes to complete the full test suite on a Raspberry Pi 3B.

Use the following run-suite command to run a suite of tests.

```
devicetester_[linux | mac | win]_x86-64 run-suite  
  --suite-id suite-id  
  --group-id group-id  
  --pool-id your-device-pool  
  --test-id test-id  
  --update-idt y|n  
  --userdata userdata.json
```

The --userdata option is the only required option for the run-suite command; all other options are optional. For example, you can omit pool-id if you have only one device pool, which is a set of identical devices, defined in your device.json file. Or, you can omit suite-id if you want to run the latest test suite version in the tests folder.

**Note**
IDT prompts you if a newer test suite version is available online. For more information, see the section called “Test suite versions” (p. 663).

**Example commands to run the qualification suite**

The following command line examples show you how to run the qualification tests for a device pool. For more information about run-suite and other IDT commands, see the section called “IDT commands” (p. 684).

Use the following command to run all test groups in a specified test suite. The list-suites command lists the test suites that are in the tests folder.

```
devicetester_[linux | mac | win]_x86-64 run-suite  
  --suite-id GGV2Q_1.0.0  
  --pool-id <pool-id>  
  --userdata userdata.json
```

Use the following command to run a specific test group in a test suite. The list-groups command lists the test groups in a test suite.

```
devicetester_[linux | mac | win]_x86-64 run-suite  
  --suite-id GGV2Q_1.0.0  
  --group-id <group-id>  
```
Run the AWS IoT Greengrass qualification suite

Use the following command to run a specific test case in a test group.

devicetester_[linux | mac | win]_x86-64 run-suite \ -group-id <group-id> \ -test-id <test-id> \ -userdata userdata.json

Use the following command to run multiple test cases in a test group.

devicetester_[linux | mac | win]_x86-64 run-suite \ -group-id <group-id> \ -test-id <test-id1>,<test-id2> \ -userdata userdata.json

Use the following command to list all of the test cases in a test group.

devicetester_[linux | mac | win]_x86-64 list-test-cases --group-id <group-id>

We recommend that you run the dependency checker test group to make sure all Greengrass dependencies are installed before you run related test groups. For example:

- Run `coredependencies` before running core qualification test groups.

**IDT for AWS IoT Greengrass V2 commands**

The IDT commands are located in the `<device-tester-extract-location>/bin` directory. To run a test suite, you provide the command in the following format:

`help`

Lists information about the specified command.

`list-groups`

Lists the groups in a given test suite.

`list-suites`

Lists the available test suites.

`list-supported-products`

Lists the supported products, in this case AWS IoT Greengrass versions, and test suite versions for the current IDT version.

`list-test-cases`

Lists the test cases in a given test group. The following option is supported:

- `group-id`. The test group to search for. This option is required and must specify a single group.

`run-suite`

Runs a suite of tests on a pool of devices. The following are some supported options:

- `suite-id`. The test suite version to run. If not specified, IDT uses the latest version in the tests folder.
- `group-id`. The test groups to run, as a comma-separated list. If not specified, IDT runs all test groups in the test suite.
Understanding results and logs

This section describes how to view and interpret IDT result reports and logs.

Viewing results

While running, IDT writes errors to the console, log files, and test reports. After IDT completes the qualification test suite, it generates two test reports. These reports are located in `<device-tester-extract-location>/results/<execution-id>/`. Both reports capture the results from running the qualification test suite.

The `awsiotdevicetester_report.xml` is the qualification test report that you submit to AWS to list your device in the AWS Partner Device Catalog. The report contains the following elements:

- The IDT version.
- The AWS IoT Greengrass version that was tested.
- The SKU and the device pool name specified in the `device.json` file.
- The features of the device pool specified in the `device.json` file.
- The aggregate summary of test results.
- A breakdown of test results by libraries that were tested based on the device features, such as local resource access, shadow, and MQTT.

The `GGV2Q_Result.xml` report is in JUnit XML format. You can integrate it into continuous integration and deployment platforms like Jenkins, Bamboo, and so on. The report contains the following elements:

- Aggregate summary of test results.
- Breakdown of test results by the AWS IoT Greengrass functionality that was tested.

Interpreting AWS IoT Device Tester results

The report section in `awsiotdevicetester_report.xml` or `awsiotdevicetester_report.xml` lists the tests that were run and the results.

The first XML tag `<testsuites>` contains the summary of the test run. For example:
<testsuites name="GGQ results" time="2299" tests="28" failures="0" errors="0" disabled="0">

Attributes used in the <testsuites> tag

name
The name of the test suite.
time
The time, in seconds, that it took to run the qualification suite.
tests
The number of tests that were run.
failures
The number of tests that were run, but did not pass.
errors
The number of tests that IDT couldn't run.
disabled
Ignore this attribute. It is not used.

The awsiotdevicetester_report.xml file contains an <awsproduct> tag that contains information about the product being tested and the product features that were validated after running a suite of tests.

Attributes used in the <awsproduct> tag

name
The name of the product being tested.
version
The version of the product being tested.
features
The features validated. Features marked as required are required to submit your board for qualification. The following snippet shows how this information appears in the awsiotdevicetester_report.xml file.

```xml
<name="aws-iot-greengrass-v2-core" value="supported" type="required"></feature>
```

If there are no test failures or errors for the required features, your device meets the technical requirements to run AWS IoT Greengrass and can interoperate with AWS IoT services. If you want to list your device in the AWS Partner Device Catalog, you can use this report as qualification evidence.

In the event of test failures or errors, you can identify the test that failed by reviewing the <testsuites> XML tags. The <testsuite> XML tags inside the <testsuites> tag show the test result summary for a test group. For example:

```
<testsuite name="combination" package="" tests="1" failures="0" time="161" disabled="0" errors="0" skipped="0">
```
Use IDT to develop and run your own test suites

Starting in IDT v4.0.1, IDT for AWS IoT Greengrass V2 combines a standardized configuration setup and result format with a test suite environment that enables you to develop custom test suites for your devices and device software. You can add custom tests for your own internal validation or provide them to your customers for device verification.

Use IDT to develop and run custom test suites, as follows:

**To develop custom test suites**
- Create test suites with custom test logic for the Greengrass device that you want to test.
- Provide IDT with your custom test suites to test runners. Include information about specific settings configurations for your test suites.
To run custom test suites

- Set up the device that you want to test.
- Implement the settings configurations as required by the test suites that you want to use.
- Use IDT to run your custom test suites.
- View the test results and execution logs for the tests run by IDT.

Download the latest version of AWS IoT Device Tester for AWS IoT Greengrass

Download the latest version (p. 661) of IDT and extract the software into a location (<device-tester-extract-location>) on your file system where you have read/write permissions.

**Note**

IDT does not support being run by multiple users from a shared location, such as an NFS directory or a Windows network shared folder. We recommend that you extract the IDT package to a local drive and run the IDT binary on your local workstation.

Windows has a path length limitation of 260 characters. If you are using Windows, extract IDT to a root directory like `C:\` or `D:\` to keep your paths under the 260 character limit.

Test suite creation workflow

Test suites are composed of three types of files:

- JSON configuration files that provide IDT with information on how to execute the test suite.
- Test executable files that IDT uses to run test cases.
- Additional files required to run tests.

Complete the following basic steps to create custom IDT tests:

1. Create JSON configuration files (p. 698) for your test suite.
2. Create test case executables (p. 719) that contain the test logic for your test suite.
3. Verify and document the configuration information required for test runners (p. 727) to run the test suite.
4. Verify that IDT can run your test suite and produce test results (p. 734) as expected.

To quickly build a sample custom suite and run it, follow the instructions in Tutorial: Build and run the sample IDT test suite (p. 688).

To get started creating a custom test suite in Python, see Tutorial: Develop a simple IDT test suite (p. 692).

**Tutorial: Build and run the sample IDT test suite**

The AWS IoT Device Tester download includes the source code for a sample test suite. You can complete this tutorial to build and run the sample test suite to understand how you can use IDT for AWS IoT Greengrass to run custom test suites.

In this tutorial, you will complete the following steps:

1. Build the sample test suite (p. 691)
2. Use IDT to run the sample test suite (p. 691)
Prerequisites

To complete this tutorial, you need the following:

- **Host computer requirements**
  - Latest version of AWS IoT Device Tester
  - **Python** 3.7 or later

  To check the version of Python installed on your computer, run the following command:

  ```bash
  python3 --version
  ```

  On Windows, if using this command returns an error, then use `python --version` instead. If the returned version number is 3.7 or greater, then run the following command in a Powershell terminal to set `python3` as an alias for your `python` command:

  ```powershell
  Set-Alias -Name "python3" -Value "python"
  ```

  If no version information is returned or if the version number is less than 3.7, follow the instructions in **Downloading Python** to install Python 3.7+. For more information, see the **Python documentation**.

- **urllib3**

  To verify that `urllib3` is installed correctly, run the following command:

  ```bash
  python3 -c 'import urllib3'
  ```

  If `urllib3` is not installed, run the following command to install it:

  ```bash
  python3 -m pip install urllib3
  ```

- **Device requirements**

  - A device with a Linux operating system and a network connection to the same network as your host computer.

    We recommend that you use a Raspberry Pi with Raspberry Pi OS. Make sure you set up SSH on your Raspberry Pi to remotely connect to it.

Configure device information for IDT

Configure your device information for IDT to run the test. You must update the `device.json` template located in the `<device-tester-extract-location>/configs` folder with the following information.

```json
[
  {
    "id": "pool",
    "sku": "N/A",
    "devices": [
    {
      "id": "<device-id>",
      "connectivity": {
      "protocol": "ssh",
      "ip": "<ip-address>"
      }
    ]
  }
```
"port": "<port>",
"auth": {
    "method": "pki | password",
    "credentials": {
        "user": "<user-name>",
        "privKeyPath": "/path/to/private/key",
        "password": "<password>"
    }
}
}
}
]

In the devices object, provide the following information:

id

A user-defined unique identifier for your device.

connectivity.ip

The IP address of your device.

connectivity.port

Optional. The port number to use for SSH connections to your device.

connectivity.auth

Authentication information for the connection.

This property applies only if connectivity.protocol is set to ssh.

connectivity.auth.method

The authentication method used to access a device over the given connectivity protocol.

Supported values are:
• pki
• password

connectivity.auth.credentials

The credentials used for authentication.

connectivity.auth.credentials.user

The user name used to sign in to your device.

connectivity.auth.credentials.privKeyPath

The full path to the private key used to sign in to your device.

This value applies only if connectivity.auth.method is set to pki.

devices.connectivity.auth.credentials.password

The password used for signing in to your device.

This value applies only if connectivity.auth.method is set to password.

Note
Specify privKeyPath only if method is set to pki.
Specify password only if method is set to password.

Build the sample test suite

The `<device-tester-extract-location>/samples/python` folder contains sample configuration files, source code, and the IDT Client SDK that you can combine into a test suite using the provided build scripts. The following directory tree shows the location of these sample files:

```
<device-tester-extract-location>
### ...
### tests
### samples
  ### ...
    ### python
    ### ...
    ### configuration
    ### src
    ### build-scripts
    ### build.sh
    ### build.ps1
### sdks
### ...
### python
### idt_client
```

To build the test suite, run the following commands on your host computer:

**Windows**

```bash
cd <device-tester-extract-location>/samples/python/build-scripts
./build.ps1
```

**Linux, macOS, or UNIX**

```bash
cd <device-tester-extract-location>/samples/python/build-scripts
./build.sh
```

This creates the sample test suite in the `IDTSampleSuitePython_1.0.0` folder within the `<device-tester-extract-location>/tests` folder. Review the files in the `IDTSampleSuitePython_1.0.0` folder to understand how the sample test suite is structured and see various examples of test case executables and test configuration JSON files.

Next step: Use IDT to run the sample test suite (p. 691) that you created.

Use IDT to run the sample test suite

To run the sample test suite, run the following commands on your host computer:

```bash
cd <device-tester-extract-location>/bin
./devicetester_[linux | mac | win_x86-64] run-suite --suite-id IDTSampleSuitePython
```

IDT runs the sample test suite and streams the results to the console. When the test has finished running, you see the following information:

```
========== Test Summary ==========
Execution Time: 5s
Tests Completed: 4
```

691
Tests Passed:           4
Tests Failed:           0
Tests Skipped:          0
----------------------------------
Test Groups:            
  sample_group:       PASSED
----------------------------------
Path to IoT Device Tester Report: /path/to/devicetester/
  results/87e673c6-1226-11eb-9269-8c6590419f30/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/devicetester/
  results/87e673c6-1226-11eb-9269-8c6590419f30/logs
Path to Aggregated JUnit Report: /path/to/devicetester/
  results/87e673c6-1226-11eb-9269-8c6590419f30/IDTSampleSuitePython_Report.xml

Troubleshooting

Use the following information to help resolve any issues with completing the tutorial.

Test case does not run successfully
If the test does not run successfully, IDT streams the error logs to the console that can help you troubleshoot the test run. Make sure that you meet all the prerequisites (p. 689) for this tutorial.

Cannot connect to the device under test
Verify the following:

- Your device.json file contains the correct IP address, port, and authentication information.
- You can connect to your device over SSH from your host computer.

Tutorial: Develop a simple IDT test suite

A test suite combines the following:

- Test executables that contain the test logic
- JSON configuration files that describe the test suite

This tutorial shows you how to use IDT for AWS IoT Greengrass to develop a Python test suite that contains a single test case. In this tutorial, you will complete the following steps:

1. Create a test suite directory (p. 693)
2. Create JSON configuration files (p. 693)
3. Create the test case executable (p. 695)
4. Run the test suite (p. 697)

Prerequisites

To complete this tutorial, you need the following:

- **Host computer requirements**
  - Latest version of AWS IoT Device Tester
  - Python 3.7 or later

To check the version of Python installed on your computer, run the following command:
python3 --version

On Windows, if using this command returns an error, then use python --version instead. If the returned version number is 3.7 or greater, then run the following command in a Powershell terminal to set python3 as an alias for your python command.

Set-Alias -Name "python3" -Value "python"

If no version information is returned or if the version number is less than 3.7, follow the instructions in Downloading Python to install Python 3.7+. For more information, see the Python documentation.

- urllib3

To verify that urllib3 is installed correctly, run the following command:

python3 -c 'import urllib3'

If urllib3 is not installed, run the following command to install it:

python3 -m pip install urllib3

- Device requirements

- A device with a Linux operating system and a network connection to the same network as your host computer.

We recommend that you use a Raspberry Pi with Raspberry Pi OS. Make sure you set up SSH on your Raspberry Pi to remotely connect to it.

Create a test suite directory

IDT logically separates test cases into test groups within each test suite. Each test case must be inside a test group. For this tutorial, create a folder called MyTestSuite_1.0.0 and create the following directory tree within this folder:

```
MyTestSuite_1.0.0
  ### suite
  ### myTestGroup
  ### myTestCase
```

Create JSON configuration files

Your test suite must contain the following required JSON configuration files (p. 698):

**Required JSON files**

**suite.json**

contains information about the test suite. See Configure suite.json (p. 699).

**group.json**

contains information about a test group. You must create a group.json file for each test group in your test suite. See Configure group.json (p. 700).
test.json

Contains information about a test case. You must create a test.json file for each test case in your test suite. See Configure test.json (p. 701).

1. In the MyTestSuite_1.0.0/suite folder, create a suite.json file with the following structure:

```json
{
    "id": "MyTestSuite_1.0.0",
    "title": "My Test Suite",
    "details": "This is my test suite.",
    "userDataRequired": false
}
```

2. In the MyTestSuite_1.0.0/myTestGroup folder, create a group.json file with the following structure:

```json
{
    "id": "MyTestGroup",
    "title": "My Test Group",
    "details": "This is my test group.",
    "optional": false
}
```

3. In the MyTestSuite_1.0.0/myTestGroup/myTestCase folder, create a test.json file with the following structure:

```json
{
    "id": "MyTestCase",
    "title": "My Test Case",
    "details": "This is my test case.",
    "execution": {
        "timeout": 300000,
        "linux": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        },
        "mac": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        },
        "win": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        }
    }
}
```

The directory tree for your MyTestSuite_1.0.0 folder should now look like the following:

```
MyTestSuite_1.0.0
    suite
        suite.json
        myTestGroup
```
Get the IDT client SDK

You use the IDT client SDK (p. 719) to enable IDT to interact with the device under test and to report test results. For this tutorial, you will use the Python version of the SDK.

From the `<device-tester-extract-location>/sdks/python` folder, copy the `idt_client` folder to your `MyTestSuite_1.0.0/suite/myTestGroup/myTestCase` folder.

To verify that the SDK was successfully copied, run the following command.

```bash
cd MyTestSuite_1.0.0/suite/myTestGroup/myTestCase
python3 -c 'import idt_client'
```

Create the test case executable

Test case executables contain the test logic that you want to run. A test suite can contain multiple test case executables. For this tutorial, you will create only one test case executable.

1. Create the test suite file.

   In the `MyTestSuite_1.0.0/suite/myTestGroup/myTestCase` folder, create a `myTestCase.py` file with the following content:

   ```python
   from idt_client import *
   def main():
       # Use the client SDK to communicate with IDT
       client = Client()
   if __name__ == '__main__':
       main()
   ```

2. Use client SDK functions to add the following test logic to your `myTestCase.py` file:

   a. Run an SSH command on the device under test.

   ```python
   from idt_client import *
   def main():
       # Use the client SDK to communicate with IDT
       client = Client()
   
       # Create an execute on device request
       exec_req = ExecuteOnDeviceRequest(ExecuteOnDeviceCommand("echo 'hello world!'"))

       # Run the command
       exec_resp = client.execute_on_device(exec_req)

       # Print the standard output
       print(exec_resp.stdout)
   
   if __name__ == '__main__':
       main()
   ```

   b. Send the test result to IDT.
from idt_client import *

def main():
    # Use the client SDK to communicate with IDT
    client = Client()

    # Create an execute on device request
    exec_req = ExecuteOnDeviceRequest(ExecuteOnDeviceCommand("echo 'hello world'"))

    # Run the command
    exec_resp = client.execute_on_device(exec_req)

    # Print the standard output
    print(exec_resp.stdout)

    # Create a send result request
    sr_req = SendResultRequest(TestResult(passed=True))

    # Send the result
    client.send_result(sr_req)

if __name__ == '__main__':
    main()

---

**Configure device information for IDT**

Configure your device information for IDT to run the test. You must update the `device.json` template located in the `<device-tester-extract-location>/configs` folder with the following information.

```json
[
    {
        "id": "pool",
        "sku": "N/A",
        "devices": [
            {
                "id": "<device-id>",
                "connectivity": {
                    "protocol": "ssh",
                    "ip": "<ip-address>",
                    "port": "<port>",
                    "auth": {
                        "method": "pki | password",
                        "credentials": {
                            "user": "<user-name>",
                            "privKeyPath": "<path/to/private/key>",
                            "password": "<password>"
                        }
                    }
                }
            }
        ]
    }
]
```

In the `devices` object, provide the following information:

- **id**
  A user-defined unique identifier for your device.
connectivity.ip

The IP address of your device.

connectivity.port

Optional. The port number to use for SSH connections to your device.

connectivity.auth

Authentication information for the connection.
This property applies only if connectivity.protocol is set to ssh.

connectivity.auth.method

The authentication method used to access a device over the given connectivity protocol.
Supported values are:
• pki
• password

connectivity.auth.credentials

The credentials used for authentication.

connectivity.auth.credentials.user

The user name used to sign in to your device.

connectivity.auth.credentials.privKeyPath

The full path to the private key used to sign in to your device.
This value applies only if connectivity.auth.method is set to pki.

devices.connectivity.auth.credentials.password

The password used for signing in to your device.
This value applies only if connectivity.auth.method is set to password.

Note
Specify privKeyPath only if method is set to pki.
Specify password only if method is set to password.

Run the test suite

After you create your test suite, you want to make sure that it functions as expected. Complete the following steps to run the test suite with your existing device pool to do so.

1. Copy your MyTestSuite_1.0.0 folder into <device-tester-extract-location>/tests.
2. Run the following commands:

   ```
   cd <device-tester-extract-location>/bin
   ./devicetester_[linux | mac | win_x86-64] run-suite --suite-id MyTestSuite
   ```

IDT runs your test suite and streams the results to the console. When the test has finished running, you see the following information:
Create IDT test suite configuration files

This section describes the formats in which you create JSON configuration files that you include when you write a custom test suite.

**Required JSON files**

- **suite.json**
  
  Contains information about the test suite. See [Configure suite.json](#) (p. 699).

---

Troubleshooting

Use the following information to help resolve any issues with completing the tutorial.

**Test case does not run successfully**

If the test does not run successfully, IDT streams the error logs to the console that can help you troubleshoot the test run. Before you check the error logs, verify the following:

- The IDT client SDK is in the correct folder as described in this step (p. 695).
- You meet all the prerequisites (p. 692) for this tutorial.

**Cannot connect to the device under test**

Verify the following:

- Your device.json file contains the correct IP address, port, and authentication information.
- You can connect to your device over SSH from your host computer.

---

Path to IoT Device Tester Report: /path/to/devicetester/results/9a52f362-1227-11eb-86c9-8c8590419f30/awsiotdevicetester_report.xml

Path to Test Execution Logs: /path/to/devicetester/results/9a52f362-1227-11eb-86c9-8c8590419f30/logs

Path to Aggregated JUnit Report: /path/to/devicetester/results/9a52f362-1227-11eb-86c9-8c8590419f30/MyTestSuite_Report.xml
group.json

Contains information about a test group. You must create a group.json file for each test group in your test suite. See Configure group.json (p. 700).

test.json

Contains information about a test case. You must create a test.json file for each test case in your test suite. See Configure test.json (p. 701).

Optional JSON files

state_machine.json

Defines how tests are run when IDT runs the test suite. See Configure state_machine.json (p. 703).

userdata_schema.json

Defines the schema for the userdata.json file (p. 730) that test runners can include in their setting configuration. The userdata.json file is used for any additional configuration information that is required to run the test but is not present in the device.json file. See Configure userdata_schema.json (p. 703).

JSON configuration files are placed in your <custom-test-suite-folder> as shown here.

```plaintext
<custom-test-suite-folder>
### suite
### suite.json
### state_machine.json
### userdata_schema.json
### <test-group-folder>
### group.json
### <test-case-folder>
### test.json
```

Configure suite.json

The suite.json file sets environment variables and determines whether user data is required to run the test suite. Use the following template to configure your <custom-test-suite-folder>/suite/suite.json file:

```json
{
    "id": "<suite-name>_<suite-version>",
    "title": "<suite-title>",
    "details": "<suite-details>",
    "userDataRequired": true | false,
    "environmentVariables": [
        {
            "key": "<name>",
            "value": "<value>",
        },
        ...
        {
            "key": "<name>",
            "value": "<value>",
        }
    ]
}
```

All fields that contain values are required as described here:
id

A unique user-defined ID for the test suite. The value of id must match the name of the test suite folder in which the suite.json file is located. The suite name and suite version must also meet the following requirements:
• <suite-name> cannot contain underscores.
• <suite-version> is denoted as x.x.x, where x is a number.

The ID is shown in IDT-generated test reports.

title

A user-defined name for the product or feature being tested by this test suite. The name is displayed in the IDT CLI for test runners.

details

A short description of the purpose of the test suite.

userDataRequired

Defines whether test runners need to include custom information in a userdata.json file. If you set this value to true, you must also include the userdata_schema.json file (p. 703) in your test suite folder.

environmentVariables

Optional. An array of environment variables to set for this test suite.

environmentVariables.key

The name of the environment variable.

environmentVariables.value

The value of the environment variable.

Configure group.json

The group.json file defines whether a test group is required or optional. Use the following template to configure your <custom-test-suite-folder>/suite/<test-group>/group.json file:

```json
{
  "id": "<group-id>",
  "title": "<group-title>",
  "details": "<group-details>",
  "optional": true | false,
}
```

All fields that contain values are required as described here:

id

A unique user-defined ID for the test group. The value of id must match the name of the test group folder in which the group.json file is located, and can't contain underscores (_). The ID is used in IDT-generated test reports.

title

A descriptive name for the test group. The name is displayed in the IDT CLI for test runners.

details

A short description of the purpose of the test group.
optional

Optional. Set to true to display this test group as an optional group after IDT finishes running required tests. Default value is false.

Configure test.json

The test.json file determines the test case executables and the environment variables that are used by a test case. For more information about creating test case executables, see Create IDT test case executables (p. 719).

Use the following template to configure your <custom-test-suite-folder>/suite/<test-group>/<test-case>/test.json file:

```json
{
  "id": "<test-id>",
  "title": "<test-title>",
  "details": "<test-details>",
  "requireDUT": true | false,
  "requiredResources": [
    {
      "name": "<resource-name>",
      "features": [
        {
          "name": "<feature-name>",
          "version": "<feature-version>",
          "jobSlots": <job-slots>
        }
      ]
    }
  ],
  "execution": {
    "timeout": <timeout>,
    "mac": {
      "cmd": "/path/to/executable",
      "args": [<argument>]
    },
    "linux": {
      "cmd": "/path/to/executable",
      "args": [<argument>]
    },
    "win": {
      "cmd": "/path/to/executable",
      "args": [<argument>]
    }
  },
  "environmentVariables": [
    {
      "key": "<name>",
      "value": "<value>",
    }
  ]
}
```

All fields that contain values are required as described here:
Create IDT test suite configuration files

id

A unique user-defined ID for the test case. The value of id must match the name of the test case folder in which the test.json file is located, and can't contain underscores (_). The ID is used in IDT-generated test reports.

title

A descriptive name for the test case. The name is displayed in the IDT CLI for test runners.
details

A short description of the purpose of the test case.
requireDUT

Optional. Set to true if a device is required to run this test, otherwise set to false. Default value is true. Test runners will configure the devices they will use to run the test in their device.json file.
requiredResources

Optional. An array that provides information about resource devices needed to run this test.
requiredResources.name

The unique name to give the resource device when this test is running.
requiredResources.features

An array of user-defined resource device features.
requiredResources.features.name

The name of the feature. The device feature for which you want to use this device. This name is matched against the feature name provided by the test runner in the resource.json file.
requiredResources.features.version

Optional. The version of the feature. This value is matched against the feature version provided by the test runner in the resource.json file. If a version is not provided, then the feature is not checked. If a version number is not required for the feature, leave this field blank.
requiredResources.features.jobSlots

Optional. The number of simultaneous tests that this feature can support. The default value is 1. If you want IDT to use distinct devices for individual features, then we recommend that you set this value to 1.
execution.timeout

The amount of time (in milliseconds) that IDT waits for the test to finish running. For more information about setting this value, see Create IDT test case executables (p. 719).
execution.os

The test case executables to run based on the operating system of the host computer that runs IDT. Supported values are linux, mac, and win.
execution.os.cmd

The path to the test case executable that you want to run for the specified operating system. This location must be in the system path.
execution.os.args

Optional. The arguments to provide to run the test case executable.
environmentVariables

Optional. An array of environment variables set for this test case.
Configure the IDT state machine

A state machine is a construct that controls the test suite execution flow. It determines the starting state of a test suite, manages state transitions based on user-defined rules, and continues to transition through those states until it reaches the end state.

If your test suite doesn’t include a user-defined state machine, IDT will generate a state machine for you. The default state machine performs the following functions:

- Provides test runners with the ability to select and run specific test groups, instead of the entire test suite.
- If specific test groups are not selected, runs every test group in the test suite in a random order.
- Generates reports and prints a console summary that shows the test results for each test group and test case.

For more information about how the IDT state machine functions, see Configure the IDT state machine (p. 703).

Configure userdata_schema.json

The userdata_schema.json file determines the schema in which test runners provide user data. User data is required if your test suite requires information that is not present in the device.json file. For example, your tests might need Wi-Fi network credentials, specific open ports, or certificates that a user must provide. This information can be provided to IDT as an input parameter called userdata, the value for which is a userdata.json file, that users create in their <device-tester-extract-location>/config folder. The format of the userdata.json file is based on the userdata_schema.json file that you include in the test suite.

To indicate that test runners must provide a userdata.json file:

1. In the suite.json file, set userDataRequired to true.
2. In your <custom-test-suite-folder>, create a userdata_schema.json file.
3. Edit the userdata_schema.json file to create a valid IETF Draft v4 JSON Schema.

When IDT runs your test suite, it automatically reads the schema and uses it to validate the userdata.json file provided by the test runner. If valid, the contents of the userdata.json file are available in both the IDT context (p. 724) and in the state machine context (p. 711).

Configure the IDT state machine

A state machine is a construct that controls the test suite execution flow. It determines the starting state of a test suite, manages state transitions based on user-defined rules, and continues to transition through those states until it reaches the end state.
Configure the IDT state machine

If your test suite doesn't include a user-defined state machine, IDT will generate a state machine for you. The default state machine performs the following functions:

- Provides test runners with the ability to select and run specific test groups, instead of the entire test suite.
- If specific test groups are not selected, runs every test group in the test suite in a random order.
- Generates reports and prints a console summary that shows the test results for each test group and test case.

The state machine for an IDT test suite must meet the following criteria:

- Each state corresponds to an action for IDT to take, such as to run a test group or produce a report file.
- Transitioning to a state executes the action associated with the state.
- Each state defines the transition rule for the next state.
- The end state must be either Succeed or Fail.

State machine format

You can use the following template to configure your own `<custom-test-suite-folder>/suite/state_machine.json` file:

```json
{
  "Comment": "<description>",
  "StartAt": "<state-name>",
  "States": {
    "<state-name>": {
      "Type": "<state-type>",
      // Additional state configuration
    }
  }
  // Required states
  "Succeed": {"Type": "Succeed"},
  "Fail": {"Type": "Fail"}
}
```

All fields that contain values are required as described here:

**Comment**

A description of the state machine.

**StartAt**

The name of the state at which IDT starts running the test suite. The value of StartAt must be set to one of the states listed in the States object.

**States**

An object that maps user-defined state names to valid IDT states. Each States.state-name object contains the definition of a valid state mapped to the state-name.

The States object must include the Succeed and Fail states. For information about valid states, see Valid states and state definitions (p. 705).
Valid states and state definitions

This section describes the state definitions of all of the valid states that can be used in the IDT state machine. Some of the following states support configurations at the test case level. However, we recommend that you configure state transition rules at the test group level instead of the test case level unless absolutely necessary.

State definitions
- RunTask (p. 705)
- Choice (p. 706)
- Parallel (p. 707)
- AddProductFeatures (p. 708)
- Report (p. 710)
- LogMessage (p. 710)
- SelectGroup (p. 711)
- Fail (p. 711)
- Succeed (p. 711)

RunTask

The RunTask state runs test cases from a test group defined in the test suite.

```
{
    "Type": "RunTask",
    "Next": "<state-name>",
    "TestGroup": "<group-id>",
    "TestCases": ["<test-id>"
    ],
    "ResultVar": "<result-name>"
}
```

All fields that contain values are required as described here:

**Next**

The name of the state to transition to after executing the actions in the current state.

**TestGroup**

Optional. The ID of the test group to run. If this value is not specified, then IDT runs the test group that the test runner selects.

**TestCases**

Optional. An array of test case IDs from the group specified in TestGroup. Based on the values of TestGroup and TestCases, IDT determines the test execution behavior as follows:

- When both TestGroup and TestCases are specified, IDT runs the specified test cases from the test group.
- When TestCases are specified but TestGroup is not specified, IDT runs the specified test cases.
- When TestGroup is specified, but TestCases is not specified, IDT runs all of the test cases within the specified test group.
- When neither TestGroup or TestCases is specified, IDT runs all test cases from the test group that the test runner selects from the IDT CLI. To enable group selection for test runners, you must
include both RunTask and Choice states in your stateMachine.json file. For an example of how this works, see Example state machine: Run user-selected test groups (p. 715).

For more information about enabling IDT CLI commands for test runners, see the section called “Enable IDT CLI commands” (p. 721).

ResultVar

The name of the context variable to set with the results of the test run. Do not specify this value if you did not specify a value for TestGroup. IDT sets the value of the variable that you define in ResultVar to true or false based on the following:

- If the variable name is of the form text_text_passed, then the value is set to whether all tests in the first test group passed or were skipped.
- In all other cases, the value is set to whether all tests in all test groups passed or were skipped.

Typically, you will use RunTask state to specify a test group ID without specifying individual test case IDs, so that IDT will run all of the test cases in the specified test group. All test cases that are run by this state run in parallel, in a random order. However, if all of the test cases require a device to run, and only a single device is available, then the test cases will run sequentially instead.

Error handling

If any of the specified test groups or test case IDs are not valid, then this state issues the RunTaskError execution error. If the state encounters an execution error, then it also sets the hasExecutionError variable in the state machine context to true.

Choice

The Choice state lets you dynamically set the next state to transition to based on user-defined conditions.

```
{
    "Type": "Choice",
    "Default": "<state-name>",
    "FallthroughOnError": true | false,
    "Choices": [
        {
            "Expression": "<expression>",
            "Next": "<state-name>"
        }
    ]
}
```

All fields that contain values are required as described here:

Default

The default state to transition to if none of the expressions defined in Choices can be evaluated to true.

FallthroughOnError

Optional. Specifies the behavior when the state encounters an error in evaluating expressions. Set to true if you want to skip an expression if the evaluation results in an error. If no expressions match, then the state machine transitions to the Default state. If the FallthroughOnError value is not specified, it defaults to false.

Choices

An array of expressions and states to determine which state to transition to after executing the actions in the current state.
Choices.Expression

An expression string that evaluates to a boolean value. If the expression evaluates to true, then the state machine transitions to the state defined in Choices.Next. Expression strings retrieve values from the state machine context and then perform operations on them to arrive at a boolean value. For information about accessing the state machine context, see State machine context (p. 711).

Choices.Next

The name of the state to transition to if the expression defined in Choices.Expression evaluates to true.

Error handling

The Choice state can require error handling in the following cases:

- Some variables in the choice expressions don’t exist in the state machine context.
- The result of an expression is not a boolean value.
- The result of a JSON lookup is not a string, number, or boolean.

You cannot use a Catch block to handle errors in this state. If you want to stop executing the state machine when it encounters an error, you must set FallthroughOnError to false. However, we recommend that you set FallthroughOnError to true, and depending on your use case, do one of the following:

- If a variable you are accessing is expected to not exist in some cases, then use the value of Default and additional Choices blocks to specify the next state.
- If a variable that you are accessing should always exist, then set the Default state to Fail.

Parallel

The Parallel state lets you define and run new state machines in parallel with each other.

```json
{
   "Type": "Parallel",
   "Next": "<state-name>",
   "Branches": [
      <state-machine-definition>
   ]
}
```

All fields that contain values are required as described here:

Next

The name of the state to transition to after executing the actions in the current state.

Branches

An array of state machine definitions to run. Each state machine definition must contain its own StartAt, Succeed, and Fail states. The state machine definitions in this array cannot reference states outside of their own definition.

Note

Because each branch state machine shares the same state machine context, setting variables in one branch and then reading those variables from another branch might result in unexpected behavior.
The Parallel state moves to the next state only after it runs all of the branch state machines. Each state that requires a device will wait to run until the device is available. If multiple devices are available, this state runs test cases from multiple groups in parallel. If enough devices are not available, then test cases will run sequentially. Because test cases are run in a random order when they run in parallel, different devices might be used to run tests from the same test group.

**Error handling**

Make sure that both the branch state machine and the parent state machine transition to the Fail state to handle execution errors.

Because branch state machines do not transmit execution errors to the parent state machine, you cannot use a Catch block to handle execution errors in branch state machines. Instead, use the hasExecutionErrors value in the shared state machine context. For an example of how this works, see Example state machine: Run two test groups in parallel (p. 717).

**AddProductFeatures**

The AddProductFeatures state lets you add product features to the awsiotdevicetester_report.xml file generated by IDT.

A product feature is user-defined information about specific criteria that a device might meet. For example, the MQTT product feature can designate that the device publishes MQTT messages properly. In the report, product features are set as supported, not-supported, or a custom value, based on whether specified tests passed.

**Note**

The AddProductFeatures state does not generate reports by itself. This state must transition to the Report state (p. 710) to generate reports.

```json
{
  "Type": "Parallel",
  "Next": "<state-name>",
  "Features": [
    {
      "Feature": "<feature-name>",
      "Groups": ["<group-id>"]
    },
    "OneOfGroups": ["<group-id>"],
    "TestCases": ["<test-id>"],
    "IsRequired": true | false,
    "ExecutionMethods": ["<execution-method>"]
  }
}
```

All fields that contain values are required as described here:

**Next**

The name of the state to transition to after executing the actions in the current state.

**Features**

An array of product features to show in the awsiotdevicetester_report.xml file.
Configure the IDT state machine

Feature

The name of the feature

FeatureValue

Optional. The custom value to use in the report instead of supported. If this value is not specified, then based on test results, the feature value is set to supported or not-supported.

If you use a custom value for FeatureValue, you can test the same feature with different conditions, and IDT concatenates the feature values for the supported conditions. For example, the following excerpt shows the MyFeature feature with two separate feature values:

```json
...
{
  "Feature": "MyFeature",
  "FeatureValue": "first-feature-supported",
  "Groups": ["first-feature-group"]
},
{
  "Feature": "MyFeature",
  "FeatureValue": "second-feature-supported",
  "Groups": ["second-feature-group"]
},
...
```

If both test groups pass, then the feature value is set to first-feature-supported, second-feature-supported.

Groups

Optional. An array of test group IDs. All tests within each specified test group must pass for the feature to be supported.

OneOfGroups

Optional. An array of test group IDs. All tests within at least one of the specified test groups must pass for the feature to be supported.

TestCases

Optional. An array of test case IDs. If you specify this value, then the following apply:

- All of the specified test cases must pass for the feature to be supported.
- Groups must contain only one test group ID.
- OneOfGroups must not be specified.

IsRequired

Optional. Set to false to mark this feature as an optional feature in the report. The default value is true.

ExecutionMethods

Optional. An array of execution methods that match the protocol value specified in the device.json file. If this value is specified, then test runners must specify a protocol value that matches one of the values in this array to include the feature in the report. If this value is not specified, then the feature will always be included in the report.

To use the AddProductFeatures state, you must set the value of ResultVar in the RunTask state to one of the following values:

- If you specified individual test case IDs, then set ResultVar to group-id_test-id_passed.
- If you did not specify individual test case IDs, then set ResultVar to group-id_passed.
Configure the IDT state machine

The AddProductFeatures state checks for test results in the following manner:

- If you did not specify any test case IDs, then the result for each test group is determined from the value of the `group-id_passed` variable in the state machine context.
- If you did specify test case IDs, then the result for each of the tests is determined from the value of the `group-id_test-id_passed` variable in the state machine context.

**Error handling**

If a group ID provided in this state is not a valid group ID, then this state results in the AddProductFeaturesError execution error. If the state encounters an execution error, then it also sets the hasExecutionErrors variable in the state machine context to true.

**Report**

The Report state generates the `suite-name_Report.xml` and `awsiotdevicetester_report.xml` files. This state also streams the report to the console.

```json
{
    "Type": "Report",
    "Next": "<state-name>",
}
```

All fields that contain values are required as described here:

**Next**

The name of the state to transition to after executing the actions in the current state.

**Level**

The error level at which to create the log message. If you specify a level that is not valid, this state generates an error message and discards it.

You should always transition to the Report state towards the end of the test execution flow so that test runners can view test results. Typically, the next state after this state is Succeed.

**Error handling**

If this state encounters issues with generating the reports, then it issues the ReportError execution error.

**LogMessage**

The LogMessage state generates the `test_manager.log` file and streams the log message to the console.

```json
{
    "Type": "LogMessage",
    "Next": "<state-name>",
    "Level": "info | warn | error",
    "Message": "<message>"
}
```

All fields that contain values are required as described here:

**Next**

The name of the state to transition to after executing the actions in the current state.

**Level**

The error level at which to create the log message. If you specify a level that is not valid, this state generates an error message and discards it.
Configure the IDT state machine

Message
The message to log.

SelectGroup
The SelectGroup state updates the state machine context to indicate which groups are selected. The values set by this state are used by any subsequent Choice states.

```
{  
    "Type": "SelectGroup",
    "Next": "<state-name>"
    "TestGroups": [  
        <group-id>"  
    ]
}
```

All fields that contain values are required as described here:

Next
The name of the state to transition to after executing the actions in the current state.

TestGroups
An array of test groups that will be marked as selected. For each test group ID in this array, the _groupId_selected_ variable is set to true in the context. Make sure that you provide valid test group IDs because IDT does not validate whether the specified groups exist.

Fail
The Fail state indicates that the state machine did not execute correctly. This is an end state for the state machine, and each state machine definition must include this state.

```
{  
    "Type": "Fail"
}
```

Succeed
The Succeed state indicates that the state machine executed correctly. This is an end state for the state machine, and each state machine definition must include this state.

```
{  
    "Type": "Succeed"
}
```

State machine context
The state machine context is a read-only JSON document that contains data that is available to the state machine during execution. The state machine context is accessible only from the state machine, and contains information that determines the test flow. For example, you can use information configured by test runners in the _userdata.json_ file to determine whether a specific test is required to run.

The state machine context uses the following format:

```
{  
    "pool": {
```
pool

Information about the device pool selected for the test run. For a selected device pool, this information is retrieved from the corresponding top-level device pool array element defined in the device.json file.

userData

Information in the userdata.json file.

cfg

Information pin the config.json file.

suiteFailed

The value is set to false when the state machine starts. If a test group fails in a RunTask state, then this value is set to true for the remaining duration of the state machine execution.

specificTestGroups

If the test runner selects specific test groups to run instead of the entire test suite, this key is created and contains the list of specific test group IDs.

specificTestCases

If the test runner selects specific test cases to run instead of the entire test suite, this key is created and contains the list of specific test case IDs.

hasExecutionErrors

Does not exit when the state machine starts. If any state encounters an execution errors, this variable is created and set to true for the remaining duration of the state machine execution.

You can query the context using JSONPath notation. The syntax for JSONPath queries in state definitions is `{{$ .query }}`. You can use JSONPath queries as placeholder strings within some states. IDT replaces the placeholder strings with the value of the evaluated JSONPath query from the context. You can use placeholders for the following values:

- The TestCases value in RunTask states.
- The Expression value Choice state.

When you access data from the state machine context, make sure the following conditions are met:

- Your JSON paths must begin with ".
- Each value must evaluate to a string, a number, or a boolean.
For more information about using JSONPath notation to access data from the context, see Use the IDT context (p. 724).

**Execution errors**

Execution errors are errors in the state machine definition that the state machine encounters when executing a state. IDT logs information about each error in the `test_manager.log` file and streams the log message to the console.

You can use the following methods to handle execution errors:

- Add a `Catch` block (p. 713) in the state definition.
- Check the value of the `hasExecutionErrors` value (p. 713) in the state machine context.

**Catch**

To use `Catch`, add the following to your state definition:

```json
"Catch": [  
  { 
    "ErrorEquals": [  
      "<error-type>"
    ] 
    "Next": "<state-name>"
  }
]
```

All fields that contain values are required as described here:

**Catch.ErrorEquals**

An array of the error types to catch. If an execution error matches one of the specified values, then the state machine transitions to the state specified in `Catch.Next`. See each state definition for information about the type of error it produces.

**Catch.Next**

The next state to transition to if the current state encounters an execution error that matches one of the values specified in `Catch.ErrorEquals`.

Catch blocks are handled sequentially until one matches. If the no errors match the ones listed in the Catch blocks, then the state machines continues to execute. Because execution errors are a result of incorrect state definitions, we recommend that you transition to the Fail state when a state encounters an execution error.

**hasExecutionError**

When some states encounter execution errors, in addition to issuing the error, they also set the `hasExecutionError` value to `true` in the state machine context. You can use this value to detect when an error occurs, and then use a `Choice` state to transition the state machine to the Fail state.

This method has the following characteristics:

- The state machine does not start with any value assigned to `hasExecutionError`, and this value is not available until a particular state sets it. This means that you must explicitly set the `FallthroughOnError` to `false` for the `Choice` states that access this value to prevent the state machine from stopping if no execution errors occur.
Configure the IDT state machine

- Once it is set to true, hasExecutionError is never set to false or removed from the context. This means that this value is useful only the first time that it is set to true, and for all subsequent states, it does not provide a meaningful value.
- The hasExecutionError value is shared with all branch state machines in the Parallel state, which can result in unexpected results depending on the order in which it is accessed.

Because of these characteristics, we do not recommend that you use this method if you can use a Catch block instead.

Example state machines

This section provides some example state machine configurations.

Examples
- Example state machine: Run a single test group (p. 714)
- Example state machine: Run user-selected test groups (p. 715)
- Example state machine: Run a single test group with product features (p. 716)
- Example state machine: Run two test groups in parallel (p. 717)

Example state machine: Run a single test group

This state machine:
- Runs the test group with id GroupA, which must be present in the suite in a group.json file.
- Checks for execution errors and transitions to Fail if any are found.
- Generates a report and transitions to Succeed if there are no errors, and Fail otherwise.

```json
{
    "Comment": "Runs a single group and then generates a report.",
    "StartAt": "RunGroupA",
    "States": {
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "Report",
            "TestGroup": "GroupA",
            "Catch": [
                {
                    "ErrorEquals": ["RunTaskError"],
                    "Next": "Fail"
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": ["ReportError"],
                    "Next": "Fail"
                }
            ]
        }
    },
    "Succeed": {
```
Example state machine: Run user-selected test groups

This state machine:

- Checks if the test runner selected specific test groups. The state machine does not check for specific test cases because test runners cannot select test cases without also selecting a test group.
- If test groups are selected:
  - Runs the test cases within the selected test groups. To do so, the state machine does not explicitly specify any test groups or test cases in the RunTask state.
  - Generates a report after running all tests and exits.
- If test groups are not selected:
  - Runs tests in test group GroupA.
  - Generates reports and exits.

```json
{
  "Comment": "Runs specific groups if the test runner chose to do that, otherwise runs GroupA.",
  "StartAt": "SpecificGroupsCheck",
  "States": {
    "SpecificGroupsCheck": {
      "Type": "Choice",
      "Default": "RunGroupA",
      "FallthroughOnError": true,
      "Choices": [
        {
          "Expression": "${.specificTestGroups[0]} != ''",
          "Next": "RunSpecificGroups"
        }
      ]
    },
    "RunSpecificGroups": {
      "Type": "RunTask",
      "Next": "Report",
      "Catch": [
        {
          "ErrorEquals": ["RunTaskError"],
          "Next": "Fail"
        }
      ]
    },
    "RunGroupA": {
      "Type": "RunTask",
      "Next": "Report",
      "TestGroup": "GroupA",
      "Catch": [
        {
          "ErrorEquals": ["RunTaskError"],
          "Next": "Fail"
        }
      ]
    }
  }
}
```
Example state machine: Run a single test group with product features

This state machine:

- Runs the test group GroupA.
- Checks for execution errors and transitions to Fail if any are found.
- Adds the FeatureThatDependsOnGroupA feature to the awsiotdevicetester_report.xml file:
  - If GroupA passes, the feature is set to supported.
  - The feature is not marked optional in the report.
- Generates a report and transitions to Succeed if there are no errors, and Fail otherwise

```json
{
   "Comment": "Runs GroupA and adds product features based on GroupA",
   "StartAt": "RunGroupA",
   "States": {
      "RunGroupA": {
         "Type": "RunTask",
         "Next": "AddProductFeatures",
         "TestGroup": "GroupA",
         "ResultVar": "GroupA_passed",
         "Catch": [
            "ErrorEquals": [
               "RunTaskError"
            ],
            "Next": "Fail"
         ]
      },
      "AddProductFeatures": {
         "Type": "AddProductFeatures",
         "Next": "Report",
         "Features": [
            "Feature": "FeatureThatDependsOnGroupA",
            "Groups": [
               "GroupA"
            ]
         ]
      }
   }
}
```
Example state machine: Run two test groups in parallel

This state machine:

- Runs the GroupA and GroupB test groups in parallel. The ResultVar variables stored in the context by the RunTask states in the branch state machines by are available to the AddProductFeatures state.
- Checks for execution errors and transitions to Fail if any are found. This state machine does not use a Catch block because that method does not detect execution errors in branch state machines.
- Adds features to the awsiotdevicetester_report.xml file based on the groups that pass
  - If GroupA passes, the feature is set to supported.
  - The feature is not marked optional in the report.
- Generates a report and transitions to Succeed if there are no errors, and Fail otherwise

If two devices are configured in the device pool, both GroupA and GroupB can run at the same time. However, if either GroupA or GroupB has multiple tests in it, then both devices may be allocated to those tests. If only one device is configured, the test groups will run sequentially.

```json
{
  "Comment": "Runs GroupA and GroupB in parallel",
  "StartAt": "RunGroupAAndB",
  "States": {
    "RunGroupAAndB": {
      "Type": "Parallel",
      "Next": "CheckForErrors",
      "Branches": [
        {
          "Comment": "Run GroupA state machine",
          "StartAt": "RunGroupA",
          "States": {
            "RunGroupA": {
              "Type": "RunTask",
              "Next": "Succeed",
              "TestGroup": "GroupA",
```
Configure the IDT state machine

"ResultVar": "GroupA_passed",
"Catch": [
    { "ErrorEquals": [ "RunTaskError" ],
      "Next": "Fail"
    }
],
"Succeed": { "Type": "Succeed" },
"Fail": { "Type": "Fail" }
},
{
"Comment": "Run GroupB state machine",
"StartAt": "RunGroupB",
"States": {
  "RunGroupA": {
    "Type": "RunTask",
    "Next": "Succeed",
    "TestGroup": "GroupB",
    "ResultVar": "GroupB_passed",
    "Catch": [
      { "ErrorEquals": [ "RunTaskError" ],
        "Next": "Fail"
      }
    ],
    "Succeed": { "Type": "Succeed" },
    "Fail": { "Type": "Fail" }
  }
},
"CheckForErrors": {
  "Type": "Choice",
  "Default": "AddProductFeatures",
  "FallthroughOnError": true,
  "Choices": [
    { "Expression": "$.hasExecutionErrors == true",
      "Next": "Fail"
    }
  ],
  "AddProductFeatures": {
    "Type": "AddProductFeatures",
    "Next": "Report",
    "Features": [
      { "Feature": "FeatureThatDependsOnGroupA",
        "Groups": [ "GroupA" ]
      }
    ]
  }
}
Create IDT test case executables

You can create and place test case executables in a test suite folder in the following ways:

- For test suites that use arguments or environment variables from the test.json files to determine which tests to run, you can create a single test case executable for the entire test suite, or a test executable for each test group in the test suite.
- For a test suite where you want to run specific tests based on specified commands, you create one test case executable for each test case in the test suite.

As a test writer, you can determine which approach is appropriate for your use case and structure your test case executable accordingly. Make sure that you provide the correct test case executable path in each test.json file, and that the specified executable runs correctly.

When all devices are ready for a test case to run, IDT reads the following files:

- The test.json for the selected test case determines the processes to start and the environment variables to set.
- The suite.json for the test suite determines the environment variables to set.

IDT starts the required test executable process based on the commands and arguments specified in the test.json file, and passes the required environment variables to the process.

Use the IDT Client SDK

The IDT Client SDKs let you simplify how you write test logic in your test executable with API commands that you can use interact with IDT and your devices under test. IDT currently provides the following SDKs:
Create IDT test case executables

- IDT Client SDK for Python
- IDT Client SDK for Go

These SDKs are located in the `<device-tester-extract-location>/sdk` folder. When you create a new test case executable, you must copy the SDK that you want to use to the folder that contains your test case executable and reference the SDK in your code. This section provides a brief description of the available API commands that you can use in your test case executables.

**In this section**
- Device interaction (p. 720)
- IDT interaction (p. 720)
- Host interaction (p. 721)

**Device interaction**

The following commands enable you to communicate with the device under test without having to implement any additional device interaction and connectivity management functions.

**ExecuteOnDevice**

Allows test suites to run shell commands on a device that support SSH or Docker shell connections.

**CopyToDevice**

Allows test suites to copy a local file from the host machine that runs IDT to a specified location on a device that supports SSH or Docker shell connections.

**ReadFromDevice**

Allows test suites to read from the serial port of devices that support UART connections.

**Note**

Because IDT does not manage direct connections to devices that are made using device access information from the context, we recommend using these device interaction API commands in your test case executables. However, if these commands do not meet your test case requirements, then you can retrieve device access information from the IDT context and use it to make a direct connection to the device from the test suite.

To make a direct connection, retrieve the information in the `device.connectivity` and the `resource.devices.connectivity` fields for your device under test and for resource devices, respectively. For more information about using the IDT context, see Use the IDT context (p. 724).

**IDT interaction**

The following commands enable your test suites to communicate with IDT.

**PollForNotifications**

Allows test suites to check for notifications from IDT.

**GetContextValue** and **GetContextString**

Allows test suites to retrieve values from the IDT context. For more information, see Use the IDT context (p. 724).

**SendResult**

Allows test suites to report test case results to IDT. This command must be called at the end of each test case in a test suite.
Host interaction

The following command enable your test suites to communicate with the host machine.

PollForNotifications

Allows test suites to check for notifications from IDT.

GetContextValue and GetContextString

Allows test suites to retrieve values from the IDT context. For more information, see Use the IDT context (p. 724).

ExecuteOnHost

Allows test suites to run commands on the local machine and lets IDT manage the test case executable lifecycle.

Enable IDT CLI commands

The run-suite command IDT CLI provides several options that let test runner customize test execution. To allow test runners to use these options to run your custom test suite, you implement support for the IDT CLI. If you do not implement support, test runners will still be able to run tests, but some CLI options will not function correctly. To provide an ideal customer experience, we recommend that you implement support for the following arguments for the run-suite command in the IDT CLI:

timeout-multiplier

Specifies a value greater than 1.0 that will be applied to all timeouts while running tests.

Test runners can use this argument to increase the timeout for the test cases that they want to run. When a test runner specifies this argument in their run-suite command, IDT uses it to calculate the value of the IDT_TEST_TIMEOUT environment variable and sets the config.timeoutMultiplier field in the IDT context. To support this argument, you must do the following:

- Instead of directly using the timeout value from the test.json file, read the IDT_TEST_TIMEOUT environment variable to obtain the correctly calculated timeout value.
- Retrieve the config.timeoutMultiplier value from the IDT context and apply it to long running timeouts.

For more information about exiting early because of timeout events, see Specify exit behavior (p. 723).

stop-on-first-failure

Specifies that IDT should stop running all tests if it encounters a failure.

When a test runner specifies this argument in their run-suite command, IDT will stop running tests as soon as it encounters a failure. However, if test cases are running in parallel, then this can lead to unexpected results. To implement support, make sure that if IDT encounters this event, your test logic instructs all running test cases to stop, clean up temporary resources, and report a test result to IDT. For more information about exiting early on failures, see Specify exit behavior (p. 723).

group-id and test-id

Specifies that IDT should run only the selected test groups or test cases.

Test runners can use these arguments with their run-suite command to specify the following test execution behavior:
Create IDT test case executables

- Run all tests inside the specified test groups.
- Run a selection of tests from within a specified test group.

To support these arguments, the state machine for your test suite must include a specific set of RunTask and Choice states in your state machine. If you are not using a custom state machine, then the default IDT state machine includes the required states for you and you do not need to take additional action. However, if you are using a custom state machine, then use Example state machine: Run user-selected test groups (p. 715) as a sample to add the required states in your state machine.

For more information about IDT CLI commands, see Debug and run custom test suites (p. 734).

Write event logs

While the test is running, you send data to stdout and stderr to write event logs and error messages to the console. For information about the format of console messages, see Console message format (p. 736).

When the IDT finishes running the test suite, this information is also available in the test_manager.log file located in the <devicetester-extract-location>/results/<execution-id>/logs folder.

You can configure each test case to write the logs from its test run, including logs from the device under test, to the <group-id>_<test-id> file located in the <devicetester-extract-location>/results/<execution-id>/logs folder. To do this, retrieve the path to the log file from the IDT context with the testData.logFilePath query, create a file at that path, and write the content that you want to it. IDT automatically updates the path based on the test case that is running. If you choose not to create the log file for a test case, then no file is generated for that test case.

You can also set up your test executable to create additional log files as needed in the <device-tester-extract-location>/logs folder. We recommend that you specify unique prefixes for log file names so your files don't get overwritten.

Report results to IDT

IDT writes test results to the awsiotdevicetester_report.xml and the suite-name_report.xml files. These report files are located in <device-tester-extract-location>/results/<execution-id>/. Both reports capture the results from the test suite execution. For more information about the schemas that IDT uses for these reports, see Review IDT test results and logs (p. 736)

To populate the contents of the suite-name_report.xml file, you must use the SendResult command to report test results to IDT before the test execution finishes. If IDT cannot locate the results of a test, it issues an error for the test case. The following Python excerpt shows the commands to send a test result to IDT:

```python
request-variable = SendResultRequest(TestResult(result))
client.send_result(request-variable)
```

If you do not report results through the API, IDT looks for test results in the test artifacts folder. The path to this folder is stored in the testData.testArtifactsPath filed in the IDT context. In this folder, IDT uses the first alphabetically sorted XML file it locates as the test result.

If your test logic produces JUnit XML results, you can write the test results to an XML file in the artifacts folder to directly provide the results to IDT instead of parsing the results and then using the API to submit them to IDT.
If you use this method, make sure that your test logic accurately summarizes the test results and format your result file in the same format as the `suite-name_report.xml` file. IDT does not perform any validation of the data that you provide, with the following exceptions:

- IDT ignores all properties of the `testsuites` tag. Instead, it calculates the tag properties from other reported test group results.
- At least one `testsuite` tag must exist within `testsuites`.

Because IDT uses the same artifacts folder for all test cases and does not delete result files between test runs, this method might also lead to erroneous reporting if IDT reads the incorrect file. We recommend that you use the same name for the generated XML results file across all test cases to overwrite the results for each test case and make sure that the correct results are available for IDT to use. Although you can use a mixed approach to reporting in your test suite, that is, use an XML result file for some test cases and submit results through the API for others, we do not recommend this approach.

**Specify exit behavior**

Configure your test executable to always exit with an exit code of 0, even if a test case reports a failure or an error result. Use non-zero exit codes only to indicate that a test case did not run or if the test case executable could not communicate any results to IDT. When IDT receives a non-zero exit code, it marks the test case as having encountered an error that prevented it from running.

IDT might request or expect a test case to stop running before it has finished in the following events. Use this information to configure your test case executable to detect each of these events from the test case:

**Timeout**

Occurs when a test case runs for longer than the timeout value specified in the `test.json` file. If the test runner used the `timeout-multiplier` argument to specify a timeout multiplier, then IDT calculates the timeout value with the multiplier.

To detect this event, use the `IDT_TEST_TIMEOUT` environment variable. When a test runner launches a test, IDT sets the value of the `IDT_TEST_TIMEOUT` environment variable to the calculated timeout value (in seconds) and passes the variable to the test case executable. You can read the variable value to set an appropriate timer.

**Interrupt**

Occurs when the test runner interrupts IDT. For example, by pressing `Ctrl+C`.

Because terminals propagate signals to all child processes, you can simply configure a signal handler in your test cases to detect interrupt signals.

Alternatively, you can periodically poll the API to check the value of the `CancellationRequested` boolean in the `PollForNotifications` API response. When IDT receives an interrupt signal, it sets the value of the `CancellationRequested` boolean to `true`.

**Stop on first failure**

Occurs when a test case that is running in parallel with the current test case fails and the test runner used the `stop-on-first-failure` argument to specify that IDT should stop when it encounters any failure.

To detect this event, you can periodically poll the API to check the value of the `CancellationRequested` boolean in the `PollForNotifications` API response. When IDT encounters a failure and is configured to stop on first failure, it sets the value of the `CancellationRequested` boolean to `true`.

When any of these events occur, IDT waits for 5 minutes for any currently running test cases to finish running. If all running test cases do not exit within 5 minutes, IDT forces each of their processes to stop.
If IDT has not received test results before the processes end, it will mark the test cases as having timed out. As a best practice, you should ensure that your test cases perform the following actions when they encounter one of the events:

1. Stop running normal test logic.
2. Clean up any temporary resources, such as test artifacts on the device under test.
3. Report a test result to IDT, such as a test failure or an error.
4. Exit.

Use the IDT context

When IDT runs a test suite, the test suite can access a set of data that can be used to determine how each test runs. This data is called the IDT context. For example, user data configuration provided by test runners in a userdata.json file is made available to test suites in the IDT context.

The IDT context can be considered a read-only JSON document. Test suites can retrieve data from and write data to the context using standard JSON data types like objects, arrays, numbers and so on.

Context schema

The IDT context uses the following format:

```json
{
    "config": {
        "config-json-content": 
        "timeoutMultiplier": timeout-multiplier
    },
    "device": {
        "device-json-device-element"
    },
    "devicePool": {
        "device-json-pool-element"
    },
    "resource": {
        "resource-json-device-element"
    },
    "testData": {
        "awsCredentials": {
            "awsAccessKeyId": "<access-key-id>",
            "awsSecretAccessKey": "<secret-access-key>",
            "awsSessionToken": "<session-token>
        },
        "logFilePath": "<path/to/log/file"
    },
    "userData": {
        "userdata-json-content"
    }
}
```

**config**

Information from the `config.json` file (p. 732). The `config` field also contains the following additional field:
Use the IDT context

config.timeoutMultiplier

The multiplier for the any timeout value used by the test suite. This value is specified by the test runner from the IDT CLI. The default value is 1.

device

Information about the device selected for the test run. This information is equivalent to the devices array element in the `device.json` file (p. 727) for the selected device.

devicePool

Information about the device pool selected for the test run. This information is equivalent to the top-level device pool array element defined in the `device.json` file for the selected device pool.

resource

Information about resource devices from the `resource.json` file.

resource.devices

This information is equivalent to the devices array defined in the `resource.json` file. Each devices element includes the following additional field:

resource.device.name

The name of the resource device. This value is set to the `requiredResource.name` value in the `test.json` file.

testData.awsCredentials

The AWS credentials used by the test to connect to the AWS cloud. This information is obtained from the `config.json` file.

testData.logFilePath

The path to the log file to which the test case writes log messages. The test suite creates this file if it doesn't exist.

userData

Information provided by the test runner in the `userdata.json` file (p. 730).

Access data in the context

You can query the context using JSONPath notation from your JSON files and from your text executable with the GetContextValue and GetContextString APIs. The syntax for JSONPath strings to access the IDT context varies as follows:

- In `suite.json` and `test.json`, you use `{query}`). That is, do not use the root element `$` to start your expression.
- In `statemachine.json`, you use `{query}`).
- In API commands, you use `query` or `{$.query}`), depending on the command. For more information, see the inline documentation in the SDKs.

The following table describes the operators in a typical JSONPath expression:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>The root element. Because the top-level context value for IDT is an object, you will typically use <code>$.</code> to start your queries.</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>.childName</td>
<td>Accesses the child element with name childName from an object. If applied to an array, yields a new array with this operator applied to each element. The element name is case sensitive. For example, the query to access the awsRegion value in the config object is $.config.awsRegion.</td>
</tr>
<tr>
<td>[start:end]</td>
<td>Filters elements from an array, retrieving items beginning from the start index and going up to the end index, both inclusive.</td>
</tr>
<tr>
<td>[index1, index2, ... , indexN]</td>
<td>Filters elements from an array, retrieving items from only the specified indices.</td>
</tr>
<tr>
<td>[?(expr)]</td>
<td>Filters elements from an array using the expr expression. This expression must evaluate to a boolean value.</td>
</tr>
</tbody>
</table>

To create filter expressions, use the following syntax:

```
(jsonPath) | (value) operator (jsonpath) | (value)
```

In this syntax:

- `jsonpath` is a JSONPath that uses standard JSON syntax.
- `value` is any custom value that uses standard JSON syntax.
- `operator` is one of the following operators:
  - `<` (Less than)
  - `<=` (Less than or equal to)
  - `==` (Equal to)
  - `>=` (Greater than or equal to)
  - `>` (Greater than)
  - `=~` (Regular expression match). To use this operator in a filter expression, the JSONPath or value on the left side of your expression must evaluate to a string and the right side must be a pattern value that follows the RE2 syntax.

You can use JSONPath queries in the form `{{{query}}}` as placeholder strings within the `args` and `environmentVariables` fields in `test.json` files and within the `environmentVariables` fields in `suite.json` files. IDT performs a context lookup and populates the fields with the evaluated value of the query. For example, in the `suite.json` file, you can use placeholder strings to specify environment variable values that change with each test case and IDT will populate the environment variables with the correct value for each test case. However, when you use placeholder strings in `test.json` and `suite.json` files, the following considerations apply for your queries:

- You must each occurrence of the `devicePool` key in your query in all lower case. That is, use `devicepool` instead.
- For arrays, you can use only arrays of strings. In addition, arrays use a non-standard `item1, item2,...,itemN` format. If the array contains only one element, then it is serialized as `item`, making it indistinguishable from a string field.
You cannot use placeholders to retrieve objects from the context.

Because of these considerations, we recommend that whenever possible, you use the API to access the context in your test logic instead of placeholder strings in test.json and suite.json files. However, in some cases it might be more convenient to use JSONPath placeholders to retrieve single strings to set as environment variables.

Configure settings for test runners

To run custom test suites, test runners must configure their settings based on the test suite that they want to run. Settings are specified based on JSON configuration file templates located in the <device-tester-extract-location>/configs/ folder. If required, test runners must also set up AWS credentials that IDT will use to connect to the AWS cloud.

As a test writer, you will need to configure these files to debug your test suite (p. 734). You must provide instructions to test runners so that they can configure the following settings as needed to run your test suites.

Configure device.json

The device.json file contains information about the devices that tests are run on (for example, IP address, login information, operating system, and CPU architecture).

Test runners can provide this information using the following template device.json file located in the <device-tester-extract-location>/configs/ folder.

```json
[
  {
    "id": "<pool-id>",
    "sku": "<pool-sku>",
    "features": [
      {
        "name": "<feature-name>",
        "value": "<feature-value>",
        "configs": [
          {
            "name": "<config-name>",
            "value": "<config-value>"
          }
        ]
      }
    ],
    "devices": [
      {
        "id": "<device-id>",
        "connectivity": {
          "protocol": "ssh | uart | docker",
          // ssh
          "ip": "<ip-address>",
          "port": <port-number>,
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              // pki
              "privKeyPath": "/path/to/private/key",
              // password
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
]```
Configure settings for test runners

```javascript
// uart
"serialPort": "<serial-port>",

// docker
"containerId": "<container-id>",
"containerUser": "<container-user-name>",
```

All fields that contain values are required as described here:

**id**

A user-defined alphanumeric ID that uniquely identifies a collection of devices called a *device pool*. Devices that belong to a pool must have identical hardware. When you run a suite of tests, devices in the pool are used to parallelize the workload. Multiple devices are used to run different tests.

**sku**

An alphanumeric value that uniquely identifies the device under test. The SKU is used to track qualified devices.

**Note**

If you want to list your board in the AWS Partner Device Catalog, the SKU you specify here must match the SKU that you use in the listing process.

**features**

Optional. An array that contains the device's supported features. Device features are user-defined values that you configure in your test suite. You must provide your test runners with information about the feature names and values to include in the `device.json` file. For example, if you want to test a device that functions as an MQTT server for other devices, then you can configure your test logic to validate specific supported levels for a feature named `MQTT_QOS`. Test runners provide this feature name and set the feature value to the QOS levels supported by their device. You can retrieve the provided information from the IDT context (p. 724) with the `devicePool.features` query, or from the state machine context (p. 711) with the `pool.features` query.

**features.name**

The name of the feature.

**features.value**

The supported feature values.

**features.configs**

Configuration settings, if needed, for the feature.

**features.config.name**

The name of the configuration setting.

**features.config.value**

The supported setting values.

**devices**

An array of devices in the pool to be tested. At least one device is required.

**devices.id**

A user-defined unique identifier for the device being tested.
configure settings for test runners

connectivity.protocol

The communication protocol used to communicate with this device. Each device in a pool must use the same protocol.

Currently, the only supported values are ssh and uart for physical devices, and docker for Docker containers.

connectivity.ip

The IP address of the device being tested.

This property applies only if connectivity.protocol is set to ssh.

connectivity.port

Optional. The port number to use for SSH connections.

The default value is 22.

This property applies only if connectivity.protocol is set to ssh.

connectivity.auth

Authentication information for the connection.

This property applies only if connectivity.protocol is set to ssh.

connectivity.auth.method

The authentication method used to access a device over the given connectivity protocol.

Supported values are:
- pki
- password

connectivity.auth.credentials

The credentials used for authentication.

connectivity.auth.credentials.password

The password used for signing in to the device being tested.

This value applies only if connectivity.auth.method is set to password.

connectivity.auth.credentials.privKeyPath

The full path to the private key used to sign in to the device under test.

This value applies only if connectivity.auth.method is set to pki.

connectivity.auth.credentials.user

The user name for signing in to the device being tested.

connectivity.serialPort

Optional. The serial port to which the device is connected.

This property applies only if connectivity.protocol is set to uart.

connectivity.containerId

The container ID or name of the Docker container being tested.

This property applies only if connectivity.protocol is set to ssh.
**connectivity.containerUser**

Optional. The name of the user to user inside the container. The default value is the user provided in the Dockerfile.

The default value is 22.

This property applies only if `connectivity.protocol` is set to `ssh`.

**Note**

To check if test runners configure the incorrect device connection for a test, you can retrieve `pool.Devices[0].Connectivity.Protocol` from the state machine context and compare it to the expected value in a `Choice` state. If an incorrect protocol is used, then print a message using the `LogMessage` state and transition to the `Fail` state. Alternatively, you can use error handling code to report a test failure for incorrect device types.

**(Optional) Configure userdata.json**

The `userdata.json` file contains any additional information that is required by a test suite but is not specified in the `device.json` file. The format of this file depends on the `userdata_scheme.json` file (p. 703) that is defined in the test suite. If you are a test writer, make sure you provide this information to users who will run the test suites that you write.

**(Optional) Configure resource.json**

The `resource.json` file contains information about any devices that will be used as resource devices. Resource devices are devices that are required to test certain capabilities of a device under test. For example, to test a device's Bluetooth capability, you might use a resource device to test that your device can connect to it successfully. Resource devices are optional, and you can require as many resource devices as you need. As a test writer, you use the `test.json` file (p. 701) to define the resource device features that are required for a test. Test runners then use the `resource.json` file to provide a pool of resource devices that have the required features. Make sure you provide this information to users who will run the test suites that you write.

Test runners can provide this information using the following template `resource.json` file located in the `<device-tester-extract-location>/configs/` folder:

```json
[
  {
    "id": "<pool-id>",
    "features": [
      {
        "name": "<feature-name>",
        "version": "<feature-version>",
        "jobSlots": <job-slots>
      }
    ],
    "devices": [
      {
        "id": "<device-id>",
        "connectivity": {
          "protocol": "ssh | uart | docker",
          "ip": "<ip-address>",
          "port": "<port-number>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>"
            }
          }
        }
      }
    ]
  }
]```
// pki
"privKeyPath": "/path/to/private/key",

// password
"password": "<password>",
},

// uart
"serialPort": "<serial-port>",

// docker
"containerId": "<container-id>",
"containerUser": "<container-user-name>",

}
}
]
]
]

All fields that contain values are required as described here:

id

A user-defined alphanumeric ID that uniquely identifies a collection of devices called a device pool. Devices that belong to a pool must have identical hardware. When you run a suite of tests, devices in the pool are used to parallelize the workload. Multiple devices are used to run different tests.

features

Optional. An array that contains the device's supported features. The information required in this field is defined in the test.json files (p. 701) in the test suite and determines which tests to run and how to run those tests. If the test suite does not require any features, then this field is not required.

features.name

The name of the feature.

features.version

The feature version.

features.jobSlots

Setting to indicate how many tests can concurrently use the device. The default value is 1.

devices

An array of devices in the pool to be tested. At least one device is required.

devices.id

A user-defined unique identifier for the device being tested.

connectivity.protocol

The communication protocol used to communicate with this device. Each device in a pool must use the same protocol.

Currently, the only supported values are ssh and uart for physical devices, and docker for Docker containers.

connectivity.ip

The IP address of the device being tested.

This property applies only if connectivity.protocol is set to ssh.
Configure settings for test runners

connectivity.port

Optional. The port number to use for SSH connections.

The default value is 22.

This property applies only if connectivity.protocol is set to ssh.

connectivity.auth

Authentication information for the connection.

This property applies only if connectivity.protocol is set to ssh.

connectivity.auth.method

The authentication method used to access a device over the given connectivity protocol.

Supported values are:

• pki
• password

connectivity.auth.credentials

The credentials used for authentication.

connectivity.auth.credentials.password

The password used for signing in to the device being tested.

This value applies only if connectivity.auth.method is set to password.

connectivity.auth.credentials.privKeyPath

The full path to the private key used to sign in to the device under test.

This value applies only if connectivity.auth.method is set to pki.

connectivity.auth.credentials.user

The user name for signing in to the device being tested.

connectivity.serialPort

Optional. The serial port to which the device is connected.

This property applies only if connectivity.protocol is set to uart.

connectivity.containerId

The container ID or name of the Docker container being tested.

This property applies only if connectivity.protocol is set to ssh.

connectivity.containerUser

Optional. The name of the user to user inside the container. The default value is the user provided in the Dockerfile.

The default value is 22.

This property applies only if connectivity.protocol is set to ssh.

(Optional) Configure config.json

The config.json file contains configuration information for IDT. Typically, test runners will not need to modify this file except to provide their AWS user credentials for IDT, and optionally, an AWS region.
Configure settings for test runners

If AWS credentials with required permissions are provided AWS IoT Device Tester collects and submits usage metrics to AWS. This is an opt-in feature and is used to improve IDT functionality. For more information, see IDT usage metrics (p. 740).

Test runners can configure their AWS credentials in one of the following ways:

- **Credentials file**

  IDT uses the same credentials file as the AWS CLI. For more information, see Configuration and credential files.

  The location of the credentials file varies, depending on the operating system you are using:
  - macOS, Linux: ~/.aws/credentials
  - Windows: C:\Users\UserName\.aws\credentials

- **Environment variables**

  Environment variables are variables maintained by the operating system and used by system commands. Variables defined during an SSH session are not available after that session is closed. IDT can use the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables to store AWS credentials.

  To set these variables on Linux, macOS, or Unix, use `export`:

  ```
  export AWS_ACCESS_KEY_ID=<your_access_key_id>
  export AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
  ```

  To set these variables on Windows, use `set`:

  ```
  set AWS_ACCESS_KEY_ID=<your_access_key_id>
  set AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
  ```

  To configure AWS credentials for IDT, test runners edit the auth section in the `config.json` file located in the `<device-tester-extract-location>/configs/` folder.

  ```
  {
    "log": {
      "location": "logs"
    },
    "configFiles": {
      "root": "configs",
      "device": "configs/device.json"
    },
    "testPath": "tests",
    "reportPath": "results",
    "awsRegion": "<region>",
    "auth": {
      "method": "file | environment",
      "credentials": {
        "profile": "<profile-name>"
      }
    }
  }
  ```

  All fields that contain values are required as described here:

  **Note**
  All paths in this file are defined relative to the `<device-tester-extract-location>`.
Debug and run custom test suites

After the required configuration (p. 727) is set, IDT can run your test suite. The runtime of the full test suite depends on the hardware and the composition of the test suite. For reference, it takes approximately 30 minutes to complete the full AWS IoT Greengrass qualification test suite on a Raspberry Pi 3B.

As you write your test suite, you can use IDT to run the test suite in debug mode to check your code before you run it or provide it to test runners.

Run IDT in debug mode

Because test suites depend on IDT to interact with devices, provide the context, and receive results, you cannot simply debug your test suites in an IDE without any IDT interaction. To do so, the IDT CLI provides the debug-test-suite command that lets you run IDT in debug mode. Run the following command to view the available options for debug-test-suite:

```bash
devicetester_[linux | mac | win_x86-64] debug-test-suite -h
```

When you run IDT in debug mode, IDT does not actually launch the test suite or run the state machine; instead, it interacts with your IDE to responds to requests made from the test suite running in the IDE and prints the logs to the console. IDT does not time out and waits to exit until manually interrupted. In debug mode, IDT also does not run the state machine and will not generate any report files. To debug your test suite, you must use your IDE to provide some information that IDT usually obtains from the configuration JSON files. Make sure you provide the following information:

- Environment variables and arguments for each test. IDT will not read this information from test.json or suite.json.
To debug your test suites, complete the following steps:

1. Create the setting configuration files that are required to run the test suite. For example, if your test suite requires the `device.json`, `resource.json`, and `user data.json`, make sure you configure all of them as needed.

2. Run the following command to place IDT in debug mode and select any devices that are required to run the test.

   ```bash
   devicetester_[linux | mac | win_x86-64] debug-test-suite [options]
   ```

   After you run this command, IDT waits for requests from the test suite and then responds to them. IDT also generates the environment variables that are required for the case process for the IDT Client SDK.

3. In your IDE, use the `run` or `debug` configuration to do the following:
   
   a. Set the values of the IDT-generated environment variables.
   b. Set the value of any environment variables or arguments that you specified in your `test.json` and `suite.json` file.
   c. Set breakpoints as needed.

4. Run the test suite in your IDE.

   You can debug and re-run the test suite as many times as needed. IDT does not time out in debug mode.

5. After you complete debugging, interrupt IDT to exit debug mode.

### IDT CLI commands to run tests

The following section describes the IDT CLI commands:

**IDT v4.0.0**

- `help`
  
  Lists information about the specified command.

- `list-groups`
  
  Lists the groups in a given test suite.

- `list-suities`
  
  Lists the available test suites.

- `list-supported-products`
  
  Lists the supported products for your version of IDT, in this case AWS IoT Greengrass versions, and AWS IoT Greengrass qualification test suite versions available for the current IDT version.

- `list-test-cases`
  
  Lists the test cases in a given test group. The following option is supported:

  • `group-id`. The test group to search for. This option is required and must specify a single group.

- `run-suite`
  
  Runs a suite of tests on a pool of devices. The following are some commonly used options:
- **suite-id**: The test suite version to run. If not specified, IDT uses the latest version in the tests folder.
- **group-id**: The test groups to run, as a comma-separated list. If not specified, IDT runs all test groups in the test suite.
- **test-id**: The test cases to run, as a comma-separated list. When specified, **group-id** must specify a single group.
- **pool-id**: The device pool to test. Test runners must specify a pool if they have multiple device pools defined in your device.json file.
- **timeout-multiplier**: Configures IDT to modify the test execution timeout specified in the test.json file for a test with a user-defined multiplier.
- **stop-on-first-failure**: Configures IDT to stop execution on the first failure. This option should be used with **group-id** to debug the specified test groups.
- **userdata**: Sets the file that contains user data information required to run the test suite. This is required only if userdataRequired is set to true in the suite.json file for the test suite.

For more information about **run-suite** options, use the **help** option:

```
devicetester_[linux | mac | win_x86-64] run-suite -h
```

devicetester run-suite

devicetester run-suite --debug

devicetester run-suite --debug-suite

devicetester run-suite --help

devicetester run-suite --help

devicetester run-suite --version

devicetester run-suite --version

devicetester run-suite --version

**Review IDT test results and logs**

This section describes the format in which IDT generates console logs and test reports.

**Console message format**

AWS IoT Device Tester uses a standard format for printing messages to the console when it starts a test suite. The following excerpt shows an example of a console message generated by IDT.

```
time="2000-01-02T03:04:05-07:00" level=info msg=Using suite: MyTestSuite_1.0.0 executionId=9a52f362-1227-11eb-86c9-8c8590419f30
```

Most console messages consist of the following fields:

- **time**: A full ISO 8601 timestamp for the logged event.
- **level**: The message level for the logged event. Typically, the logged message level is one of info, warn, or error. IDT issues a fatal or panic message if it encounters an expected event that causes it to exit early.
- **msg**: The logged message.
- **executionId**: A unique ID string for the current IDT process. This ID is used to differentiate between individual IDT runs.
Console messages generated from a test suite provide additional information about the device under test and the test suite, test group, and test cases that IDT runs. The following excerpt shows an example of a console message generated from a test suite.

```plaintext
time="2000-01-02T03:04:05-07:00" level=info msg=Hello world! suiteId=MyTestSuite groupId=myTestGroup testCaseId=myTestCase deviceId=my-device executionId=9a52f362-1227-11eb-86c9-8c8590419f30
```

The test-suite specific part of the console message contains the following fields:

- **suiteId**
  - The name of the test suite currently running.
- **groupId**
  - The ID of the test group currently running.
- **testCaseId**
  - The ID of the test case current running.
- **deviceId**
  - A ID of the device under test that the current test case is using.

To print a test summary to the console when an IDT finishes running a test, you must include a **Report state** (p. 710) in your state machine. The test summary contains information about the test suite, the test results for each group that was run, and the locations of the generated logs and report files. The following example shows a test summary message.

```
========== Test Summary ==========
Execution Time:     5m00s
Tests Completed:    4
Tests Passed:       3
Tests Failed:       1
Tests Skipped:      0
----------------------------------
Test Groups:
  GroupA:         PASSED
  GroupB:         FAILED
----------------------------------
Failed Tests:
  Group Name: GroupB
  Test Name: TestB1
  Reason: Something bad happened
----------------------------------
Path to IoT Device Tester Report: /path/to/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/logs
Path to Aggregated JUnit Report: /path/to/MyTestSuite_Report.xml
```

**AWS IoT Device Tester report schema**

`awsiotdevicetester_report.xml` is a signed report that contains the following information:

- The IDT version.
- The test suite version.
- The report signature and key used to sign the report.
- The device SKU and the device pool name specified in the `device.json` file.
- The product version and the device features that were tested.
• The aggregate summary of test results. This information is the same as that contained in the `suite-name_report.xml` file.

```xml
<apnreport>
    <awsiotdevicetestversion idt-version@awsiotdevicetestversion>
    <testsuiteversion test-suite-version@testsuiteversion>
    <signature>signature</signature>
    <keyname>keyname</keyname>
    <session>
        <testsession execution-id@testsessi>
        <starttime start-time@starttime>
        <endtime end-time@endtime>
    </session>
    <awsproduct>
        <name>product-name</name>
        <version>product-version</version>
        <features>
            <feature name="feature-name" value="supported | not-supported | feature-value" type="optional | required"/>
        </features>
    </awsproduct>
    <device>
        <sku>device-sku</sku>
        <name>device-name</name>
        <features>
            <feature name="feature-name" value="feature-value"/>
        </features>
        <executionMethod>ssh | uart | docker</executionMethod>
    </device>
    <devenvironment>
        <os name="os-name"/>
    </devenvironment>
    <report>
        <suite-name-report-contents>
    </report>
</apnreport>
```

The `awsiotdevicetest_report.xml` file contains an `<awsproduct>` tag that contains information about the product being tested and the product features that were validated after running a suite of tests.

**Attributes used in the `<awsproduct>` tag**

- **name**
  - The name of the product being tested.
- **version**
  - The version of the product being tested.
- **features**
  - The features validated. Features marked as `required` are required for the test suite to validate the device. The following snippet shows how this information appears in the `awsiotdevicetest_report.xml` file.

  ```xml
  <feature name="ssh" value="supported" type="required"/>
  ```

  Features marked as `optional` are not required for validation. The following snippets show optional features.
Test suite report schema

The suite-name_Result.xml report is in JUnit XML format. You can integrate it into continuous integration and deployment platforms like Jenkins, Bamboo, and so on. The report contains an aggregate summary of test results.

```
<testsuites name="suite-name" results time="run-duration" tests="number-of-tests" failures="number-of-tests" skipped="number-of-tests" errors="number-of-tests" disabled="0">
  <testsuite name="test-group-id" package="" tests="number-of-tests" failures="number-of-tests" skipped="number-of-tests" errors="number-of-tests" disabled="0">
    <testcase classname="classname" name="name" time="run-duration"/>
    <failure type="failure-type">
      reason
    </failure>
    <skipped>
      reason
    </skipped>
    <error>
      reason
    </error>
  </testsuite>
</testsuites>
```

The report section in both the awsiotdevicetester_report.xml or suite-name_report.xml lists the tests that were run and the results.

The first XML tag <testsuites> contains the summary of the test execution. For example:

```
<testsuites name="MyTestSuite results" time="2299" tests="28" failures="0" errors="0" disabled="0"/>
```

Attributes used in the <testsuites> tag

- **name**
  - The name of the test suite.

- **time**
  - The time, in seconds, it took to run the test suite.

- **tests**
  - The number of tests executed.
failures
The number of tests that were run, but did not pass.

errors
The number of tests that IDT couldn't execute.

disabled
This attribute is not used and can be ignored.

In the event of test failures or errors, you can identify the test that failed by reviewing the <testsuites> XML tags. The <testsuite> XML tags inside the <testsuites> tag show the test result summary for a test group. For example:

```xml
<testsuite name="combination" package="" tests="1" failures="0" time="161" disabled="0" errors="0" skipped="0">
```

The format is similar to the <testsuites> tag, but with a skipped attribute that is not used and can be ignored. Inside each <testsuite> XML tag, there are <testcase> tags for each executed test for a test group. For example:

```xml
<testcase classname="Security Test" name="IP Change Tests" attempts="1"></testcase>
```

Attributes used in the <testcase> tag

name
The name of the test.

attempts
The number of times IDT executed the test case.

When a test fails or an error occurs, <failure> or <error> tags are added to the <testcase> tag with information for troubleshooting. For example:

```xml
<testcase classname="mcu.Full_MQTT" name="MQTT_TestCase" attempts="1">
  <failure type="Failure">Reason for the test failure</failure>
  <error>Reason for the test execution error</error>
</testcase>
```

IDT usage metrics

If you provide AWS credentials with required permissions, AWS IoT Device Tester collects and submits usage metrics to AWS. This is an opt-in feature and is used to improve IDT functionality. IDT collects information such as the following:

- The AWS account ID used to run IDT
- The IDT CLI commands used to run tests
- The test suite that are run
- The test suites in the <device-tester-extract-location> folder
- The number of devices configured in the device pool
- Test case names and run times
- Test result information, such as whether tests passed, failed, encountered errors, or were skipped
Product features tested
IDT exit behavior, such as unexpected or early exits

All of the information that IDT sends is also logged to a metrics.log file in the <device-tester-extract-location>/results/<execution-id>/ folder. You can view the log file to see the information that was collected during a test run. This file is generated only if you choose to collect usage metrics.

To disable metrics collection, you do not need to take additional action. Simply do not store your AWS credentials, and if you do have stored AWS credentials, do not configure the config.json file to access them.

Configure your AWS credentials

If you do not already have an AWS account, you must create one (p. 741). If you already have an AWS account, you simply need to configure the required permissions (p. 742) for your account that allow IDT to send usage metrics to AWS on your behalf.

Step 1: Create an AWS account

In this step, create and configure an AWS account. If you already have an AWS account, skip to the section called “Step 2: Configure permissions for IDT” (p. 742).

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

2. Follow the online instructions.
   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

To create an administrator user for yourself and add the user to an administrators group (console)

1. Sign in to the IAM console as the account owner by choosing Root user and entering your AWS account email address. On the next page, enter your password.
   Note
   We strongly recommend that you adhere to the best practice of using the Administrator IAM user that follows and securely lock away the root user credentials. Sign in as the root user only to perform a few account and service management tasks.

2. In the navigation pane, choose Users and then choose Add user.
3. For User name, enter Administrator.
4. Select the check box next to AWS Management Console access. Then select Custom password, and then enter your new password in the text box.
5. (Optional) By default, AWS requires the new user to create a new password when first signing in. You can clear the check box next to User must create a new password at next sign-in to allow the new user to reset their password after they sign in.
6. Choose Next: Permissions.
7. Under Set permissions, choose Add user to group.
8. Choose Create group.
9. In the Create group dialog box, for Group name enter Administrators.
10. Choose **Filter policies**, and then select **AWS managed - job function** to filter the table contents.

11. In the policy list, select the check box for **AdministratorAccess**. Then choose **Create group**.

**Note**
You must activate IAM user and role access to Billing before you can use the AdministratorAccess permissions to access the AWS Billing and Cost Management console. To do this, follow the instructions in step 1 of the tutorial about delegating access to the billing console.

12. Back in the list of groups, select the check box for your new group. Choose **Refresh** if necessary to see the group in the list.

13. Choose **Next: Tags**.

14. (Optional) Add metadata to the user by attaching tags as key-value pairs. For more information about using tags in IAM, see **Tagging IAM entities** in the **IAM User Guide**.

15. Choose **Next: Review** to see the list of group memberships to be added to the new user. When you are ready to proceed, choose **Create user**.

You can use this same process to create more groups and users and to give your users access to your AWS account resources. To learn about using policies that restrict user permissions to specific AWS resources, see **Access management** and **Example policies**.

**Step 2: Configure permissions for IDT**

In this step, configure the permissions that IDT uses to run tests and collect IDT usage data. You can use the AWS Management Console or AWS Command Line Interface (AWS CLI) to create an IAM policy and a user for IDT, and then attach policies to the user.

- To Configure Permissions for IDT (Console) (p. 742)
- To Configure Permissions for IDT (AWS CLI) (p. 743)

**To configure permissions for IDT (console)**

Follow these steps to use the console to configure permissions for IDT for AWS IoT Greengrass.

1. Sign in to the **IAM console**.

2. Create a customer managed policy that grants permissions to create roles with specific permissions.
   a. In the navigation pane, choose **Policies**, and then choose **Create policy**.
   b. On the **JSON** tab, replace the placeholder content with the following policy.

   ```json
   {
   "Version": "2012-10-17",
   "Statement": [
   {  
   "Effect": "Allow",
   "Action": [  
   "iot-device-tester:SendMetrics"
   ],
   "Resource": "*"
   }
   
   }  
   }
   ```
   c. Choose **Review policy**.
   d. For **Name**, enter **IDTUsageMetricsIAMPermissions**. Under **Summary**, review the permissions granted by your policy.
   e. Choose **Create policy**.
3. Create an IAM user and attach permissions to the user.
   a. Create an IAM user. Follow steps 1 through 5 in Creating IAM users (console) in the IAM User Guide. If you already created an IAM user, skip to the next step.
   b. Attach the permissions to your IAM user:
      i. On the Set permissions page, choose Attach existing policies to user directly.
      ii. Search for the IDTUsageMetricsIAMPermissions policy that you created in the previous step. Select the check box.
   c. Choose Next: Tags.
   d. Choose Next: Review to view a summary of your choices.
   e. Choose Create user.
   f. To view the user's access keys (access key IDs and secret access keys), choose Show next to the password and access key. To save the access keys, choose Download.csv and save the file to a secure location. You use this information later to configure your AWS credentials file.

To configure permissions for IDT (AWS CLI)

Follow these steps to use the AWS CLI to configure permissions for IDT for AWS IoT Greengrass.

1. On your computer, install and configure the AWS CLI if it's not already installed. Follow the steps in Installing the AWS CLI in the AWS Command Line Interface User Guide.

   Note
   The AWS CLI is an open source tool that you can use to interact with AWS services from your command-line shell.

2. Create the following customer managed policy that grants permissions to manage IDT and AWS IoT Greengrass roles.

   Linux, macOS, or Unix

   ```bash
   aws iam create-policy --policy-name IDTUsageMetricsIAMPermissions --policy-document
   '{
     "Version": "2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Action": [
           "iot-device-tester:SendMetrics"
         ],
         "Resource": "*"
       }
     ]
   }'
   ```

   Windows command prompt

   ```bash
   aws iam create-policy --policy-name IDTUsageMetricsIAMPermissions --policy-document
   ```

   Note
   This step includes a Windows command prompt example because it uses a different JSON syntax than Linux, macOS, or Unix terminal commands.
3. Create an IAM user and attach the permissions required by IDT for AWS IoT Greengrass.
   a. Create an IAM user.
      
      ```bash
      aws iam create-user --user-name user-name
      ```
   b. Attach the IDTUsageMetricsIAMPermissions policy you created to your IAM user. Replace `user-name` with your IAM user name and `<account-id>` in the command with the ID of your AWS account.
      
      ```bash
      aws iam attach-user-policy --user-name user-name --policy-arn arn:aws:iam::<account-id>:policy/IDTGreengrassIAMPermissions
      ```

4. Create a secret access key for the user.
   
   ```bash
   aws iam create-access-key --user-name user-name
   ```

   Store the output in a secure location. You use this information later to configure your AWS credentials file.

**Provide AWS credentials to IDT**

To allow IDT to access your AWS credentials and submit metrics to AWS, do the following:

1. Store the AWS credentials for your IAM user as environment variables or in a credentials file:
   a. To use environment variables, run the following command:
      
      ```bash
      AWS_ACCESS_KEY_ID=access-key
      AWS_SECRET_ACCESS_KEY=secret-access-key
      ```
   b. To use the credentials file, add the following information to the .aws/credentials file:
      
      ```
      [profile-name]
      aws_access_key_id=access-key
      aws_secret_access_key=secret-access-key
      ```

2. Configure the auth section of the config.json file. For more information, see (Optional) Configure config.json (p. 732).

**Troubleshooting IDT for AWS IoT Greengrass V2**

IDT for AWS IoT Greengrass V2 writes errors to various locations based on the type of errors. IDT writes errors to the console, log files, and test reports.

**Where to look for errors**

High-level errors display on the console while the test is running, and a summary of the failed tests displays when all tests are complete. `awsiotdevicetester_report.xml` contains a summary of all the errors that caused a test to fail. IDT stores the log files for each test run in a directory named with a UUID for the test execution, displayed on the console during the test run.

The IDT test logs directory is `<device-tester-extract-location>/results/<execution-id>/logs/`. This directory contains the following files, which are useful for debugging.
Resolving IDT for AWS IoT Greengrass V2 errors

Before you run IDT for AWS IoT Greengrass, get the correct configuration files in place. If you receive parsing and configuration errors, your first step is to locate and use a configuration template appropriate for your environment.

If you are still having issues, see the following debugging process.

Topics

- Command not found errors while testing (p. 746)
- Conflict errors (p. 746)
- Could not start test error (p. 746)
- Docker qualification image exists errors (p. 746)
- Machine learning qualification errors (p. 747)
- Parsing errors (p. 747)
- Permission denied errors (p. 747)
- Qualification report generation error (p. 747)
- Required parameter missing error (p. 747)
- Security exception on macOS (p. 748)
- SSH connection errors (p. 748)
- Stream manager qualification errors (p. 748)
• Throttling errors during Docker qualification (p. 748)
• Timeout errors (p. 749)
• Version check errors (p. 749)

**Command not found errors while testing**

You need an older version of the OpenSSL library (libssl1.0.0) to run tests on AWS IoT Greengrass devices. Most current Linux distributions use libssl version 1.0.2 or later (v1.1.0).

For example, on a Raspberry Pi, run the following commands to install the required version of libssl:

1. `wget http://ftp.us.debian.org/debian/pool/main/o/openssl/libssl1.0.0_1.0.2l-1-bpo8+1_armhf.deb`
2. `sudo dpkg -i libssl1.0.0_1.0.2l-1-bpo8+1_armhf.deb`

**Conflict errors**

You might see the following error when you run the AWS IoT Greengrass qualification suite concurrently on more than one device.

```plaintext
```

Concurrent test execution isn't yet supported for the AWS IoT Greengrass qualification suite. Run the qualification suite sequentially for each device.

**Could not start test error**

You might encounter errors that point to failures that occurred when the test was attempting to start. There are several possible causes, so do the following:

• Make sure that the pool name in your execution command actually exists. IDT references the pool name directly from your device.json file.
• Make sure that the devices in your pool have correct configuration parameters.

**Docker qualification image exists errors**

The Docker application manager qualification tests use the `amazon/amazon-ec2-metadata-mock` container image in Docker Hub to qualify the device under test.

You might receive the following error if the image is already present in a Docker container on the device under test.

```plaintext
The Docker image amazon/amazon-ec2-metadata-mock:version already exists on the device.
```

If you previously downloaded this image and ran the `amazon/amazon-ec2-metadata-mock` container on your device, make sure you remove this image from the device under test before you run the qualification tests.
Machine learning qualification errors

When you run machine learning (ML) qualification tests, you might encounter qualification failures if your device doesn’t meet the requirements to deploy the AWS-provided ML components. To troubleshoot ML qualification errors, do the following:

- Look for error details in the component logs for the components that were deployed during the test run. Component logs are located in the `<device-tester-extract-location>/results/<execution-id>/logs/<test-group-id>` directory.
- Add the `-Dgg.persist=installed.software` argument to the `test.json` file for the failing test case. The `test.json` file is located in the `<device-tester-extract-location>/tests/GGV2Q_version` directory.

Parsing errors

Typos in a JSON configuration can lead to parsing errors. Most of the time, the issue is a result of omitting a bracket, comma, or quotation mark from your JSON file. IDT performs JSON validation and prints debugging information. It prints the line where the error occurred, the line number, and the column number of the syntax error. This information should be enough to help you fix the error, but if you still can’t locate the error, you can perform validation manually in your IDE, a text editor such as Atom or Sublime, or through an online tool like JSONLint.

Permission denied errors

IDT performs operations on various directories and files in a device under test. Some of these operations require root access. To automate these operations, IDT must be able to run commands with sudo without typing a password.

Follow these steps to allow sudo access without typing a password.

**Note**

user and username refer to the SSH user used by IDT to access the device under test.

1. Use `sudo usermod -aG sudo <ssh-username>` to add your SSH user to the sudo group.
2. Sign out and then sign in for changes to take effect.
3. Open `/etc/sudoers` file and add the following line to the end of the file: `<ssh-username>
   ALL=(ALL) NOPASSWD: ALL

**Note**

As a best practice, we recommend that you use `sudo visudo` when you edit `/etc/sudoers`.

Qualification report generation error

IDT supports the four latest `major.minor` versions of the AWS IoT Greengrass V2 qualification suite (GGV2Q) to generate qualification reports that you can submit to AWS Partner Network to include your devices in the AWS Partner Device Catalog. Earlier versions of the qualification suite don’t generate qualification reports.

If you have questions about the support policy, contact AWS Support at [https://aws.amazon.com/contact-us/](https://aws.amazon.com/contact-us/).

Required parameter missing error

When IDT adds new features, it might introduce changes to the configuration files. Using an old configuration file might break your configuration. If this happens, the `<test_case_id>.log` file under
Resolving IDT for AWS IoT Greengrass V2 errors

/\results/<execution-id>/logs explicitly lists all missing parameters. IDT also validates your JSON configuration file schemas to verify that you are using the latest supported version.

Security exception on macOS

When you run IDT on a host computer that uses macOS 10.15, the notarization ticket for IDT isn't detected, which blocks IDT from being run. To run IDT, grant a security exception to the devicetester_mac_x86-64 executable. Do one of the following:

To grant a security exception to IDT executables

1. Launch System Preferences from the Apple menu.
2. Choose Security & Privacy, then on the General tab, choose the lock icon to make changes to security settings.
3. Look for the message "devicetester_mac_x86-64" was blocked from use because it is not from an identified developer. and choose Allow Anyway.
4. Accept the security warning.

SSH connection errors

When IDT can't connect to a device under test, it logs connection failures in /\results/<execution-id>/logs/<test-case-id>.log. SSH messages appear at the top of this log file because connecting to a device under test is one of the first operations that IDT performs.

Most Windows configurations use the PuTTy terminal application to connect to Linux hosts. This application requires that you convert standard PEM private key files into a proprietary Windows format called PPK. If you configure SSH in your device.json file, use PEM files. If you use a PPK file, IDT can't create an SSH connection with the AWS IoT Greengrass device and can't run tests.

Stream manager qualification errors

When you run stream manager qualification tests, you might see the following error in the com.aws.StreamManagerExport.log file.

Failed to upload data to S3

This error can occur when stream manager uses the AWS credentials in the ~/.aws/credentials file on your device instead of using the environment credentials that IDT exports to the device under test. To prevent this issue, delete the credentials file on your device, and re-run the qualification test.

Throttling errors during Docker qualification

Docker Hub limits the number of pull requests that anonymous and Free Docker Hub users can make. When you run IDT tests for Docker qualification, you might receive one of the following errors if you exceed the rate limits for anonymous or free user pull requests:

ERROR: toomanyrequests: Too Many Requests.

You have reached your pull rate limit.

To resolve these errors, you can wait for a few hours before you run the qualification test. If you plan on consistently running a large number of tests, which can result in submitting a large number of pull...
requests, see the Docker Hub website for information about rate limits, and options for authenticating and upgrading your Docker account.

**Timeout errors**

You can increase the timeout for each test by specifying a timeout multiplier applied to the default value of each test's timeout. Any value configured for this flag must be greater than or equal to 1.0.

To use the timeout multiplier, use the flag `--timeout-multiplier` when running the tests. For example:

```bash
./devicetester_linux run-suite --suite-id GGV2Q_1.0.0 --pool-id DevicePool1 --timeout-multiplier 2.5
```

For more information, run `run-suite --help`.

Some timeout errors occur when IDT test cases can't be completed because of configuration issues. You can't resolve these errors by increasing the timeout multiplier. Use the logs from the test run to troubleshoot the underlying configuration issues.

- If the MQTT or Lambda component logs contain `Access denied` errors, your Greengrass installation folder might not have the correct file permissions. Run the following command for each folder in the installation path that you defined in your `userdata.json` file.

  ```bash
  sudo chmod 755 folder-name
  ```

- If the Greengrass logs indicate that the Greengrass CLI deployment isn't complete, do the following:
  - Verify that `bash` is installed on the device under test.
  - If your `userdata.json` file includes the `GreengrassCliVersion` configuration parameter, remove it. This parameter is deprecated in IDT v4.1.0 and later versions. For more information, see Configure `userdata.json` (p. 682).

**Version check errors**

IDT issues the following error when the AWS user credentials for the IDT user don't have the required IAM permissions.

```bash
Failed to check version compatibility
```

The AWS user that doesn't have the required IAM permissions.

---

**Support policy for AWS IoT Device Tester for AWS IoT Greengrass**

AWS IoT Device Tester for AWS IoT Greengrass is a test automation tool used to validate and qualify your AWS IoT Greengrass devices for inclusion in the AWS Partner Device Catalog. We recommend that you use the most recent version of AWS IoT Greengrass and AWS IoT Device Tester to test or qualify your devices.

At least one version of AWS IoT Device Tester is available for each supported version of AWS IoT Greengrass. For supported versions of AWS IoT Greengrass, see Greengrass nucleus versions (p. 137). For
supported versions of AWS IoT Device Tester, see Supported versions of AWS IoT Device Tester for AWS IoT Greengrass V2 (p. 661).

For each version of AWS IoT Device Tester, the three latest test suite versions are supported for qualification of devices.

You can also use any of the supported versions of AWS IoT Greengrass and AWS IoT Device Tester to test or qualify your devices. Although you can continue to use unsupported versions of AWS IoT Device Tester, those versions do not receive bug fixes or updates. If you have questions about the support policy, contact AWS Support.
Troubleshooting AWS IoT Greengrass V2

Use the troubleshooting information and solutions in this section to help resolve issues with AWS IoT Greengrass Version 2.

Topics
- View AWS IoT Greengrass Core software logs (p. 751)
- View component logs (p. 751)
- AWS IoT Greengrass Core software issues (p. 752)
- AWS IoT Greengrass cloud issues (p. 753)
- Core device deployment issues (p. 753)
- Core device component issues (p. 757)
- AWS Command Line Interface issues (p. 757)

View AWS IoT Greengrass Core software logs

The AWS IoT Greengrass Core software log file provides real-time information about the core device. This can help you identify issues with components and deployments.

This log file also includes logs for plugin components, such as log manager (p. 206) and secret manager (p. 285). For more information, see Component types (p. 323).

To view the core device log file
- Run the following command to view the log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

View component logs

Component log files provide real-time information about a component that runs on the Greengrass core device. This can help you identify issues with components.

Note
The AWS IoT Greengrass Core software doesn't create component log files for plugin components, such as log manager (p. 206) and secret manager (p. 285). For more information, see Component types (p. 323). For more information about how to view plugin component logs, see View AWS IoT Greengrass Core software logs (p. 751).

To view the log file for a component
- Run the following command to view the log file in real time. Replace `/greengrass/v2` with the path to the AWS IoT Greengrass root folder, and replace `com.example.HelloWorld` with the name of the component.

```
sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
```
AWS IoT Greengrass Core software issues

Troubleshoot AWS IoT Greengrass Core software issues.

Topics
- Unable to set up core device (p. 752)
- Unable to connect to AWS IoT Core (p. 752)
- Out of memory error (p. 752)
- Unable to install Greengrass CLI (p. 752)
- User root is not allowed to execute (p. 753)
- Failed to map segment from shared object: operation not permitted (p. 753)

Unable to set up core device

If the AWS IoT Greengrass Core software installer fails and you aren't able to set up a core device, you might need to uninstall the software and try again. For more information, see Uninstall the AWS IoT Greengrass Core software (p. 131).

Unable to connect to AWS IoT Core

You might see this error when the AWS IoT Greengrass Core software can't connect to AWS IoT Core to retrieve deployment jobs, for example. Do the following:

- Check that your core device can connect to the internet and AWS IoT Core. For more information about the AWS IoT Core endpoint to which your device connects, see Configure the AWS IoT Greengrass Core software (p. 119).
- Check that your core device's AWS IoT thing uses a certificate that allows the `iot:Connect`, `iot:Publish`, `iot:Receive`, and `iot:Subscribe` permissions.
- If your core device uses a network proxy (p. 127), check that your core device has a device role (p. 627) and that its role allows the `iot:Connect`, `iot:Publish`, `iot:Receive`, and `iot:Subscribe` permissions.

Out of memory error

This error typically occurs if your device doesn't have sufficient memory to allocate an object in the Java heap. On devices with limited memory, you might need to specify a maximum heap size to control memory allocation. For more information, see Control memory allocation with JVM options (p. 121).

Unable to install Greengrass CLI

You might see the following console message when you use the `--deploy-dev-tools` argument in your installation command for AWS IoT Greengrass Core.

```
Thing group exists, it could have existing deployment and devices, hence NOT creating deployment for Greengrass first party dev tools, please manually create a deployment if you wish to
```

This occurs when the Greengrass CLI component is not installed because your core device is a member of a thing group that has an existing deployment. If you see this message, you can manually deploy the
Greengrass CLI component (aws.greengrass.Cli) to the device to install the Greengrass CLI. For more information, see Install the Greengrass CLI (p. 595).

User root is not allowed to execute

You might see this error when the user that runs the AWS IoT Greengrass Core software (typically root) doesn't have permission to run sudo with any user and any group. For the default ggc_user system user, this error looks like the following:

```
Sorry, user root is not allowed to execute <command> as ggc_user:ggc_group.
```

Check that your /etc/sudoers file gives the user permission to run sudo as other groups. The permission for the user in /etc/sudoers should look like the following example.

```
root    ALL=(ALL:ALL) ALL
```

Failed to map segment from shared object: operation not permitted

This error typically occurs when the AWS IoT Greengrass Core software fails to start because the /tmp directory is mounted with noexec permissions.

Run the following command to remount the /tmp directory with exec permissions and try again.

```
sudo mount -o remount,exec /tmp
```

AWS IoT Greengrass cloud issues

Use the following information to troubleshoot issues with the AWS IoT Greengrass console and API. Each entry corresponds to an error message that you might see when you perform an action.

An error occurred (AccessDeniedException) when calling the CreateComponentVersion operation: User: arn:aws:iam::123456789012:user/<username> is not authorized to perform: null

You might see this error when you create a component version from the AWS IoT Greengrass console or with the CreateComponentVersion operation.

This error indicates that your recipe isn't valid JSON or YAML. Check the syntax of your recipe, fix any syntax issues, and try again. You can use an online JSON or YAML syntax checker to identify syntax issues in your recipe.

Core device deployment issues

Troubleshoot deployment issues on Greengrass core devices. Each entry corresponds to a log message that you might see on your core device.
Error:
com.aws.greengrass.componentmanager.exceptions.PackageDownloadException:
Failed to download artifact

You might see this error when the AWS IoT Greengrass Core software fails to download a component artifact when the core device applies a deployment. The deployment fails as a result of this error.

When you receive this error, the log also includes a stack trace that you can use to identify the specific issue. Each of the following entries corresponds to a message that you might see in the stack trace of the Failed to download artifact error message.

software.amazon.awssdk.services.s3.model.S3Exception: null
(Service: S3, Status Code: 403, Request ID: null, ...)

The PackageDownloadException error (p. 754) might include this stack trace in the following cases:

• The component artifact isn't available at the Amazon S3 URL that you specify in the component's recipe. Check that you uploaded the artifact to the S3 bucket and that the artifact URL matches the Amazon S3 URL of the artifact in the bucket.

• The core device role (p. 627) doesn't allow the AWS IoT Greengrass Core software to download the component artifact from the Amazon S3 URL that you specify in the component's recipe. Check that the device role allows s3:GetObject for the Amazon S3 URL where the artifact is available.

Error:
com.aws.greengrass.componentmanager.exceptions.ArtifactChecksumMismatchException:
Integrity check for downloaded artifact failed. Probably due to file corruption.

You might see this error when the AWS IoT Greengrass Core software fails to download a component artifact when the core device applies a deployment. The deployment fails because the downloaded
Error:
com.aws.greengrass.componentmanager.exceptions.NoAvailableComponentVersionException:
Failed to negotiate component <name> version with cloud and no local applicable version satisfying requirement <requirements>

Do the following:

• Check if the artifact file changed in the S3 bucket where you host it. If the file changed since you created the component, restore it to the previous version that the core device expects. If you can't restore the file to its previous version, or if you want to use the new version of the file, create a new version of the component with the artifact file.

• Check your core device's internet connection. This error can occur if the artifact file becomes corrupted while it downloads. Create a new deployment and try again.

Error:
com.aws.greengrass.componentmanager.exceptions.NoAvailableComponentVersionException:
Failed to negotiate component <name> version with cloud and no local applicable version satisfying requirement <requirements>

You might see this error when a core device can't find a component version that meets the requirements of the deployments for that core device. The core device checks for the component in the AWS IoT Greengrass service and on the local device. The error message includes each deployment's target and that deployment's version requirements for the component. The deployment target can be a thing, a thing group, or LOCAL_DEPLOYMENT, which represents the local deployment on the core device.

This issue can occur in the following cases:

• The core device is the target of multiple deployments that have conflicting component version requirements. For example, the core device might be the target of multiple deployments that include a com.example.HelloWorld component, where one deployment requires version 1.0.0 and the other requires version 1.0.1. It's impossible to have a component that meets both requirements, so the deployment fails.

• The component version doesn't exist in the AWS IoT Greengrass service or on the local device. The component might have been deleted, for example.

• There exists component versions that meet the version requirements, but none are compatible with the core device's platform.

To resolve this issue, revise the deployments to include compatible component versions or remove incompatible ones. For more information about how to revise cloud deployments, see Revise deployments (p. 393). For more information about how to revise local deployments, see the AWS IoT Greengrass CLI deployment create (p. 600) command.

software.amazon.awssdk.services.secretsmanager.model.SecretsManagerException:
User: <user> is not authorized to perform: secretsmanager:GetSecretValue on resource: <arn>

This error can occur when you use the secret manager component (p. 285) to deploy an AWS Secrets Manager secret. If the core device's token exchange IAM role (p. 627) doesn't grant permission to get the secret, the deployment fails and the Greengrass logs include this error.
To authorize a core device to download a secret:

1. Add the `secretsmanager:GetSecretValue` permission to the core device's token exchange role. The following example policy statement grants permission to get the value of a secret.

   ```json
   {
     "Effect": "Allow",
     "Action": ["secretsmanager:GetSecretValue"],
   }
   ```

   For more information, see Authorize core devices to interact with AWS services (p. 627).

2. Reapply the deployment to the core device. Do one of the following:
   - Revise the deployment without any changes. The core device tries to download the secret again when it receives the revised deployment. For more information, see Revise deployments (p. 393).
   - Restart the AWS IoT Greengrass Core software to retry the deployment. For more information, see Run the AWS IoT Greengrass Core software (p. 91)

   The deployment succeeds if secret manager downloads the secret successfully.

Info:
com.aws.greengrass.deployment.exceptions.RetryableDeploymentDocumentDownloadException:
Greengrass Cloud Service returned an error when getting full deployment configuration

You might see this error when the core device receives a large deployment document, which is a deployment document larger than 7 KB (for deployments that target things) or 31 KB (for deployments that target thing groups). To retrieve a large deployment document, a core device's AWS IoT policy must allow the `greengrass:GetDeploymentConfiguration` permission. This error can occur when the core device doesn't have this permission. When this error occurs, the deployment retries indefinitely, and its status is **In progress** (IN_PROGRESS).

To resolve this issue, add the `greengrass:GetDeploymentConfiguration` permission to the core device's AWS IoT policy. For more information, see Update a core device's AWS IoT policy (p. 613).

Info:
com.aws.greengrass.deployment.DeploymentDocumentDownloader:
Calling Greengrass cloud to get full deployment configuration

You might see this information message printed multiple times without an error, because the core device logs the error at the DEBUG log level. This issue can occur when the core device receives a large deployment document. When this issue occurs, the deployment retries indefinitely, and its status is **In**
Core device component issues

Troubleshoot Greengrass component issues on core devices.

Topics

- Python script doesn't log messages (p. 757)

Python script doesn't log messages

Greengrass core devices collect logs that you can use to identify issues with components. If your Python script's stdout and stderr messages don't appear in your component logs, you might need to flush the buffer or disable buffering for these standard output streams in Python. Do any of the following:

- Run Python with the `-u` argument to disable buffering on stdout and stderr.

```
python3 -u hello_world.py
```

- Use `Setenv` (p. 346) in your component's recipe to set the `PYTHONUNBUFFERED` environment variable to a non-empty string. This environment variable disables buffering on stdout and stderr.

- Flush the buffer for the stdout or stderr streams. Do one of the following:
  - Flush a message when you print.

```
import sys
print('Hello, error!', file=sys.stderr, flush=True)
```

  - Flush a message after you print. You can send multiple messages before you flush the stream.

```
import sys
print('Hello, error!', file=sys.stderr)
sys.stderr.flush()
```

For more information about how to verify that your Python script outputs log messages, see View component logs (p. 751).

AWS Command Line Interface issues

Troubleshoot AWS CLI issues for AWS IoT Greengrass V2.

Topics

- Error: Invalid choice: 'greengrassv2' (p. 757)

Error: Invalid choice: 'greengrassv2'

You might see this error when you run an AWS IoT Greengrass V2 command with the AWS CLI (for example, `aws greengrassv2 list-core-devices`).
This error indicates that you have a version of the AWS CLI that doesn't support AWS IoT Greengrass V2. To use AWS IoT Greengrass V2 with the AWS CLI, you must have one of the following versions or later:

- Minimum AWS CLI V1 version: v1.18.197
- Minimum AWS CLI V2 version: v2.1.11

**Tip**
You can run the following command to check the version of the AWS CLI that you have.

```
aws --version
```

To resolve this issue, update the AWS CLI to a later version that supports AWS IoT Greengrass V2. For more information, see Installing, updating, and uninstalling the AWS CLI in the *AWS Command Line Interface User Guide*. 
Tag your AWS IoT Greengrass Version 2 resources

With tags, you can organize and manage your resources in AWS IoT Greengrass. You can use tags to assign metadata to your resources, and you can use tags in IAM policies to define conditional access to your resources.

Using tags in AWS IoT Greengrass V2

You can use tags to categorize your AWS IoT Greengrass resources by purpose, owner, environment, or any other classification for your use case. When you have many resources of the same type, tags help you more readily identify a specific resource.

Each tag consists of a key and an optional value, both of which you define. For example, you could define a set of tags for your core devices that helps you track them by the customers who own the devices. We recommend that you create a set of tag keys that meets your needs for each kind of resource. By using a consistent set of tag keys, you can more easily manage your resources.

Tag with the AWS Management Console

The Tag Editor in the AWS Management Console provides a central, unified way for you to create and manage your tags for resources from all AWS services. For more information, see Tag Editor in the AWS Resource Groups User Guide.

Tag with the AWS IoT Greengrass V2 API

You can also use the AWS IoT Greengrass V2 API to work with tags. Before you create tags, be aware of tagging restrictions. For more information, see Tag naming and usage conventions in the AWS General Reference.

- To add tags when you create a resource, define them in the tags property of the resource.
- To add tags to an existing resource, or to update tag values, use the TagResource operation.
- To remove tags from a resource, use the UntagResource operation.
- To retrieve the tags that are associated with a resource, use the ListTagsForResource operation, or describe the resource and inspect its tags property.

The following table lists resources that you can tag using the AWS IoT Greengrass V2 API and their corresponding Create and Describe or Get operations.

Taggable AWS IoT Greengrass V2 resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Create operation</th>
<th>Describe or get operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core device</td>
<td>None. Run the AWS IoT Greengrass Core software on a device to create a core device.</td>
<td>GetCoreDevice</td>
</tr>
</tbody>
</table>
### Using tags with IAM policies

In your IAM policies, you can use resource tags to control user access and permissions. For example, policies can allow users to create only those resources that have a specific tag. Policies can also restrict users from creating or modifying resources that have certain tags.

**Note**

If you use tags to allow or deny users' access to resources, you should deny users the ability to add or remove those tags for the same resources. Otherwise, a user could circumvent your restrictions and gain access to a resource by modifying its tags.

You can use the following condition context keys and values in the `Condition` element, also called the **Condition** block, of a policy statement.

- `greengrassv2:ResourceTag/tag-key:  tag-value`
  
  Allow or deny actions on resources with specific tags.

- `aws:RequestTag/tag-key:  tag-value`
  
  Require that a specific tag be used, or not used, when creating or modifying a taggable resource.

- `aws:TagKeys:  [tag-key,  ...]`
  
  Require that a specific set of tag keys be used, or not used, when creating or modifying a taggable resource.

**Note**

The condition context keys and values in an IAM policy apply only to actions that have a taggable resource as a required parameter. For example, you can set tag-based conditional access for `ListCoreDevices`.

For more information, see [Controlling access to AWS resources using resource tags](https://docs.aws.amazon.com/IAM/latest/UserGuide/access_tag-based.html) and [IAM JSON policy reference](https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements.html) in the IAM User Guide.
Open source AWS IoT Greengrass Core software

The AWS IoT Greengrass Version 2 edge runtime (nucleus) and other components of the AWS IoT Greengrass Core software are open source. This means that you can review the code to troubleshoot interactions with your applications. You can also customize and extend the AWS IoT Greengrass Core software to meet your specific software and hardware needs.

For information about the open source repositories for the AWS IoT Greengrass Core software, see the aws-greengrass organization on GitHub. Your use of open source software is governed by the open source license in the corresponding GitHub repository.

Your use of the AWS IoT Greengrass Core software and components not subject to an open source license is governed by the AWS Greengrass Core Software License.
Document history for the AWS IoT Greengrass V2 Developer Guide

The following table describes the documentation for this release of AWS IoT Greengrass Version 2.

- **API version**: 2020-11-30

<table>
<thead>
<tr>
<th>update-history-change</th>
<th>update-history-description</th>
<th>update-history-date</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS IoT Device Tester v4.2.0 with GGV2Q v2.0.1 released</td>
<td>Version 4.2.0 of IDT for AWS IoT Greengrass V2 has been updated with AWS IoT Greengrass V2 qualification suite (GGV2Q) v2.0.1. This release supports Greengrass nucleus version 2.4.0 for device qualification.</td>
<td>August 31, 2021</td>
</tr>
<tr>
<td>Updated machine learning installer components</td>
<td>DLR installer component v1.6.5 and TensorFlow Lite installer component v2.5.4 are available. These component versions include the new <code>UseInstaller</code> configuration parameter that lets you disable the default installation script.</td>
<td>August 30, 2021</td>
</tr>
<tr>
<td>Embedded Linux support for AWS IoT Greengrass</td>
<td>The BitBake recipe for AWS IoT Greengrass V2 is available in the <code>meta-aws</code> project on GitHub. You can use this recipe to build a custom Linux-based operating system using the Yocto Project.</td>
<td>August 20, 2021</td>
</tr>
<tr>
<td>Code integrity</td>
<td>Added information about how AWS IoT Greengrass V2 verifies the integrity of software that Greengrass core devices download from the AWS Cloud.</td>
<td>August 19, 2021</td>
</tr>
<tr>
<td>VPC endpoints (AWS PrivateLink)</td>
<td>AWS IoT Greengrass now supports interface VPC endpoints (AWS PrivateLink) for the AWS IoT Greengrass control plane. You can establish a private connection between your VPC and the AWS IoT Greengrass control plane.</td>
<td>August 16, 2021</td>
</tr>
<tr>
<td>Stream manager v2.0.12 released</td>
<td>Stream manager v2.0.12 is now available. This releases fixes an issue that prevented upgrades from version 2.0.7 of the stream</td>
<td>August 10, 2021</td>
</tr>
<tr>
<td>Release Date</td>
<td>Event Description</td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>August 9, 2021</td>
<td>The Dockerfile and Docker image for AWS IoT Greengrass v2.4.0 are now available.</td>
<td></td>
</tr>
<tr>
<td>August 3, 2021</td>
<td>This release provides version 2.4.0 of the Greengrass nucleus component and updates AWS-provided components. It includes support for component system resource limits, IPC operations to pause and resume components, and provisioning plugins.</td>
<td></td>
</tr>
<tr>
<td>July 29, 2021</td>
<td>Added the following AWS-provided components for AWS IoT SiteWise: IoT SiteWise OPC-UA collector, IoT SiteWise publisher, and IoT SiteWise processor.</td>
<td></td>
</tr>
<tr>
<td>July 14, 2021</td>
<td>Version 4.2.0 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualification suite (GGV2Q) v2.0.0 and includes support for optional qualification tests for Docker components, machine learning, and stream manager.</td>
<td></td>
</tr>
<tr>
<td>July 14, 2021</td>
<td>Version 1.13.0 of the AWS IoT Device SDK for C++ v2 supports AWS IoT Greengrass Core IPC, so you can develop components in C++ that interact with the AWS IoT Greengrass Core software.</td>
<td></td>
</tr>
<tr>
<td>July 12, 2021</td>
<td>Version 1.0.2 of the Amazon SageMaker Edge Manager component is available. This version updates the installation script in the component lifecycle. Your core devices must now have Python 3.6 or later, including pip for your version of Python, installed on the device before you deploy this component.</td>
<td></td>
</tr>
<tr>
<td>July 8, 2021</td>
<td>IDT for AWS IoT Greengrass V2 version 4.1.0 now supports using Greengrass nucleus version 2.3.0 for device qualification.</td>
<td></td>
</tr>
<tr>
<td>Release</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td><strong>Dockerfile and Docker images for AWS IoT Greengrass Core software v2.3.0</strong></td>
<td>The Dockerfile and Docker image for AWS IoT Greengrass v2.3.0 are now available.</td>
<td></td>
</tr>
<tr>
<td><strong>AWS managed policies</strong></td>
<td>Added information about AWS managed policies for AWS IoT Greengrass.</td>
<td></td>
</tr>
<tr>
<td><strong>New recommended JVM options</strong></td>
<td>Added information about recommended JVM options to control memory allocation for AWS IoT Greengrass Core software.</td>
<td></td>
</tr>
<tr>
<td><strong>AWS IoT Greengrass Core v2.3.0 software update</strong></td>
<td>This release provides version 2.3.0 of the Greengrass nucleus component and updates AWS-provided components. It includes support for large component configuration documents in deployments.</td>
<td></td>
</tr>
<tr>
<td><strong>Dockerfile and Docker images for AWS IoT Greengrass Core software v2.2.0</strong></td>
<td>The Dockerfile and Docker image for AWS IoT Greengrass v2.2.0 are now available.</td>
<td></td>
</tr>
<tr>
<td><strong>AWS IoT Device Tester v4.1.0 with GGV2Q v1.1.1 released</strong></td>
<td>Version 4.1.0 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualification suite (GGV2Q) v1.1.1 and supports using Greengrass nucleus v2.2.0, v2.1.0, and v2.0.5 for device qualification.</td>
<td></td>
</tr>
<tr>
<td><strong>AWS IoT Greengrass Core v2.2.0 software update</strong></td>
<td>This release provides version 2.2.0 of the Greengrass nucleus component and updates AWS-provided components. It includes components that you can deploy to add support for client devices and add the local shadow service.</td>
<td></td>
</tr>
<tr>
<td><strong>Lambda launcher v2.0.6 released</strong></td>
<td>Version 2.0.6 of the Lambda launcher component is available. This version includes performance improvements and bug fixes.</td>
<td></td>
</tr>
<tr>
<td><strong>New SageMaker Edge Manager component released</strong></td>
<td>Version 1.0.0 of the Amazon SageMaker Edge Manager component is available for AWS IoT Greengrass. This component installs the SageMaker Edge Manager agent binary on Greengrass core devices.</td>
<td></td>
</tr>
<tr>
<td>Component types</td>
<td>Added information about component types in AWS IoT Greengrass. The component type specifies how the AWS IoT Greengrass Core software runs a component.</td>
<td>June 3, 2021</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>AWS IoT Device Tester v4.0.2 with GGV2Q v1.1.0 released</td>
<td>Version 4.0.2 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualification suite (GGV2Q) v1.1.0 and supports using Greengrass nucleus v2.1.0 with Greengrass CLI v2.1.0 for device qualification. This also includes new required test groups for MQTT and Lambda, and other minor bug fixes and improvements.</td>
<td>May 5, 2021</td>
</tr>
<tr>
<td>Dockerfile and Docker images for AWS IoT Greengrass Core software v2.1.0</td>
<td>The Dockerfile and Docker image for AWS IoT Greengrass v2.1.0 are now available. The Docker image enables you to run AWS IoT Greengrass in a Docker container that uses Amazon Linux 2 as the base operating system.</td>
<td>April 27, 2021</td>
</tr>
<tr>
<td>AWS IoT Greengrass Core v2.1.0 software update</td>
<td>This release provides version 2.1.0 of the Greengrass nucleus component and updates AWS-provided components. It includes a new component that you can use to download Docker images from private Amazon ECR repositories, and new sample components to perform machine learning inference using TensorFlow Lite.</td>
<td>April 26, 2021</td>
</tr>
<tr>
<td>Example component that uses Secrets Manager</td>
<td>Added an example component that prints the value of an AWS Secrets Manager secret that you deploy to a core device.</td>
<td>April 8, 2021</td>
</tr>
<tr>
<td>Minimal AWS IoT policy for Greengrass core devices</td>
<td>Added information about the minimal set of permissions required to support basic Greengrass functionality on a core device.</td>
<td>April 2, 2021</td>
</tr>
<tr>
<td>Subscribe to IPC event streams</td>
<td>Added information about how to use interprocess communication (IPC) operations to subscribe to streams of events on a Greengrass core device.</td>
<td>April 1, 2021</td>
</tr>
<tr>
<td>Support update for AWS IoT Device Tester for AWS IoT Greengrass</td>
<td>IDT for AWS IoT Greengrass V2 version 4.0.1 now supports using Greengrass nucleus version 2.0.5 with Greengrass CLI version 2.0.5 for device qualification.</td>
<td>March 17, 2021</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Create custom components that use stream manager</td>
<td>Added information about how to configure component recipes and artifacts to develop applications that manage data streams.</td>
<td>March 9, 2021</td>
</tr>
<tr>
<td>AWS IoT Greengrass Core v2.0.5 software update</td>
<td>This release provides version 2.0.5 of the Greengrass nucleus component and updates AWS-provided components. It fixes an issue with network proxy support and an issue with the Greengrass data plane endpoint in AWS China Regions.</td>
<td>March 9, 2021</td>
</tr>
<tr>
<td>Component environment variable reference</td>
<td>Added information about the environment variables that the AWS IoT Greengrass Core software sets for components. You can use these environment variables to get the thing name, AWS Region, and Greengrass nucleus version.</td>
<td>February 23, 2021</td>
</tr>
<tr>
<td>Manual installation</td>
<td>Added information about how to create required AWS resources manually or to install behind a firewall or network proxy. By using a manual installation, you don't need to give the installer permission to create resources in your AWS account, because you create the required AWS IoT and IAM resources. You can also configure your device to connect on port 443 or through a network proxy.</td>
<td>February 17, 2021</td>
</tr>
<tr>
<td>AWS IoT Greengrass Core IPC library update in AWS IoT Device SDK for Python v2</td>
<td>Version 1.5.4 of the AWS IoT Device SDK for Python v2 simplifies the steps required to connect to the AWS IoT Greengrass Core IPC service.</td>
<td>February 11, 2021</td>
</tr>
<tr>
<td>Support update for AWS IoT Device Tester for AWS IoT Greengrass</td>
<td>IDT for AWS IoT Greengrass V2 version 4.0.1 now supports using Greengrass nucleus version 2.0.4 with Greengrass CLI version 2.0.4 for device qualification.</td>
<td>February 5, 2021</td>
</tr>
<tr>
<td><strong>New tutorial to import Lambda functions</strong></td>
<td>Added a new console-based tutorial to import a Lambda function as a component that runs on Greengrass core device.</td>
<td>February 5, 2021</td>
</tr>
<tr>
<td><strong>AWS IoT Greengrass Core v2.0.4 software update</strong></td>
<td>This release provides version 2.0.4 of the Greengrass nucleus component. It includes the new <code>greengrassDataPlanePort</code> parameter to configure HTTPS communication over port 443 and fixes bugs. The minimal IAM policy now requires the <code>iam:GetPolicy</code> and <code>sts:GetCallerIdentity</code> when the AWS IoT Greengrass Core software installer is run with <code>--provision true.</code></td>
<td>February 4, 2021</td>
</tr>
<tr>
<td><strong>New secure tunneling component released</strong></td>
<td>Version 1.0.0 of the secure tunneling component is available for AWS IoT Greengrass. This public component enables you to use AWS IoT secure tunneling to establish secure bidirectional communication with a Greengrass core device that is behind restricted firewalls.</td>
<td>January 21, 2021</td>
</tr>
<tr>
<td><strong>AWS IoT Device Tester for AWS IoT Greengrass v4.0.1 released</strong></td>
<td>Version 4.0.1 of IDT for AWS IoT Greengrass V2 is available. This version enables you to use IDT to develop and run your custom test suites for device validation. This also includes code signed IDT applications for macOS and Windows.</td>
<td>December 22, 2020</td>
</tr>
<tr>
<td><strong>Initial release of AWS IoT Greengrass Version 2</strong></td>
<td>AWS IoT Greengrass V2 is a new major version release of AWS IoT Greengrass. This version adds several features such as modular software components and continuous deployments. These features make it easier for you to develop and manage edge applications.</td>
<td>December 15, 2020</td>
</tr>
</tbody>
</table>
AWS glossary

For the latest AWS terminology, see the AWS glossary in the AWS General Reference.