AWS Serverless Application Model

Developer Guide

AWS Serverless Application Model: Developer Guide

Table of Contents

What Is the AWS Serverless Application Model (AWS SAM)?

The AWS Serverless Application Model (AWS SAM) is an open-source framework that you can use to build serverless applications on AWS.

A **serverless application** is a combination of Lambda functions, event sources, and other resources that work together to perform tasks. Note that a serverless application is more than just a Lambda function—it can include additional resources such as APIs, databases, and event source mappings.

You can use AWS SAM to define your serverless applications. AWS SAM consists of the following components:

- **AWS SAM template specification**. You use this specification to define your serverless application. It provides you with a simple and clean syntax to describe the functions, APIs, permissions, configurations, and events that make up a serverless application. You use an AWS SAM template file to operate on a single, deployable, versioned entity that's your serverless application. For the full AWS SAM template specification, see AWS Serverless Application Model Specification.

- **AWS SAM command line interface (AWS SAM CLI)**. You use this tool to build serverless applications that are defined by AWS SAM templates. The CLI provides commands that enable you to verify that AWS SAM template files are written according to the specification, invoke Lambda functions locally, step-through debug Lambda functions, package and deploy serverless applications to the AWS Cloud, and so on. For details about how to use the AWS SAM CLI, including the full AWS SAM CLI Command Reference, see AWS SAM CLI (p. 48).

This guide shows you how to use AWS SAM to define, test, and deploy a simple serverless application. It also provides an example application (p. 3) that you can download, test locally, and deploy to the AWS Cloud. You can use this example application as a starting point for developing your own serverless applications.

Benefits of Using AWS SAM

Because AWS SAM integrates with other AWS services, creating serverless applications with AWS SAM provides the following benefits:

- **Single-deployment configuration**. AWS SAM makes it easy to organize related components and resources, and operate on a single stack. You can use AWS SAM to share configuration (such as memory and timeouts) between resources, and deploy all related resources together as a single, versioned entity.

- **Extension of AWS CloudFormation**. Because AWS SAM is an extension of AWS CloudFormation, you get the reliable deployment capabilities of AWS CloudFormation. You can define resources by using AWS CloudFormation in your AWS SAM template. Also, you can use the full suite of resources, intrinsic functions, and other template features that are available in AWS CloudFormation.

- **Built-in best practices**. You can use AWS SAM to define and deploy your infrastructure as config. This makes it possible for you to use and enforce best practices such as code reviews. Also, with a few lines of configuration, you can enable safe deployments through CodeDeploy, and can enable tracing by using AWS X-Ray.

- **Local debugging and testing**. The AWS SAM CLI lets you locally build, test, and debug serverless applications that are defined by AWS SAM templates. The CLI provides a Lambda-like execution environment locally. It helps you catch issues upfront by providing parity with the actual Lambda execution environment. To step through and debug your code to understand what the code is doing, you can use AWS SAM with AWS toolkits like the AWS Toolkit for JetBrains, AWS Toolkit for PyCharm, AWS Toolkit for IntelliJ, and AWS Toolkit for Visual Studio Code. This tightens the feedback loop by making it possible for you to find and troubleshoot issues that you might run into in the cloud.

- **Deep integration with development tools**. You can use AWS SAM with a suite of AWS tools for building serverless applications. You can discover new applications in the AWS Serverless Application Repository. For authoring, testing, and debugging AWS SAM–based serverless applications, you can use the AWS Cloud9 IDE. To build a deployment pipeline for your serverless applications, you can use CodeBuild, CodeDeploy, and CodePipeline. You can also use AWS CodeStar to get started with a project structure, code repository, and a CI/CD pipeline that's automatically configured for you. To deploy your serverless application, you can use the Jenkins plugin. You can use the Stackery.io toolkit to build production-ready applications.

Next Step

Quick Start (p. 3)

Quick Start

This guide walks you through the steps to build an example serverless application using the AWS Serverless Application Model (AWS SAM). You can use this example application as a starting point for developing your own serverless application.

Overview

The following steps outline how to download, test, and deploy a serverless application using AWS SAM:

1. **Initialize**. Download a sample application from template using `sam init`.
2. **Test Locally**. Test the application locally using `sam local invoke` and/or `sam local start-api`. Note that with these commands, even though your Lambda function is invoked locally, it reads from and writes to AWS resources in the AWS Cloud.
3. **Package**. When you're satisfied with your Lambda function, bundle the Lambda function, AWS SAM template, and any dependencies into an AWS CloudFormation deployment package using `sam package`.
4. **Deploy**. Deploy the application to the AWS Cloud using `sam deploy`. At this point, you're able to test your application in the AWS Cloud by invoking it using standard Lambda methods.

The example Hello World Application (p. 3) in the next section walks you through these steps in building your first serverless application using AWS SAM.

Hello World Application

In this exercise, you build a Hello World serverless application that represents a simple API backend. It has an Amazon API Gateway endpoint that supports a GET operation and a Lambda function. When a GET request is sent to the endpoint, API Gateway invokes the Lambda function. Then, AWS Lambda executes the function, which simply returns a `hello world` message.

The application has the following components:

- An AWS SAM template that defines two AWS resources for the Hello Word application: an API Gateway service with a GET operation, and a Lambda function. The template also defines the mapping between the API Gateway GET operation and the Lambda function.
- Application code that's written in Python.

Before You Begin

Make sure that you have the required setup for this exercise:

- You must have an AWS account with an IAM user that has administrator permissions. See Set Up an AWS Account.
- You must have the AWS SAM CLI (command line interface) installed. See Installing the AWS SAM CLI (p. 48).

Step 1: Initialize the Application

In this section, you download the sample application, which consists of an AWS SAM template and application code.

To initialize the application

1. Run the following command at an AWS SAM CLI command prompt.

```
sam init --runtime python3.6
```

2. Review the contents of the directory that the command created (`sam-app/`):

   - `template.yaml` – Defines two AWS resources that the Hello World application needs: a Lambda function and an API Gateway endpoint that supports a GET operation. The template also defines mapping between the two resources.
   - Content related to the Hello World application code:
     - `hello_world/` directory – Contains the application code, which returns `hello world` when you run it.

     Note
     For this exercise, the application code is written in Python, and you specify the runtime in the `init` command. AWS Lambda supports additional languages for creating application code. If you specify another supported runtime, the `init` command provides the Hello World code in the specified language, and a `README.md` file that you can follow along for that language. For information about supported runtimes, see Lambda Execution Environment and Available Libraries.

Step 2: Test the Application Locally

Now that you have the AWS SAM application on your local machine, follow the steps below to test it locally.

To test the application locally

1. Start the API Gateway endpoint locally. You must run the following command from the directory that contains the `template.yaml` file.

```
sam local start-api
```

   The command returns an API Gateway endpoint, which you can send requests to for local testing.
2. Test the application. Copy the API Gateway endpoint URL, paste it in the browser, and choose **Enter**. An example API Gateway endpoint URL is `http://127.0.0.1:3000/hello`.

   API Gateway locally invokes the Lambda function that the endpoint is mapped to. The Lambda function executes in the local Docker container and returns `hello world`. API Gateway returns a response to the browser that contains the text.

Exercise: Change the message string

After successfully testing the sample application, you can experiment with making a simple modification: change the message string that's returned.

1. Edit the `hello_world/app.py` file to change the message string from `'hello world'` to `'Hello World!'`.
2. Reload the test URL in your browser and observe the new string.

You will notice that your new code is loaded dynamically, without your having restart the `sam local` process.

Step 3: Package the Application

After testing your application locally, you use the AWS SAM CLI to create a deployment package. You use this package to deploy the application to the AWS Cloud.

Note
In the following steps, you create a .zip file for the contents of the `hello_world/` directory, which contains the application code. This .zip file is the **deployment package** for your serverless application. For more information, see Creating a Deployment Package (Python) in the *AWS Lambda Developer Guide*.

To create a Lambda deployment package

1. Create an S3 bucket in the location where you want to save the packaged code. If you want to use an existing S3 bucket, skip this step.

```
aws s3 mb s3://bucketname
```

2. Create the Lambda function deployment package by running the following `package` AWS SAM CLI command at the command prompt.

```
sam package \
    --output-template-file packaged.yaml \
    --s3-bucket bucketname
```

The command does the following:

- Zips the contents of the `sam-app/hello_world/` directory and uploads it to Amazon S3.
- Outputs a new template file, called `packaged.yaml`, which you use in the next step to deploy the application to the AWS Cloud. The `packaged.yaml` template file is similar to the original template file (`template.yaml`), but has one key difference—the `CodeUri` property points to the Amazon S3 bucket and object that contains the Lambda function code and dependencies. The following snippet from an example `packaged.yaml` template file shows this property:

```
HelloWorldFunction:
    Type: AWS::Serverless::Function # For more information about function
 resources, see https://github.com/awslabs/serverless-application-model/blob/master/
versions/2016-10-31.md#awsserverlessfunction
    Properties:
      CodeUri: s3://bucketname/fbd77a3647a4f47a352fcObjectGUID

...
```

Step 4: Deploy the Application

Now that you've created the deployment package, you use it to deploy the application to the AWS Cloud. You then test the application there.

To deploy the serverless application to the AWS Cloud

- In the AWS SAM CLI, use the `deploy` command to deploy all of the resources that you defined in the template.

```
sam deploy \
    --template-file packaged.yaml \
    --stack-name sam-app \
    --capabilities CAPABILITY_IAM \
    --region us-east-1
```

  In the command, the `--capabilities` parameter enables AWS CloudFormation to create an IAM role.

  The example uses the us-east-1 AWS Region to create all resources. You can choose to specify any other Region.

  AWS CloudFormation creates the AWS resources as defined in the template, and groups them in an entity called a *stack* in AWS CloudFormation. You can access this stack in the console.

To test the serverless application in the AWS Cloud

1. Open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. In the navigation column on the left, choose **Applications**. In the list shown, choose the application that you created in the preceding step.
3. Under **Resources**, expand the **ServerlessRestApi** item, and copy the **API endpoint** URL.
4. Open a browser, paste the endpoint URL, append `/hello` to the end of the URL and choose **Enter**.

   This sends a GET request to the endpoint. API Gateway invokes the Lambda function that the endpoint is mapped to. AWS Lambda executes the Lambda function and returns `hello world`. API Gateway returns a response with the text to the browser.

Additional Example Applications

There are two ways you can explore additional serverless applications that are built using AWS SAM:

- Download applications from templates using the AWS SAM CLI command `sam init`. See Example Serverless Applications (p. 43) for instructions about these examples.
- Explore additional AWS SAM example applications in the AWS SAM GitHub repository.

AWS SAM Template Concepts

When you create a serverless application by using AWS SAM, your main objective is to construct an AWS SAM template file that represents the architecture of your serverless application.

The AWS SAM template file is a YAML or JSON configuration file that adheres to the open source AWS Serverless Application Model specification. You use the template to declare all of the AWS resources that comprise your serverless application.

AWS SAM templates are an extension of AWS CloudFormation templates. That is, any resource that you can declare in an AWS CloudFormation template you can also declare in an AWS SAM template. In addition, you can use the additional resource types provided by AWS SAM—for instance, the resources described in Declaring Serverless Resources (p. 8)—as shortcuts for some components of your serverless application.

For an introduction to AWS CloudFormation templates, see Learn Template Basics.

For information about the AWS SAM template specification, see AWS Serverless Application Model Specification.

The following sections describe the types of AWS resources that can be declared in AWS SAM template files.

Topics

Example

The following example AWS SAM template describes the configuration of a Lambda function and an API Gateway endpoint.

```
Transform: 'AWS::Serverless-2016-10-31'
Resources:

  ThumbnailFunction:
    # This resource creates a Lambda function.
    Type: 'AWS::Serverless::Function'

    Properties:

      # This function uses the Nodejs v6.10 runtime.
      Runtime: nodejs6.10

      # This is the Lambda function's handler.
      Handler: index.handler

      # The location of the Lambda function code.
      CodeUri: ./src

      # Event sources to attach to this function. In this case, we are attaching
      # one API Gateway endpoint to the Lambda function. The function is
```

```
      # called when a HTTP request is made to the API Gateway endpoint.
      Events:

        ThumbnailApi:
            # Define an API Gateway endpoint that responds to HTTP GET at /thumbnail
            Type: Api
            Properties:
                Path: /thumbnail
                Method: GET
```

Declaring Serverless Resources

AWS SAM defines the following resources that are specifically designed for serverless applications:

Topics

AWS::Serverless::Api

This resource type describes an API Gateway resource. It's useful for advanced use cases where you want full control and flexibility when you configure your APIs. For most scenarios, we recommend that you create APIs by specifying this resource type as an event source of your `AWS::Serverless::Function` resource, as shown in the following example.

```
AWS::Serverless::API
  Properties:
    StageName: prod
    DefinitionUri: swagger.yml
```

For a list of properties, see AWS::Serverless::Api in the AWS SAM GitHub repository.

AWS::Serverless::Application

This resource type embeds a serverless application from the AWS Serverless Application Repository or from an Amazon S3 bucket as a nested application. Nested applications are deployed as nested stacks, which can contain multiple other resources. For more information about nested applications, see Nested Applications (p. 11).

The following is an example of a nested application from the AWS Serverless Application Repository:

```
AWS::Serverless::Application
  Properties:
    Location:
      ApplicationId: arn:aws:serverlessrepo:region:account-id:applications/application-name
      SemanticVersion: 1.0.0
    Parameters:
      StringParameter: parameter-value
      IntegerParameter: 2
```

The following is an example of a nested application that's hosted in an Amazon S3 bucket. In this example, *sam-template-object* is the name of a packaged AWS SAM template:

```
AWS::Serverless::Application
  Properties:
    Location: https://s3.region.amazonaws.com/bucket-name/sam-template-object
    Parameters:
      StringParameter: parameter-value
      IntegerParameter: 2
```

For a list of properties, see AWS::Serverless::Application in the AWS SAM GitHub repository.

For details about how to obtain the required values of an application that you want to nest, see Nested Applications (p. 11).

AWS::Serverless::Function

This resource type describes configuration information for creating a Lambda function. You can describe any event source that you want to attach to the Lambda function—such as Amazon S3, Amazon DynamoDB Streams, and Amazon Kinesis Data Streams.

The following is an example of a serverless function:

```
AWS::Serverless::Function
  Handler: index.js
  Runtime: nodejs6.10
  CodeUri: 's3://my-code-bucket/my-function.zip'
  Description: Creates thumbnails of uploaded images
  MemorySize: 1024
  Timeout: 15
  Policies:
   - AWSLambdaExecute # Managed Policy
   - Version: '2012-10-17' # Policy Document
     Statement:
       - Effect: Allow
         Action:
           - s3:GetObject
           - s3:GetObjectACL
         Resource: 'arn:aws:s3:::my-bucket/*'
  Environment:
    Variables:
      TABLE_NAME: my-table
  Events:
    PhotoUpload:
      Type: S3
      Properties:
        Bucket: my-photo-bucket
  Tags:
    AppNameTag: ThumbnailApp
    DepartmentNameTag: ThumbnailDepartment
```

For a list of properties, see AWS::Serverless::Function in the AWS SAM GitHub repository.

AWS::Serverless::LayerVersion

This resource type creates a Lambda layer version (LayerVersion) that contains library or runtime code that's needed by a Lambda function. When a serverless layer version is transformed, AWS SAM also transforms the logical ID of the resource so that old layer versions aren't automatically deleted by AWS CloudFormation when the resource is updated.

The following is an example of a layer version:

```
AWS::Serverless::LayerVersion
  Properties:
    LayerName: MyLayer
    Description: Layer description
    ContentUri: 's3://my-bucket/my-layer.zip'
    CompatibleRuntimes:
      - nodejs6.10
      - nodejs8.10
    LicenseInfo: 'Available under the MIT-0 license.'
    RetentionPolicy: Retain
```

For a list of properties, see AWS::Serverless::LayerVersion in the AWS SAM GitHub repository.

AWS::Serverless::SimpleTable

This resource type provides simple syntax for describing how to create DynamoDB tables. Here's an example:

```
AWS::Serverless::SimpleTable
  Properties:
    PrimaryKey:
      Name: id
      Type: String
    ProvisionedThroughput:
      ReadCapacityUnits: 5
      WriteCapacityUnits: 5
```

For a list of properties, see AWS::Serverless::SimpleTable in the AWS SAM GitHub repository.

For reference documentation for SAM template resources, see AWS SAM Specification on GitHub.

Declaring AWS CloudFormation Resources

AWS SAM is a higher-level abstraction of AWS CloudFormation that simplifies serverless application development. AWS SAM template files are AWS CloudFormation template files with a few additional resource types defined that are specific to serverless applications—such as API Gateway endpoints and Lambda functions. This means that AWS SAM supports the full suite of resources, intrinsic functions, and other template features that are available in AWS CloudFormation.

For example, the following AWS SAM template creates a Lambda function by using AWS SAM resource syntax. It also creates an Amazon S3 bucket by using AWS CloudFormation resource syntax:

```
Transform: 'AWS::Serverless-2016-10-31'
Resources:

  MyFunction:
    # SAM resource to create a Lambda function
    Type: 'AWS::Serverless::Function'

    Properties:
      Runtime: nodejs6.10
      ....

  MyBucket:
    # AWS CloudFormation resource to create an S3 bucket
```

```
    Type: 'AWS::S3::Bucket'
```

For an introduction to AWS CloudFormation templates, see Learn Template Basics.

For more information about working with AWS CloudFormation templates, see Working with AWS CloudFormation Templates.

For the full reference for AWS CloudFormation templates, see the AWS CloudFormation Template Reference.

Nested Applications

A serverless application can include one or more **nested applications**. You can deploy a nested application as a stand-alone artifact or as a component of a larger application.

As serverless architectures grow, common patterns emerge in which the same components are defined in multiple application templates. You can now separate out common patterns as dedicated applications, and then nest them as part of new or existing application templates. With nested applications, you can stay more focused on the business logic that's unique to your application.

To define a nested application in your serverless application, use the AWS::Serverless::Application resource type.

You can define nested applications from the following two sources:

- An **AWS Serverless Application Repository application** – You can define nested applications by using applications that are available to your account in the AWS Serverless Application Repository. These can be *private* applications in your account, applications that are *privately shared* with your account, or applications that are *publicly shared* in the AWS Serverless Application Repository. For more information about the different deployment permissions levels, see Application Deployment Permissions and Publishing Applications in the *AWS Serverless Application Repository Developer Guide*.
- A **local application** – You can define nested applications by using applications that are stored on your local file system.

See the following sections for details on how to use AWS SAM to define both of these types of nested applications in your serverless application.

> **Note**
> The maximum number of applications that can be nested in a serverless application is 200.
> The maximum number of parameters a nested application can have is 60.

Defining a Nested Application from the AWS Serverless Application Repository

You can define nested applications by using applications that are available in the AWS Serverless Application Repository. You can also store and distribute applications that contain nested applications using the AWS Serverless Application Repository. To review details of a nested application in the AWS Serverless Application Repository, you can use the AWS SDK, the AWS CLI, or the Lambda console.

To define an application that's hosted in the AWS Serverless Application Repository in your serverless application's AWS SAM template, use the **Copy as SAM Resource** button on the detail page of every AWS Serverless Application Repository application. To do this, follow these steps:

1. Make sure that you're signed in to the AWS Management Console.

2. Find the application that you want to nest in the AWS Serverless Application Repository by using the steps in the Browsing, Searching, and Deploying Applications section of the *AWS Serverless Application Repository Developer Guide*.
3. Choose the **Copy as SAM Resource** button. The SAM template section for the application that you're viewing is now in your clipboard.
4. Paste the SAM template section into the `Resources:` section of the SAM template file for the application that you want to nest in this application.

The following is an example SAM template section for a nested application that's hosted in the AWS Serverless Application Repository:

```
Transform: AWS::Serverless-2016-10-31

Resources:
  applicationaliasname:
    Type: AWS::Serverless::Application
    Properties:
      Location:
        ApplicationId: arn:aws:serverlessrepo:us-
east-1:123456789012:applications/application-alias-name
        SemanticVersion: 1.0.0
      Parameters:
        # Optional parameter that can have default value overridden
        # ParameterName1: 15 # Uncomment to override default value
        # Required parameter that needs value to be provided
        ParameterName2: YOUR_VALUE
```

If there are no required parameter settings, you can omit the `Parameters:` section of the template.

Important
Applications that contain nested applications hosted in the AWS Serverless Application Repository inherit the nested applications' sharing restrictions.
For example, suppose an application is publicly shared, but it contains a nested application that's only privately shared with the AWS account that created the parent application. In this case, if your AWS account doesn't have permission to deploy the nested application, you aren't able to deploy the parent application. For more information about permissions to deploy applications, see Application Deployment Permissions and Publishing Applications in the *AWS Serverless Application Repository Developer Guide*.

Defining a Nested Application from the Local File System

You can define nested applications by using applications that are stored on your local file system. You do this by specifying the path to the AWS SAM template file that's stored on your local file system.

The following is an example SAM template section for a nested local application:

```
Transform: AWS::Serverless-2016-10-31

Resources:
  applicationaliasname:
    Type: AWS::Serverless::Application
    Properties:
      Location: ../my-other-app/template.yaml
      Parameters:
        # Optional parameter that can have default value overridden
        # ParameterName1: 15 # Uncomment to override default value
        # Required parameter that needs value to be provided
```

```
        ParameterName2: YOUR_VALUE
```

If there are no parameter settings, you can omit the `Parameters:` section of the template.

Deploying Nested Applications

You can deploy your nested application by using the AWS SAM CLI command `sam deploy`. For more details, see Deploying Serverless Applications (p. 34).

> **Note**
> When you deploy an application that contains nested applications, you must acknowledge that. You do this by passing CAPABILITY_AUTO_EXPAND to the CreateCloudFormationChangeSet API,git status or using the `aws serverlessrepo create-cloud-formation-change-set` AWS CLI command.
> For more information about acknowledging nested applications, see Acknowledging IAM Roles, Resource Policies, and Nested Applications when Deploying Applications in the *AWS Serverless Application Repository Developer Guide*.

Controlling Access to API Gateway APIs

You can use AWS SAM to control who can access your API Gateway APIs by enabling authorization within your AWS SAM template.

AWS SAM supports a few mechanisms for controlling access to your API Gateway APIs:

- **Lambda authorizers**. A Lambda authorizer (formerly known as a *custom authorizer*) is a Lambda function that you provide to control access to your API. When your API is called, this Lambda function is invoked with a request context or an authorization token that are provided by the client application. The Lambda function returns a policy document that specifies the operations that the caller is authorized to perform, if any. For more information about Lambda authorizers, see Use API Gateway Lambda Authorizers in the *API Gateway Developer Guide*. For examples of Lambda authorizers, see the Defining Lambda Token Authorizers (p. 14) and Defining Lambda Request Authorizers (p. 14) sections in this topic.

- **Amazon Cognito user pools**. Amazon Cognito user pools are user directories in Amazon Cognito. A client of your API must first sign a user in to the user pool and obtain an identity or access token for the user. Then your API is called with one of the returned tokens. The API call succeeds only if the required token is valid. For more information about Amazon Cognito user pools, see Control Access to REST API Using Amazon Cognito User Pools as Authorizer in the *API Gateway Developer Guide*. For an example of Amazon Cognito user pools, see the Defining Cognito User Pools (p. 15) section.

Choosing a Mechanism to Control Access

The mechanism that you choose to control access to your API Gateway APIs depends on a few factors. For example, if you have a greenfield project that doesn't have either authorization or access control set up yet, then Amazon Cognito user pools might be your best option. This is because by setting up user pools, you also set up both authentication and access control automatically.

However, if your application already has authentication set up, then using Lambda authorizers might be the best option. This is because you can call your existing authentication service and return a policy document based on the response. Also, if the nature of your application requires custom authentication and/or access control logic that user pools don't support, then Lambda authorizers might again be your best option.

After you've decided which mechanism to use, see the corresponding section in this topic to see how to use AWS SAM to configure your application to use that mechanism.

Defining Lambda Token Authorizers

You can control access to your APIs by defining a Lambda Token authorizer within your AWS SAM template. To do this, you use the API Auth Object data type.

The following is an example AWS SAM template section for a Lambda Token authorizer:

```
Resources:
  MyApi:
    Type: AWS::Serverless::Api
    Properties:
      StageName: Prod
      Auth:
        DefaultAuthorizer: MyLambdaTokenAuthorizer
        Authorizers:
          MyLambdaTokenAuthorizer:
            FunctionArn: !GetAtt MyAuthFunction.Arn

  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./src
      Handler: index.handler
      Runtime: nodejs8.10
      Events:
        GetRoot:
          Type: Api
          Properties:
            RestApiId: !Ref MyApi
            Path: /
            Method: get

  MyAuthFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./src
      Handler: authorizer.handler
      Runtime: nodejs8.10
```

For more information about API Gateway Lambda authorizers, see Use API Gateway Lambda Authorizers in the *API Gateway Developer Guide*.

For a full sample application that includes a Lambda Token authorizer, see API Gateway + Lambda TOKEN Authorizer Example.

Defining Lambda Request Authorizers

You can control access to your APIs by defining a Lambda Request authorizer within your AWS SAM template. To do this, you use the API Auth Object data type.

The following is an example AWS SAM template section for a Lambda Request authorizer:

```
Resources:
  MyApi:
    Type: AWS::Serverless::Api
    Properties:
      StageName: Prod
```

```
      Auth:
        DefaultAuthorizer: MyLambdaRequestAuthorizer
        Authorizers:
          MyLambdaRequestAuthorizer:
            FunctionPayloadType: REQUEST
            FunctionArn: !GetAtt MyAuthFunction.Arn
            Identity:
              QueryStrings:
                - auth

  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./src
      Handler: index.handler
      Runtime: nodejs8.10
      Events:
        GetRoot:
          Type: Api
          Properties:
            RestApiId: !Ref MyApi
            Path: /
            Method: get

  MyAuthFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./src
      Handler: authorizer.handler
      Runtime: nodejs8.10
```

For more information about API Gateway Lambda authorizers, see Use API Gateway Lambda Authorizers in the *API Gateway Developer Guide*.

For a full sample application that includes a Lambda Request authorizer, see API Gateway + Lambda REQUEST Authorizer Example.

Defining Cognito User Pools

You can control access to your APIs by defining Amazon Cognito user pools within your AWS SAM template. To do this, you use the API Auth Object data type.

The following is an example AWS SAM template section for a user pool:

```
Resources:
  MyApi:
    Type: AWS::Serverless::Api
    Properties:
      StageName: Prod
      Cors: "'*'"
      Auth:
        DefaultAuthorizer: MyCognitoAuthorizer
        Authorizers:
          MyCognitoAuthorizer:
            UserPoolArn: !GetAtt MyCognitoUserPool.Arn

  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./src
      Handler: lambda.handler
      Runtime: nodejs8.10
      Events:
```

```
      Root:
        Type: Api
        Properties:
          RestApiId: !Ref MyApi
          Path: /
          Method: GET

MyCognitoUserPool:
  Type: AWS::Cognito::UserPool
  Properties:
    UserPoolName: !Ref CognitoUserPoolName
    Policies:
      PasswordPolicy:
        MinimumLength: 8
    UsernameAttributes:
      - email
    Schema:
      - AttributeDataType: String
        Name: email
        Required: false

MyCognitoUserPoolClient:
  Type: AWS::Cognito::UserPoolClient
  Properties:
    UserPoolId: !Ref MyCognitoUserPool
    ClientName: !Ref CognitoUserPoolClientName
    GenerateSecret: false
```

For more information about Amazon Cognito user pools, see Control Access to a REST API Using Amazon Cognito User Pools as Authorizer in the *API Gateway Developer Guide*.

For a full sample application that includes a user pool as an authorizer, see API Gateway + Cognito Auth + Cognito Hosted Auth Example.

Testing and Debugging Serverless Applications

With the AWS SAM command line interface (CLI), you can locally test and "step-through" debug your serverless applications, before uploading your application to the AWS Cloud. You can verify whether your application is behaving as expected, debug what's wrong, and fix any issues, before going through the steps of packaging and deploying your application.

When you locally invoke a Lambda function in debug mode within the AWS SAM CLI, you can then attach a debugger to it. With the debugger, you can step through your code line by line, see the values of various variables, and fix issues the same way you would for any other application.

Topics

Building Applications with Dependencies

You can use the `sam build (p. 56)` command to compile dependencies for Lambda functions written in Python. For example, if you write code that uses Python packages, such as a graphics library for image processing, you need to create a deployment package that works on the Amazon Linux AMI. The `sam build` command allows you to easily create deployment artifacts that target Lambda's execution environment, so that the functions you build locally run in a similar environment in the AWS Cloud.

The `sam build` command iterates through the functions in your application, looks for a manifest file (such as `requirements.txt`) that contain the dependencies, and automatically creates deployment artifacts that you can deploy to Lambda using the `sam package` and `sam deploy` commands.

If your Lambda function depends on packages that have natively compiled programs, you can use the `--use-container` flag. The `--use-container` flag compiles your functions in a Lambda-like environment locally, so they are in the right format when you deploy them to the AWS Cloud.

Examples:

```
# Build a deployment package
```

```
$ sam build

# Run the build process inside an AWS Lambda-like Docker container
$ sam build --use-container

# Build and run your functions locally
$ sam build && sam local invoke

# Build and package for deployment
$ sam build && sam package --s3-bucket <bucketname>

# For more options
$ sam build --help
```

Invoking Functions Locally

You can invoke your function locally by using the `sam local invoke` (p. 60) command and providing its function logical ID and an event file. Alternatively, `sam local invoke` also accepts `stdin` as an event.

> **Note**
> The `sam local invoke` command described in this section corresponds to the AWS CLI command `aws lambda invoke`. You can use either version of this command to invoke a Lambda function that you've uploaded to the AWS Cloud.

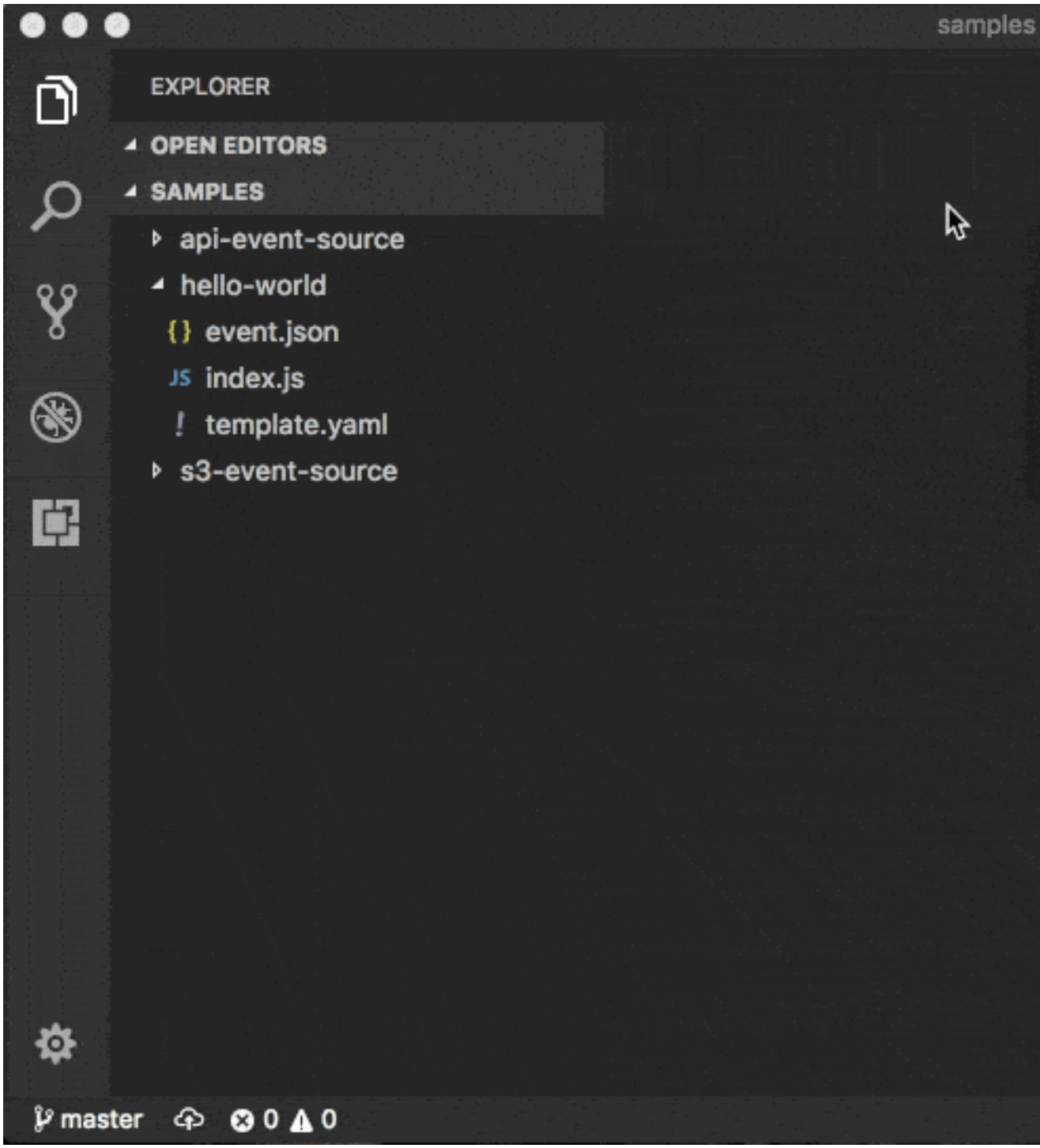
Examples:

```
# Invoking function with event file
$ sam local invoke "Ratings" -e event.json

# Invoking function with event via stdin
$ echo '{"message": "Hey, are you there?" }' | sam local invoke "Ratings"

# For more options
$ sam local invoke --help
```

This animation shows invoking a Lambda function locally using Microsoft Visual Studio Code:



Environment Variable File

You can use the `--env-vars` argument with the `invoke` or `start-api` commands. You do this to provide a JSON file that contains values to override the environment variables that are already defined in your function template. Structure the file as follows:

```
{
```

```
  "MyFunction1": {
    "TABLE_NAME": "localtable",
    "BUCKET_NAME": "testBucket"
  },
  "MyFunction2": {
    "TABLE_NAME": "localtable",
    "STAGE": "dev"
  },
}
```

For example, if you save this content in a file named `env.json`, then the following command uses this file to override the included environment variables:

```
$ sam local invoke --env-vars env.json
```

Layers

If your application includes layers, see Working with Layers (p. 22) for more information about how to debug layers issues on your local host.

Running API Gateway Locally

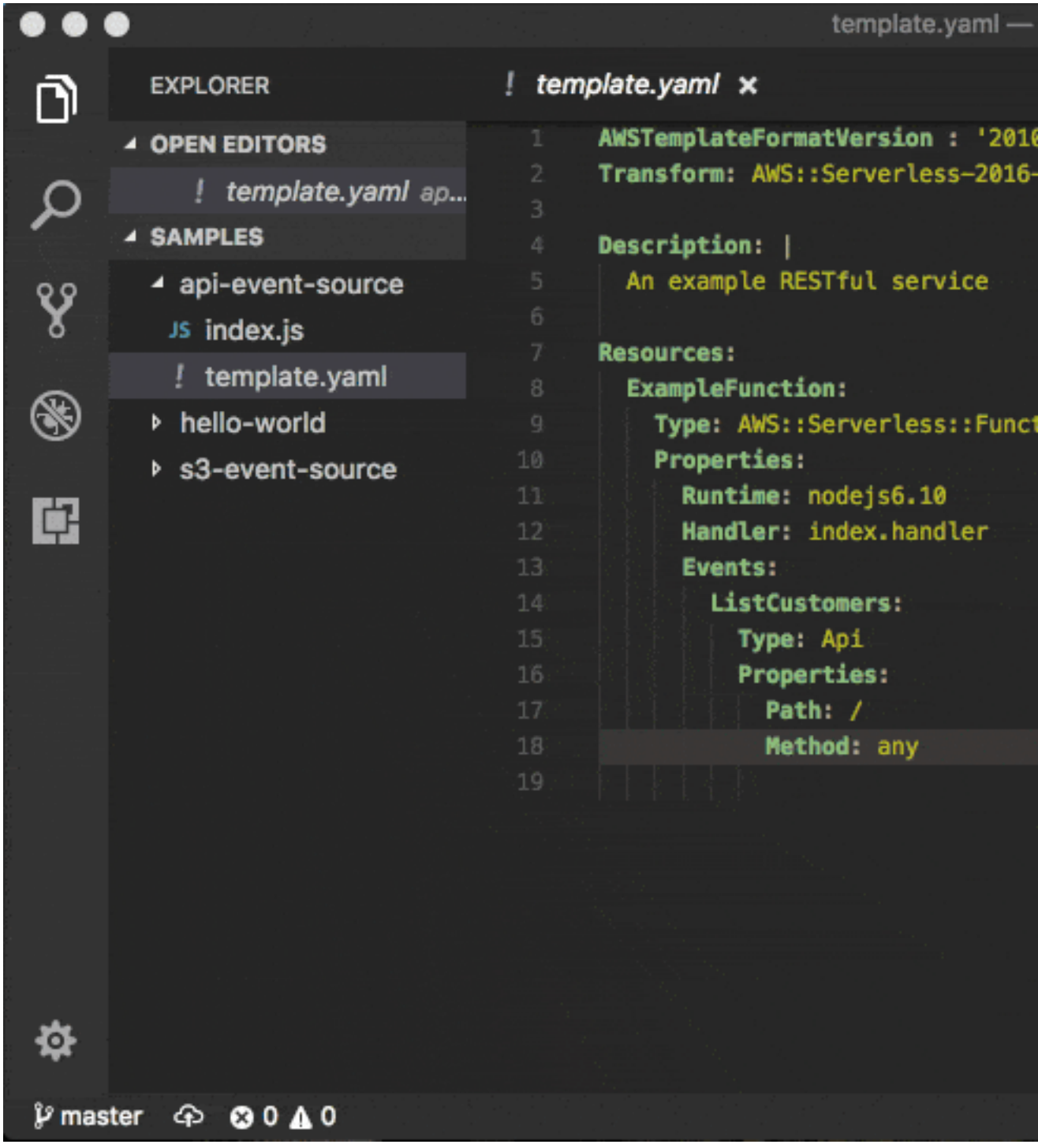
Use the `sam local start-api (p. 61)` command to start a local instance of API Gateway that you will use to test HTTP request/response functionality. This functionality features hot reloading to enable you to quickly develop and iterate over your functions.

> **Note**
> "Hot reloading" is when only the files that changed are refreshed without losing the state of the application. In contrast, "live reloading" is when the entire application is refreshed, such that the state of the application is lost.

```
$ sam local start-api
```

AWS SAM automatically finds any functions within your AWS SAM template that have `Api` event sources defined. Then, it mounts them at the defined HTTP paths.

This animation shows running API Gateway locally using Microsoft Visual Studio Code:



In the following example, the `Ratings` function mounts `ratings.py:handler()` at `/ratings` for `GET` requests:

```
Ratings:
  Type: AWS::Serverless::Function
  Properties:
    Handler: ratings.handler
```

```
    Runtime: python3.6
    Events:
      Api:
        Type: Api
        Properties:
          Path: /ratings
          Method: get
```

By default, AWS SAM uses Proxy Integration and expects the response from your Lambda function to include one or more of the following: `statusCode`, `headers`, or `body`.

For example:

```
// Example of a Proxy Integration response
exports.handler = (event, context, callback) => {
    callback(null, {
        statusCode: 200,
        headers: { "x-custom-header" : "my custom header value" },
        body: "hello world"
    });
}
```

For examples in other AWS Lambda languages, see Proxy Integration.

Environment Variable File

You can use the `--env-vars` argument with the `invoke` or `start-api` commands to provide a JSON file that contains values to override the environment variables already defined in your function template. Structure the file as follows:

```
{
  "MyFunction1": {
    "TABLE_NAME": "localtable",
    "BUCKET_NAME": "testBucket"
  },
  "MyFunction2": {
    "TABLE_NAME": "localtable",
    "STAGE": "dev"
  },
}
```

For example, if you save this content in a file named `env.json`, then the following command uses this file to override the included environment variables:

```
$ sam local start-api --env-vars env.json
```

Layers

If your application includes layers, see Working with Layers (p. 22) for more information about how to debug layers issues on your local host.

Working with Layers

The AWS SAM CLI supports applications that include layers. For more information about layers, see Lambda Layers.

The following is an example AWS SAM template with a Lambda function that includes a layer:

```
ServerlessFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: .
    Handler: my_handler
    Runtime: Python3.7
    Layers:
        - <LayerVersion ARN>
```

For more information about including layers in your application, see either AWS::Serverless::Function in the AWS SAM GitHub repository, or AWS::Lambda::Function in the *AWS CloudFormation User Guide*.

When you invoke your function using one of the sam local CLI subcommands, the layers package of your function is downloaded and cached on your local host. See the following chart for default cache directory locations. After the package is cached, the AWS SAM CLI overlays the layers onto a Docker image that's used to invoke your function. The AWS SAM CLI generates the names of the images it builds, as well as the LayerVersions that are held in the cache. You can find more details about the schema in the following sections.

To inspect the overlaid layers, execute the following command to start a bash session in the image that you want to inspect:

```
docker run -it --entrypoint=/bin/bash samcli/lambda:<Tag following the schema outlined in
 Docker Image Tag Schema> -i
```

Layer Caching Directory name schema

Given a LayerVersionArn that's defined in your template, the AWS SAM CLI extracts the LayerName and Version from the ARN. It creates a directory to place the layer contents in named `LayerName-Version-<first 10 characters of sha256 of ARN>`.

Example:

```
ARN = arn:aws:lambda:us-west-2:111111111111:layer:myLayer:1
Directory name = myLayer-1-926eeb5ff1
```

Docker Images tag schema

To compute the unique layers hash, combine all unique layer names with a delimiter of '-', take the SHA256 hash, and then take the first 25 characters.

Example:

```
ServerlessFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: .
    Handler: my_handler
    Runtime: Python3.7
    Layers:
        - arn:aws:lambda:us-west-2:111111111111:layer:myLayer:1
        - arn:aws:lambda:us-west-2:111111111111:layer:mySecondLayer:1
```

Unique names are computed the same as the Layer Caching Directory name schema:

```
arn:aws:lambda:us-west-2:111111111111:layer:myLayer:1 = myLayer-1-926eeb5ff1
arn:aws:lambda:us-west-2:111111111111:layer:mySecondLayer:1 = mySecondLayer-1-6bc1022bdf
```

To compute the unique layers hash, combine all unique layer names with a delimiter of '-', take the sha256 hash, and then take the first 25 characters:

```
myLayer-1-926eeb5ff1-mySecondLayer-1-6bc1022bdf = 2dd7ac5ffb30d515926aef
```

Then combine this value with the function's runtime, with a delimiter of '-':

```
python3.7-2dd7ac5ffb30d515926aefffd
```

Default Cache Directory Locations

OS	Location
Windows 7	C:\Users\<user>\AppData\Roaming\AWS SAM
Windows 8	C:\Users\<user>\AppData\Roaming\AWS SAM
Windows 10	C:\Users\<user>\AppData\Roaming\AWS SAM
macOS	~/.aws-sam/layers-pkg
Unix	~/.aws-sam/layers-pkg

Running Automated Tests

You can use the `sam local invoke` command to manually test your code by running Lambda functions locally. With the AWS SAM CLI, you can easily author automated integration tests by first running tests against local Lambda functions before deploying to the AWS Cloud.

The `sam local start-lambda` command starts a local endpoint that emulates the AWS Lambda invoke endpoint. You can invoke it from your automated tests. Because this endpoint emulates the AWS Lambda invoke endpoint, you can write tests once, and then run them (without any modifications) against the local Lambda function, or against a deployed Lambda function. You can also run the same tests against a deployed AWS SAM stack in your CI/CD pipeline.

This is how the process works:

1. Start the local Lambda endpoint.

   Start the local Lambda endpoint by running the following command in the directory that contains your AWS SAM template:

   ```
   sam local start-lambda
   ```

   This command starts a local endpoint at `http://127.0.0.1:3001` that emulates AWS Lambda. You can run your automated tests against this local Lambda endpoint. When you invoke this endpoint using the AWS CLI or SDK, it locally executes the Lambda function that's specified in the request, and returns a response.
2. Run an integration test against the local Lambda endpoint.

In your integration test, you can use the AWS SDK to invoke your Lambda function with test data, wait for response, and verify that the response is what you expect. To run the integration test locally, you should configure the AWS SDK to send a Lambda Invoke API call to invoke the local Lambda endpoint that you started in previous step.

The following is a Python example (the AWS SDKs for other languages have similar configurations):

```
import boto3
import botocore

# Set "running_locally" flag if you are running the integration test locally
running_locally = True

if running_locally:

    # Create Lambda SDK client to connect to appropriate Lambda endpoint
    lambda_client = boto3.client('lambda',
        region_name="us-west-2",
        endpoint_url="http://127.0.0.1:3001",
        use_ssl=False,
        verify=False,
        config=botocore.client.Config(
            signature_version=botocore.UNSIGNED,
            read_timeout=0,
            retries={'max_attempts': 0},
        )
    )
else:
    lambda_client = boto3.client('lambda')


# Invoke your Lambda function as you normally usually do. The function will run
# locally if it is configured to do so
response = lambda_client.invoke(FunctionName="HelloWorldFunction")

# Verify the response
assert response == "Hello World"
```

You can use this code to test deployed Lambda functions by setting `running_locally` to `False`. This sets up the AWS SDK to connect to AWS Lambda in the AWS Cloud.

Generating Sample Event Payloads

To make local development and testing of Lambda functions easier, you can generate and customize event payloads for a number of AWS services like API Gateway, AWS CloudFormation, Amazon S3, and so on.

For the full list of services that you can generate sample event payloads for, use this command:

```
sam local generate-event --help
```

For the list of options you can use for a particular service, use this command:

```
sam local generate-event [SERVICE] --help
```

Examples:

```
#Generates the event from S3 when a new object is created
$ sam local generate-event s3 put

# Generates the event from S3 when an object is deleted
$ sam local generate-event s3 delete
```

Working with Logs

To simplify troubleshooting, the AWS SAM CLI has a command called `sam logs (p. 64)`. This command lets you fetch logs generated by your Lambda function from the command line.

> **Note**
> The `sam logs` command works for all AWS Lambda functions, not just the ones you deploy using AWS SAM.

Fetching Logs by AWS CloudFormation Stack

When your function is a part of an AWS CloudFormation stack, you can fetch logs by using the function's logical ID:

```
sam logs -n HelloWorldFunction --stack-name mystack
```

Fetching Logs by Lambda Function Name

Or, you can fetch logs by using the function's name:

```
sam logs -n mystack-HelloWorldFunction-1FJ8PD
```

Tailing Logs

Add the `--tail` option to wait for new logs and see them as they arrive. This is helpful during deployment or when you're troubleshooting a production issue.

```
sam logs -n HelloWorldFunction --stack-name mystack --tail
```

Viewing Logs for a Specific Time Range

You can view logs for a specific time range by using the `-s` and `-e` options:

```
sam logs -n HelloWorldFunction --stack-name mystack -s '10min ago' -e '2min ago'
```

Filtering Logs

Use the `--filter` option to quickly find logs that match terms, phrases, or values in your log events:

```
sam logs -n HelloWorldFunction --stack-name mystack --filter "error"
```

In the output, the AWS SAM CLI underlines all occurrences of the word "error" so you can easily locate the filter keyword within the log output.

Error Highlighting

When your Lambda function crashes or times out, the AWS SAM CLI highlights the timeout message in red. This helps you easily locate specific executions that are timing out within a giant stream of log output.

JSON Pretty Printing

If your log messages print JSON strings, the AWS SAM CLI automatically pretty prints the JSON to help you visually parse and understand the JSON.

Step-Through Debugging Lambda Functions Locally

You can use AWS SAM with a number of AWS toolkits to test and debug your serverless applications locally.

For example, you can perform step-through debugging of your Lambda functions. Step-through debugging makes it easier to understand what the code is doing. It tightens the feedback loop by making it possible for you to find and troubleshoot issues that you might run into in the cloud.

Using AWS Toolkits

AWS toolkits are plugins that provide you with the ability to perform many common debugging tasks, like setting breakpoints, executing code line by line, and inspecting the values of variables. Toolkits make it easier for you to develop, debug, and deploy serverless applications that are built using AWS. They provide an experience for building, testing, debugging, deploying, and invoking Lambda functions that's integrated into the integrated development environment (IDE).

For more information about AWS toolkits that you can use with AWS SAM, see the following:

- AWS Toolkit for JetBrains
- AWS Toolkit for PyCharm
- AWS Toolkit for IntelliJ
- AWS Toolkit for Visual Studio Code

Running AWS SAM Locally

The commands `sam local invoke` and `sam local start-api` both support local step-through debugging of your Lambda functions. To run AWS SAM locally with step-through debugging support enabled, specify `--debug-port` or `-d` on the command line. For example:

```
# Invoke a function locally in debug mode on port 5858
$ sam local invoke -d 5858 <function logical id>
```

```
# Start local API Gateway in debug mode on port 5858
$ sam local start-api -d 5858
```

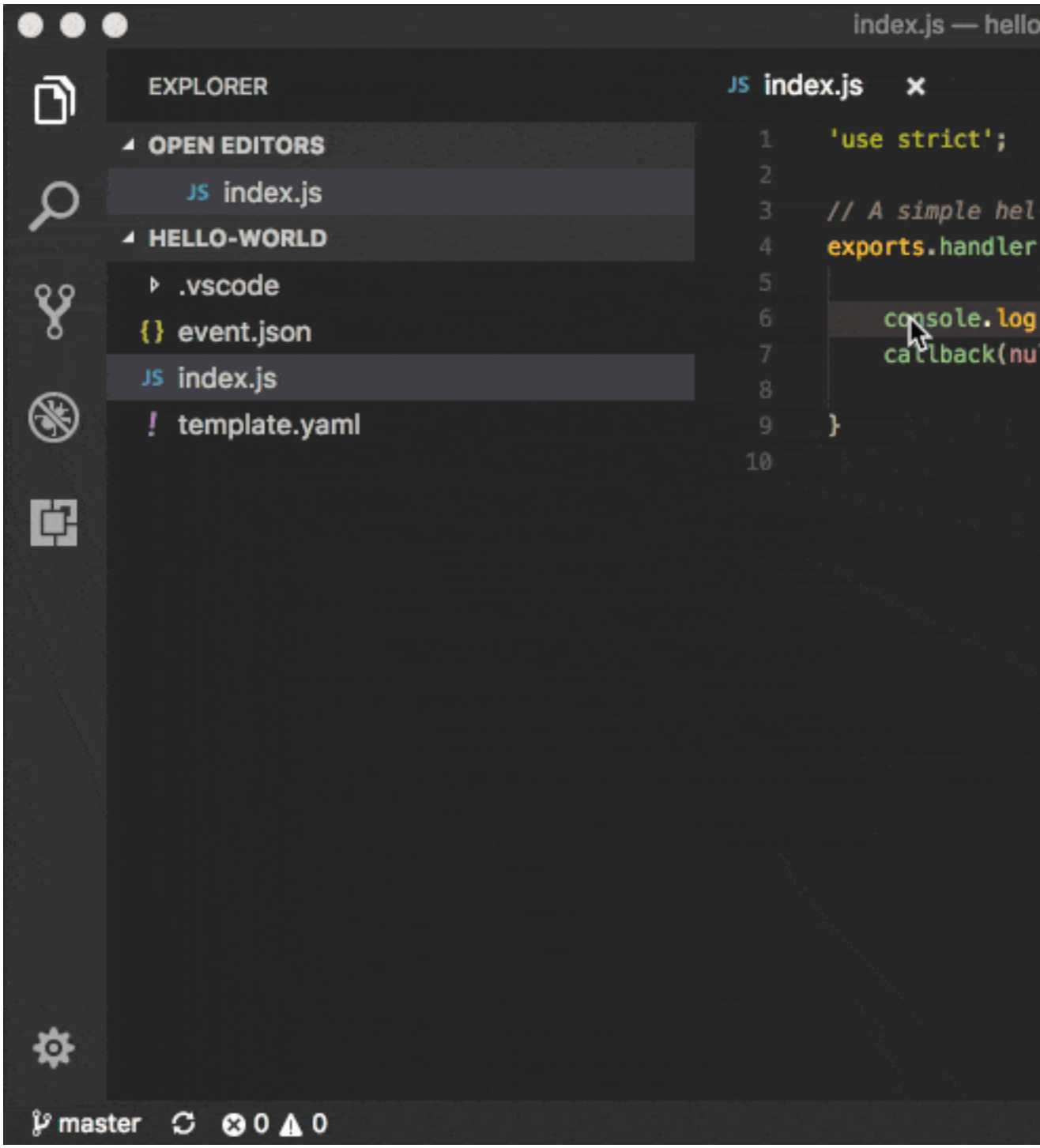
Note
If you're using `sam local start-api`, the local API Gateway instance exposes all of your Lambda functions. However, because you can specify a single debug port, you can only debug one function at a time. You need to call your API before the AWS SAM CLI binds to the port, which allows the debugger to connect.

Topics

The following topics provide examples of how to set up your environment to test and debug your serverless applications locally.

Step-Through Debugging Node.js Functions Locally

The following is an example that shows how to debug a Node.js function with Microsoft Visual Studio Code:



To set up Microsoft Visual Studio Code for step-through debugging Node.js functions with the AWS SAM CLI, use the following launch configuration. Before you do this, set the directory where the `template.yaml` file is located as the workspace root in Microsoft Visual Studio Code:

```
{
    "version": "0.2.0",
    "configurations": [
```

```
        {
            "name": "Attach to SAM CLI",
            "type": "node",
            "request": "attach",
            "address": "localhost",
            "port": 5858,
            // From the sam init example, it would be "${workspaceRoot}/hello_world"
            "localRoot": "${workspaceRoot}/{directory of node app}",
            "remoteRoot": "/var/task",
            "protocol": "inspector",
            "stopOnEntry": false
        }
    ]
}
```

Note
The `localRoot` is set based on what the CodeUri points at in the `template.yaml` file. If there are nested directories within the CodeUri, that needs to be reflected in the `localRoot`.

Note
Node.js versions earlier than 7 (for example, Node.js 4.3 and Node.js 6.10) use the `legacy` protocol, while Node.js versions including and later than 7 (for example, Node.js 8.10) use the `inspector` protocol. Be sure to specify the corresponding protocol in the `protocol` entry of your launch configuration. This was tested with Microsoft Visual Studio Code versions 1.26, 1.27, and 1.28 for the `legacy` and `inspector` protocols.

Step-Through Debugging Python Functions Locally

Python step-through debugging requires you to enable remote debugging in your Lambda function code. This is a two-step process:

1. Install the ptvsd library and enable it within your code.
2. Configure your IDE to connect to the debugger that you configured for your function.

Because this might be your first time using the AWS SAM CLI, start with a boilerplate Python application, and install both the application's dependencies and ptvsd:

```
sam init --runtime python3.6 --name python-debugging
cd python-debugging/

# Install dependencies of our boilerplate app
pip install -r requirements.txt -t hello_world/build/

# Install ptvsd library for step through debugging
pip install ptvsd -t hello_world/build/

cp hello_world/app.py hello_world/build/
```

Ptvsd Configuration

Next, you need to enable ptvsd within your code. To do this, open `hello_world/build/app.py`, and add the following ptvsd specifics:

```
import ptvsd

# Enable ptvsd on 0.0.0.0 address and on port 5890 that we'll connect later with our IDE
ptvsd.enable_attach(address=('0.0.0.0', 5890), redirect_output=True)
```

```
ptvsd.wait_for_attach()
```

Use `0.0.0.0` instead of `localhost` for listening across all network interfaces. `5890` is the debugging port that you want to use.

Microsoft Visual Studio Code

Now that you have the dependencies and ptvsd enabled within your code, you can configure Microsoft Visual Studio Code debugging. Assuming that you're still in the application folder and have the code command in your path, open Microsoft Visual Studio Code by using this command:

```
code .
```

> **Note**
> If you don't have code in your path, open a new instance of Microsoft Visual Studio Code from the `python-debugging/` folder that you created earlier.

To set up Microsoft Visual Studio Code for debugging with the AWS SAM CLI, use the following launch configuration:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "SAM CLI Python Hello World",
            "type": "python",
            "request": "attach",
            "port": 5890,
            "host": "localhost",
            "pathMappings": [
                {
                    "localRoot": "${workspaceFolder}/hello_world/build",
                    "remoteRoot": "/var/task"
                }
            ]
        }
    ]
}
```

For Microsoft Visual Studio Code, the property `localRoot` under the `pathMappings` key is important. There are two reasons that help explain this setup:

- **localRoot**: This path is to be mounted in the Docker container, and needs to have both the application and dependencies at the root level.
- **workspaceFolder**: This path is the absolute path where the Microsoft Visual Studio Code instance was opened.

If you opened Microsoft Visual Studio Code in a different location other than `python-debugging/`, you need to replace it with the absolute path where `python-debugging/` is located.

After the Microsoft Visual Studio Code debugger configuration is complete, make sure to add a breakpoint anywhere you want to in `hello_world/build/app.py`, and then proceed as follows:

1. Run the AWS SAM CLI to invoke your function.
2. Send a request to the URL to invoke the function and initialize ptvsd code execution.
3. Start the debugger within Microsoft Visual Studio Code.

```
# Remember to hit the URL before starting the debugger in Microsoft Visual Studio Code
sam local start-api -d 5890

# OR

# Change HelloWorldFunction to reflect the logical name found in template.yaml
sam local generate-event apigateway aws-proxy | sam local invoke HelloWorldFunction -d 5890
```

Step-Through Debugging Golang Functions Locally

Golang function step-through debugging is slightly different when compared to Node.js, Java, and Python. We require delve as the debugger, and wrap your function with it at runtime. The debugger is run in headless mode, listening on the debug port.

When you're debugging, you must compile your function in debug mode:

```
GOARCH=amd64 GOOS=linux go build -gcflags='-N -l' -o <output path> <path to code directory>
```

You must compile delve to run in the container and provide its local path with the `--debugger-path` argument. Build delve locally as follows:

```
GOARCH=amd64 GOOS=linux go build -o <delve folder path>/dlv github.com/derekparker/delve/
cmd/dlv
```

Note
The output path needs to end in `/dlv`. The Docker container expects the dlv binary file to be in the `<delve folder path>`. If it's not, a mounting issue occurs.

Then invoke AWS SAM similar to the following:

```
sam local start-api -d 5986 --debugger-path <delve folder path>
```

Note
The `--debugger-path` is the path to the directory that contains the dlv binary file that's compiled from the previous code.

The following is an example launch configuration for Microsoft Visual Studio Code to attach to a debug session.

```
{
  "version": "0.2.0",
  "configurations": [
  {
      "name": "Connect to Lambda container",
      "type": "go",
      "request": "launch",
      "mode": "remote",
      "remotePath": "",
      "port": <debug port>,
      "host": "127.0.0.1",
      "program": "${workspaceRoot}",
      "env": {},
      "args": [],
    },
  ]
}
```

Passing Additional Runtime Debug Arguments

To pass additional runtime arguments when you're debugging your function, use the environment variable `DEBUGGER_ARGS`. This passes a string of arguments directly into the run command that the AWS SAM CLI uses to start your function.

For example, if you want to load a debugger like iKPdb at the runtime of your Python function, you could pass the following as `DEBUGGER_ARGS: -m ikpdb --ikpdb-port=5858 --ikpdb-working-directory=/var/task/ --ikpdb-client-working-directory=/myApp --ikpdb-address=0.0.0.0`. This would load iKPdb at runtime with the other arguments you've specified.

In this case, your full AWS SAM CLI command would be:

```
$ DEBUGGER_ARGS="-m ikpdb --ikpdb-port=5858 --ikpdb-working-directory=/var/task/ --ikpdb-
client-working-directory=/myApp --ikpdb-address=0.0.0.0" echo {} | sam local invoke -d 5858
 myFunction
```

You can pass debugger arguments to the functions of all runtimes.

Validating AWS SAM Template Files

Validate your templates with `sam validate (p. 66)`. Currently, this command validates that the template provided is valid JSON / YAML. As with most AWS SAM CLI commands, it looks for a `template.[yaml|yml]` file in your current working directory by default. You can specify a different template file/location with the `-t` or `--template` option.

Example:

```
$ sam validate
<path-to-file>/template.yml is a valid SAM Template
```

Note
The `sam validate` command requires AWS credentials to be configured. For more information, see Configuration and Credential Files.

Deploying Serverless Applications

AWS SAM uses AWS CloudFormation as the underlying deployment mechanism. For more information, see What Is AWS CloudFormation?.

You can deploy your application by using AWS SAM command line interface (CLI) commands. You can also use other AWS services that integrate with AWS SAM to automate your deployments.

Packaging and Deploying Using the AWS SAM CLI

After you develop and test your serverless application locally, you can deploy your application by using the `sam package` and `sam deploy` commands.

> **Note**
> Both the `sam package` and `sam deploy` commands described in this section are identical to their AWS CLI equivalent commands `aws cloudformation package` and `aws cloudformation deploy`, respectively.

The `sam package` command zips your code artifacts, uploads them to Amazon S3, and produces a packaged AWS SAM template file that's ready to be used. The `sam deploy` command uses this file to deploy your application. For example, the following command generates a `packaged.yaml` file:

```
# Package SAM template
$ sam package --template-file sam.yaml --s3-bucket mybucket --output-template-file
 packaged.yaml
```

The following `sam deploy` command takes the packaged AWS SAM template file that was created earlier, and deploys your serverless application:

```
# Deploy packaged SAM template
$ sam deploy --template-file ./packaged.yaml --stack-name mystack --capabilities
 CAPABILITY_IAM
```

> **Note**
> To deploy an application that contains one or more nested applications, you must include the `CAPABILITY_AUTO_EXPAND` capability in the `sam deploy` command.

Publishing Serverless Applications

The AWS Serverless Application Repository is a service that hosts serverless applications that are built using AWS SAM. If you want to share serverless applications with others, you can publish them in the AWS Serverless Application Repository. You can also search, browse, and deploy serverless applications that have been published by others. For more information, see What Is the AWS Serverless Application Repository?.

Automating Deployments

You can use AWS SAM with a number of other AWS services to automate the deployment process of your serverless application.

- **CodeBuild**: You use CodeBuild to build, locally test, and package your serverless application. For more information, see What Is CodeBuild?.

- **CodeDeploy**: You use CodeDeploy to gradually deploy updates to your serverless applications. For more information on how to do this, see Gradual Code Deployment (p. 35).
- **CodePipeline**: You use CodePipeline to model, visualize, and automate the steps that are required to release your serverless application. For more information, see What Is CodePipeline?.

Topics

Gradual Code Deployment

If you use AWS SAM to create your serverless application, it comes built-in with CodeDeploy to help ensure safe Lambda deployments. With just a few lines of configuration, AWS SAM does the following for you:

- Deploys new versions of your Lambda function, and automatically creates aliases that point to the new version.
- Gradually shifts customer traffic to the new version until you're satisfied that it's working as expected, or you roll back the update.
- Defines pre-traffic and post-traffic test functions to verify that the newly deployed code is configured correctly and your application operates as expected.
- Rolls back the deployment if CloudWatch alarms are triggered.

> **Note**
> If you enable gradual deployments through your AWS SAM template, an CodeDeploy resource is automatically created for you. You can view the CodeDeploy resource directly through the AWS Management Console.

Example

The following example demonstrates a simple version of using CodeDeploy to gradually shift customers to your newly deployed version:

```
Resources:
 MyLambdaFunction:
   Type: AWS::Serverless::Function
   Properties:
     Handler: index.handler
     Runtime: nodejs4.3
     CodeUri: s3://bucket/code.zip

     AutoPublishAlias: live

     DeploymentPreference:
       Type: Canary10Percent10Minutes
       Alarms:
         # A list of alarms that you want to monitor
         - !Ref AliasErrorMetricGreaterThanZeroAlarm
         - !Ref LatestVersionErrorMetricGreaterThanZeroAlarm
       Hooks:
         # Validation Lambda functions that are run before & after traffic shifting
         PreTraffic: !Ref PreTrafficLambdaFunction
         PostTraffic: !Ref PostTrafficLambdaFunction
```

These revisions to the AWS SAM template do the following:

- **AutoPublishAlias**: By adding this property and specifying an alias name, AWS SAM:
  - Detects when new code is being deployed, based on changes to the Lambda function's Amazon S3 URI.
  - Creates and publishes an updated version of that function with the latest code.
  - Creates an alias with a name that you provide (unless an alias already exists), and points to the updated version of the Lambda function. Function invocations should use the alias qualifier to take advantage of this. If you aren't familiar with Lambda function versioning and aliases, see AWS Lambda Function Versioning and Aliases .
- **Deployment Preference Type**: In the previous example, 10 percent of your customer traffic is immediately shifted to your new version. After 10 minutes, all traffic is shifted to the new version. However, if your pre-hook/post-hook tests fail, or if a CloudWatch alarm is triggered, CodeDeploy rolls back your deployment. The following table outlines other traffic-shifting options that are available beyond the one used earlier. Note the following:
  - **Canary**: Traffic is shifted in two increments. You can choose from predefined canary options. The options specify the percentage of traffic that's shifted to your updated Lambda function version in the first increment, and the interval, in minutes, before the remaining traffic is shifted in the second increment.
  - **Linear**: Traffic is shifted in equal increments with an equal number of minutes between each increment. You can choose from predefined linear options that specify the percentage of traffic that's shifted in each increment and the number of minutes between each increment.
  - **All-at-once**: All traffic is shifted from the original Lambda function to the updated Lambda function version at once.

Deployment Preference Type
Canary10Percent30Minutes
Canary10Percent5Minutes
Canary10Percent10Minutes
Canary10Percent15Minutes
Linear10PercentEvery10Minutes
Linear10PercentEvery1Minute
Linear10PercentEvery2Minutes
Linear10PercentEvery3Minutes
AllAtOnce

- **Alarms**: These are CloudWatch alarms that are triggered by any errors raised by the deployment. They automatically roll back your deployment. An example is if the updated code you're deploying is creating errors within the application. Another example is if any AWS Lambda or custom CloudWatch metrics that you specified have breached the alarm threshold.
- **Hooks**: These are pre-traffic and post-traffic test functions that run sanity checks before traffic shifting starts to the new version, and after traffic shifting completes.
  - **PreTraffic**: Before traffic shifting starts, CodeDeploy invokes the pre-traffic hook Lambda function. This Lambda function must call back to CodeDeploy and indicate success or failure. If the function fails, it aborts and reports a failure back to AWS CloudFormation. If the function succeeds, CodeDeploy proceeds to traffic shifting.
  - **PostTraffic**: After traffic shifting completes, CodeDeploy invokes the post-traffic hook Lambda function. This is similar to the pre-traffic hook, where the function must call back to CodeDeploy to report a success or failure. Use post-traffic hooks to run integration tests or other validation actions.

For more information, see SAM Reference to Safe Deployments.

Publishing Serverless Applications Using the AWS SAM CLI

You can use the AWS SAM CLI to publish your application to the AWS Serverless Application Repository to make it available for others to find and deploy.

The application you want to publish must be one that you've defined using AWS SAM, and that you've tested locally and/or in the AWS Cloud. The application's deployment package and AWS SAM template are the inputs to the steps below.

The following instructions either create a new application, create a new version of an existing application, or update the metadata of an existing application. This depends on whether the application already exists in the AWS Serverless Application Repository, and whether any application metadata is changing. For more information about application metadata that's used to publish applications, see AWS SAM Template Metadata Section Properties (p. 40).

Prerequisites

Before you publish an application to the AWS Serverless Application Repository, you need the following:

- A valid AWS account with an IAM user that has administrator permissions. See Set Up an AWS Account.
- Version 1.16.77 or later of the AWS CLI installed. See Installing the AWS Command Line Interface.
- The AWS SAM CLI (command line interface) installed. See Installing the AWS SAM CLI.
- A valid AWS Serverless Application Model (AWS SAM) template.
- Your application code and dependencies referenced by the AWS SAM template.
- A semantic version for your application (which is needed to share your application publicly).
- A URL that points to your application's source code.
- A README.md file. This file should describe how customers can use your application, and how to configure it before deploying it in their own AWS accounts.
- A LICENSE.txt file.
- A valid Amazon S3 bucket policy that grants the service read permissions for artifacts uploaded to Amazon S3 when you packaged your application. To do this, follow these steps:

  1. Open the Amazon S3 console at https://console.aws.amazon.com/s3/.
  2. Choose the Amazon S3 bucket that you used to package your application.
  3. Choose the **Permissions** tab.
  4. Choose the **Bucket Policy** button.
  5. Paste the example policy statement below. Make sure to substitute your bucket name in the Resource property value.
  6. Choose the **Save** button.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service":  "serverlessrepo.amazonaws.com"
            },
```

```
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::<your-bucket-name>/*"
        }
    ]
}
```

Step 1: Add a Metadata Section to the AWS SAM Template

First add a `Metadata` section to your AWS SAM template. Provide the application information to be published to the AWS Serverless Application Repository.

The following is an example `Metadata` section:

```
Metadata:
  AWS::ServerlessRepo::Application:
    Name: my-app
    Description: hello world
    Author: user1
    SpdxLicenseId: Apache-2.0
    LicenseUrl: LICENSE.txt
    ReadmeUrl: README.md
    Labels: ['tests']
    HomePageUrl: https://github.com/user1/my-app-project
    SemanticVersion: 0.0.1
    SourceCodeUrl: https://github.com/user1/my-app-project

Resources:
  HelloWorldFunction:
    Type: AWS::Lambda::Function
      Properties:
        ...
        CodeUri: source-code1
        ...
```

For more information about the properties of the `Metadata` section in the AWS SAM template, see AWS SAM Template Metadata Section Properties (p. 40).

Step 2: Package the Application

Execute the following AWS SAM CLI command:

```
sam-app> sam package \
    --template-file template.yaml \
    --output-template-file packaged.yaml \
    --s3-bucket <your-bucket-name>
```

The command uploads the application artifacts to Amazon S3 and outputs a new template file called `packaged.yaml`. You use this file in the next step to publish the application to the AWS Serverless Application Repository. The `packaged.yaml` template file is similar to the original template file (`template.yaml`), but has a key difference—the `CodeUri`, `LicenseUrl`, and `ReadmeUrl` properties point to the Amazon S3 bucket and objects that contain the respective artifacts.

The following snippet from an example `packaged.yaml` template file shows the `CodeUri` property:

```
MySampleFunction:
```

```
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://bucketname/fbd77a3647a4f47a352fcObjectGUID

...
```

Step 3: Publish the Application

Execute the following AWS SAM CLI command:

```
sam-app> sam publish \
    --template packaged.yaml \
    --region us-east-1
```

The output of the `sam publish` command includes a link to the AWS Serverless Application Repository directly to your application. You can also go to the AWS Serverless Application Repository landing page directly and search for your application.

Your application is set to private by default, so it isn't visible to other AWS accounts. In order to share your application with others, you must either make it public or grant permission to a specific list of AWS accounts. For information on sharing your application by using the AWS CLI, see Using Resource-based Policies for the AWS Serverless Application Repository. For information on sharing your application using the console, see Sharing an Application Through the Console.

Additional Topics

- AWS SAM Template Metadata Section Properties (p. 40)

AWS SAM Template Metadata Section Properties

This table provides information about the properties of the `Metadata` section of the AWS SAM template. The template is needed to publish applications to the AWS Serverless Application Repository using the AWS SAM CLI.

Property	Type	Required	Description
Name	String	TRUE	The name of the application. Minimum length=1. Maximum length=140. Pattern: "[a-zA-Z0-9\\-]+";
Description	String	TRUE	The description of the application. Minimum length=1. Maximum length=256.
Author	String	TRUE	The name of the author publishing the application. Minimum length=1. Maximum length=127. Pattern "^[a-z0-9](([a-z0-9]|-(?!-))*[a-z0-9])?$";
SpdxLicenseId	String	TRUE	A valid license identifier: https://spdx.org/licenses/

Property	Type	Required	Description
LicenseUrl	String	TRUE	Reference to a local license file, or an Amazon S3 link to a license file, of the application that matches the spdxLicenseID value of your application. An AWS SAM template file that has not been packaged using the `sam package` command can have a reference to a local file for this property. However, to be published using the `sam publish` command, this property must be a reference to an Amazon S3 bucket. Maximum size: 5 MB.
ReadmeUrl	String	FALSE	Reference to a local readme file, or an Amazon S3 link to the readme file that contains a more detailed description of the application and how it works. An AWS SAM template file that has not been packaged using the `sam package` command can have a reference to a local file for this property. However, to be published using the `sam publish` command, this property must be a reference to an Amazon S3 bucket. Maximum size: 5 MB.
Labels	String	FALSE	Labels to improve discovery of applications in search results. Minimum length=1. Maximum length=127. Maximum number of labels: 10. Pattern: "^[a-zA-Z0-9+\\-_:\\/@]+$";
HomePageUrl	String	FALSE	A URL with more information about the application—for example, the location of your GitHub repository for the application.
SemanticVersion	String	FALSE	The semantic version of the application: https://semver.org/ You must provide a value for this property in order to make your application public.
SourceCodeUrl	String	FALSE	A link to a public repository for the source code of your application.

Use Cases

The use cases for publishing applications are listed below, along with the `Metadata` properties that are processed for that use case. Properties that are *not* listed for a given use case are ignored.

- **Create New Application** – A new application is created if there is no application in the AWS Serverless Application Repository with a matching name for an account.
  - Name
  - SpdxLicenseId
  - LicenseUrl

  - Description
  - Author
  - ReadmeUrl
  - Labels
  - HomePageUrl
  - SourceCodeUrl
  - SemanticVersion
  - The content of the SAM template (for example, any event sources, resources, Lambda function code, etc.)

- **Create Application Version** – An application version is created if there is already an application in the AWS Serverless Application Repository with a matching name for an account *and* the SemanticVersion *is* changing.
  - Description
  - Author
  - ReadmeUrl
  - Labels
  - HomePageUrl
  - SourceCodeUrl
  - SemanticVersion
  - The content of the SAM template (for example, any Event Sources, Resources, Lambda function code, etc.)

- **Update Application** – An application is updated if there is already an application in the AWS Serverless Application Repository with a matching name for an account *and* the SemanticVersion *is not* changing.
  - Description
  - Author
  - ReadmeUrl
  - Labels
  - HomePageUrl

Example Serverless Applications

The following examples show you how to download, test, and deploy a number of additional serverless applications—including how to configure event sources and AWS resources.

Topics

Process DynamoDB Events

With this example application, you build on what you learned in the overview and the Quick Start guide, and install another example application. This application consists of a Lambda function that's invoked by a DynamoDB table event source. The Lambda function is very simple—it logs data that was passed in through the event source message.

This exercise shows you how to mimic event source messages that are passed to Lambda functions when they're invoked.

Before You Begin

Make sure that you've completed the required setup in the Quick Start (p. 3). See the Before You Begin (p. 3) section of the Quick Start.

Step 1: Initialize the Application

In this section, you download the application package, which consists of an AWS SAM template and application code.

To initialize the application

1. Run the following command at an AWS SAM CLI command prompt.

```
sam init \
--location gh:aws-samples/cookiecutter-aws-sam-dynamodb-python \
--no-input
```

2. Review the contents of the directory that the command created (`dynamodb_event_reader/`):

   - `template.yaml` – Defines two AWS resources that the Read DynamoDB application needs: a Lambda function and a DynamoDB table. The template also defines mapping between the two resources.
   - `read_dynamodb_event/` directory – Contains the DynamoDB application code.

Step 2: Test the Application Locally

For local testing, use the AWS SAM CLI to generate a sample DynamoDB event and invoke the Lambda function:

```
$ sam local generate-event dynamodb update | sam local invoke ReadDynamoDBEvent
```

The `generate-event` command creates a test event source message like the messages that are created when all components are deployed to the AWS Cloud. This event source message is piped to the Lambda function ReadDynamoDBEvent.

Verify that the expected messages are printed to the console, based on the source code in `app.py`.

Step 3: Package the Application

After testing your application locally, you use the AWS SAM CLI to create a deployment package, which you use to deploy the application to the AWS Cloud.

To create a Lambda deployment package

1. Create an S3 bucket in the location where you want to save the packaged code. If you want to use an existing S3 bucket, skip this step.

   ```
   sam-app> aws s3 mb s3://bucketname
   ```

2. Create the deployment package by running the following `package` CLI command at the command prompt.

   ```
   sam-app> sam package \
       --template-file template.yaml \
       --output-template-file packaged.yaml \
       --s3-bucket bucketname
   ```

   You specify the new template file, `packaged.yaml`, when you deploy the application in the next step.

Step 4: Deploy the Application

Now that you've created the deployment package, you use it to deploy the application to the AWS Cloud. You then test the application.

To deploy the serverless application to the AWS Cloud

- In the AWS SAM CLI, use the `deploy` CLI command to deploy all of the resources that you defined in the template.

  ```
  sam-app> sam deploy \
      --template-file packaged.yaml \
      --stack-name sam-app \
      --capabilities CAPABILITY_IAM \
      --region us-east-1
  ```

  In the command, the `--capabilities` parameter allows AWS CloudFormation to create an IAM role.

  AWS CloudFormation creates the AWS resources that are defined in the template. You can access the names of these resources in the AWS CloudFormation console.

To test the serverless application in the AWS Cloud

1. Open the DynamoDB console.

2. Insert a record into the table that you just created.
3. Go to the **Metrics** tab of the table, and choose **View all CloudWatch metrics**. In the CloudWatch console, choose **Logs** to be able to view the log output.

Process Amazon S3 Events

With this example application, you build on what you learned in the previous examples, and install a more complex application. This application consists of a Lambda function that's invoked by an Amazon S3 object upload event source. This exercise shows you how to access AWS resources and make AWS service calls through a Lambda function.

This sample serverless application processes object-creation events in Amazon S3. For each image that's uploaded to a bucket, Amazon S3 detects the object-created event and invokes a Lambda function. The Lambda function invokes Amazon Rekognition to detect text that's in the image. It then stores the results returned by Amazon Rekognition in a DynamoDB table.

> **Note**
> With this example application, you perform steps in a slightly different order than in previous examples. The reason for this is that this example requires that AWS resources are created and IAM permissions are configured *before* you can test the Lambda function locally. We're going to leverage AWS CloudFormation to create the resources and configure the permissions for you. Otherwise, you would need to do this manually before you can test the Lambda function locally. Because this example is more complicated, be sure that you're familiar with installing the previous example applications before executing this one.

Before You Begin

Make sure that you've completed the required setup in the Quick Start (p. 3). See the Before You Begin (p. 3) section of the Quick Start.

Step 1: Initialize the Application

In this section, you download the sample application, which consists of an AWS SAM template and application code.

To initialize the application

1. Run the following command at an AWS SAM CLI command prompt.

```
sam init \
--location https://github.com/aws-samples/cookiecutter-aws-sam-s3-rekognition-dynamodb-python \
--no-input
```

2. Review the contents of the directory that the command created (`aws_sam_ocr/`):

   - `template.yaml` – Defines three AWS resources that the Amazon S3 application needs: a Lambda function, an Amazon S3 bucket, and a DynamoDB table. The template also defines the mappings and permissions between these resources.
   - `src/` directory – Contains the Amazon S3 application code.
   - `SampleEvent.json` – The sample event source, which is used for local testing.

Step 2: Package the Application

Before you can test this application locally, you must use the AWS SAM CLI to create a deployment package, which you use to deploy the application to the AWS Cloud. This deployment creates the necessary AWS resources and permissions that are required to test the application locally.

To create a Lambda deployment package

1. Create an S3 bucket in the location where you want to save the packaged code. If you want to use an existing S3 bucket, skip this step.

   ```
   aws_sam_ocr> aws s3 mb s3://bucketname
   ```

2. Create the deployment package by running the following `package` CLI command at the command prompt.

   ```
   aws_sam_ocr> sam package \
       --template-file template.yaml \
       --output-template-file packaged.yaml \
       --s3-bucket bucketname
   ```

   You specify the new template file, `packaged.yaml`, when you deploy the application in the next step.

Step 3: Deploy the Application

Now that you've created the deployment package, you use it to deploy the application to the AWS Cloud. You then test the application by invoking it in the AWS Cloud.

To deploy the serverless application to the AWS Cloud

- In the AWS SAM CLI, use the `deploy` command to deploy all of the resources that you defined in the template.

  ```
  aws_sam_ocr> sam deploy \
      --template-file packaged.yaml \
      --stack-name aws-sam-ocr \
      --capabilities CAPABILITY_IAM \
      --region us-east-1
  ```

  In the command, the `--capabilities` parameter allows AWS CloudFormation to create an IAM role.

  AWS CloudFormation creates the AWS resources that are defined in the template. You can access the names of these resources in the AWS CloudFormation console.

To test the serverless application in the AWS Cloud

1. Upload an image to the Amazon S3 bucket that you created for this sample application.
2. Open the DynamoDB console and find the table that was created. See the table for results returned by Amazon Rekognition.
3. Verify that the DynamoDB table contains new records that contain text that Amazon Rekognition found in the uploaded image.

Step 4: Test the Application Locally

Before you can test the application locally, you must first retrieve the names of the AWS resources that were created by AWS CloudFormation.

- Retrieve the Amazon S3 key name and bucket name from AWS CloudFormation. Modify the `SampleEvent.json` file by replacing the values for the object key, bucket name, and bucket ARN.
- Retrieve the DynamoDB table name. This name is used for the following `sam local invoke` command.

Use the AWS SAM CLI to generate a sample Amazon S3 event and invoke the Lambda function:

```
$ TABLE_NAME=Table name obtained from AWS CloudFormation console sam local invoke --event
 SampleEvent.json
```

The `TABLE_NAME=` portion sets the DynamoDB table name. The `--event` parameter specifies the file that contains the test event message to pass to the Lambda function.

You can now verify that the expected DynamoDB records were created, based on the results returned by Amazon Rekognition.

AWS SAM Reference

AWS SAM Specification

The AWS SAM specification is an open-source specification under the Apache 2.0 license. The current version of the AWS SAM specification is available in the AWS SAM GitHub repo.

AWS SAM templates are an extension of AWS CloudFormation templates. That is, any resource that you can declare in an AWS CloudFormation template, you can also declare in an AWS SAM template. For the full reference for AWS CloudFormation templates, see AWS CloudFormation Template Reference.

AWS SAM CLI

The **AWS SAM CLI** is a command line tool that operates on an AWS SAM template and application code. With the AWS SAM CLI, you can invoke Lambda functions locally, create a deployment package for your serverless application, deploy your serverless application to the AWS Cloud, and so on.

You can use the AWS SAM CLI commands to develop, test, and deploy your serverless applications to the AWS Cloud. The following are some examples of AWS SAM CLI commands:

- `sam init` – If you're a first-time AWS SAM CLI user, you can run the `sam init` command without any parameters to create a Hello World application. The command generates a preconfigured AWS SAM template and example application code in the language that you choose.
- `sam local invoke` and `sam local start-api` – Use these commands to test your application code locally, before deploying it to the AWS Cloud.
- `sam logs` – Use this command to fetch logs generated by your Lambda function to help with testing and debugging your application after you've deployed it to the AWS Cloud.
- `sam package` – Use this command to bundle your application code and dependencies into a "deployment package" that's needed in order to upload to the AWS Cloud.
- `sam deploy` – Use this command to deploy your serverless application to the AWS Cloud. It creates the AWS resources and sets permissions and other configurations that are defined in the AWS SAM template.

Topics

Installing the AWS SAM CLI

You can install the AWS SAM CLI on Linux, Windows, and macOS. For detailed instructions on how to install the AWS SAM CLI on each platform, see the topics listed after the prerequisites.

Prerequisites

Before installing the AWS SAM CLI, you need the following:

- AWS Command Line Interface (AWS CLI). For instructions on installing the AWS CLI, see Installing the AWS Command Line Interface.
- Docker. For instructions on installing Docker, see the platform-specific topic for each platform in the following list.

Topics

Installing the AWS SAM CLI on Linux

To install the AWS SAM CLI on Linux, first make sure that you've installed the AWS Command Line Interface (AWS CLI) and Docker for Linux (p. 49).

Docker for Linux

You need to have Docker installed and running to be able to run serverless projects and functions locally with the AWS SAM CLI. The AWS SAM CLI uses the `DOCKER_HOST` environment variable to contact the Docker daemon.

To install Docker on Linux, see About Docker CE. In the left-hand column, choose **Linux**, choose your Linux distribution (for example, **CentOS**, **Debian**, or **Ubuntu**), and follow the installation instructions.

Verify that Docker is working, and that you can run Docker commands from the AWS SAM CLI (for example, `docker ps`). You don't need to install/fetch/pull any containers because the AWS SAM CLI does it automatically, as required.

Install the AWS SAM CLI Using Linuxbrew

Follow these steps to install the AWS SAM CLI by using Linuxbrew:

1. To install the Linuxbrew package manager, follow the instructions on the Linuxbrew website.
2. Upgrade Linuxbrew, and update it to the latest version.

   ```
   brew upgrade
   brew update
   ```

3. Add a brew tap from GitHub.

   ```
   brew tap aws/tap
   ```

4. Install aws-sam-cli from the brew tap.

   ```
   brew install aws-sam-cli
   ```

Now sam is installed to the following location:

```
/home/linuxbrew/.linuxbrew/bin/sam
```

You should be able to invoke sam from the command line.

```
sam --version
```

Install the AWS SAM CLI Using Pip

An alternate method of installing the AWS SAM CLI is by using pip. For details on how to do this, see Installing Using Pip (p. 52).

Installing the AWS SAM CLI on Windows

To install the AWS SAM CLI on Windows, first make sure that you've installed the AWS Command Line Interface (AWS CLI) and Docker for Windows (p. 50).

Docker for Windows

You need to have Docker installed and running to be able to run serverless projects and functions locally with the AWS SAM CLI. The AWS SAM CLI uses the `DOCKER_HOST` environment variable to contact the Docker daemon.

To install Docker for Windows, see Docker For Windows.

> **Note**
> For Windows users: The AWS SAM CLI requires that the project directory (or any parent directory) is listed in Docker Shared Drives.

Verify that Docker is working, and that you can run Docker commands from the AWS SAM CLI (for example, `docker ps`). You don't need to install/fetch/pull any containers because the AWS SAM CLI does it automatically, as required.

Install the AWS SAM CLI Using the MSI File

Install the MSI file from one of these locations:

- 64-bit
- 32-bit

After completing the installation by using one of these links, open a new command prompt or PowerShell prompt. You should be able to invoke sam from the command line.

```
sam --version
```

Install the AWS SAM CLI Using Pip

An alternate method of installing the AWS SAM CLI is using pip. For details on how to do this, see Installing Using Pip (p. 52).

Troubleshooting

Python Not Installed

If you get an error similar to the following, then you may not have Python installed.

```
'"' is not recognized as an internal or external command, operable program or batch file.
```

Make sure that you install Python, and try again. To install Python for Windows see Python Releases for Windows.

Installing the AWS SAM CLI on macOS

To install the AWS SAM CLI on macOS, first make sure that you've installed the AWS Command Line Interface (AWS CLI) and Docker for macOS (p. 51).

Docker for macOS

You need to have Docker installed and running to be able to run serverless projects and functions locally with the AWS SAM CLI. The AWS SAM CLI uses the `DOCKER_HOST` environment variable to contact the Docker daemon.

To install Docker on macOS, see Docker for macOS.

> **Note**
> For macOS users: The AWS SAM CLI requires that the project directory (or any parent directory) is listed in Docker File System Sharing.

Verify that Docker is working, and that you can run Docker commands from the AWS SAM CLI (for example, `docker ps`). You don't need to install/fetch/pull any containers because the AWS SAM CLI does it automatically, as required.

Install the AWS SAM CLI Using Homebrew

Follow these steps to install the AWS SAM CLI by using Homebrew:

1. To install the Homebrew package manager, follow the instructions on the Homebrew website .
2. Add a brew tap from GitHub.

```
brew tap aws/tap
```

3. Install aws-sam-cli from the brew tap.

```
brew install aws-sam-cli
```

Now sam is installed to following location:

```
/usr/local/bin/sam
```

You should be able to invoke sam from the command line.

```
sam --version
```

Install the AWS SAM CLI Using Pip

An alternate method of installing the AWS SAM CLI is by using pip. For details on how to do this, see Installing Using Pip (p. 52).

Troubleshooting

TLSV1_ALERT_PROTOCOL_VERSION

If you get an error similar to the following, then you're probably using the default version of Python that came with your macOS. This is outdated.

```
Could not fetch URL https://pypi.python.org/simple/click/: There was a problem confirming
 the ssl certificate: [SSL: TLSV1_ALERT_PROTOCOL_VERSION] tlsv1 alert protocol version
 (_ssl.c:590) - skipping
```

Make sure that you install Python again using Homebrew, and try again:

```
$ brew install python
```

After it's installed, repeat the installation process.

Additional Installation Topics

The topics listed below contain additional information about installing the AWS SAM CLI.

Installing Using Pip

To install the AWS SAM CLI using pip, first make sure that you've installed the AWS Command Line Interface (AWS CLI) and Docker for the platform you're using:

- Docker for Linux (p. 49)
- Docker for Windows (p. 50)
- Docker for macOS (p. 51)

Follow these steps to install the AWS SAM CLI by using pip:

1. Verify that the Python version is 2.7 or 3.6.

   ```
   $ python --version
   ```

   If it isn't installed, download and install Python.
2. Verify that pip is installed.

   ```
   $ pip --version
   ```

   If it isn't installed, download and install pip.
3. Install `aws-sam-cli`.

   ```
   pip install --user aws-sam-cli
   ```
4. Adjust your `PATH` to include the Python scripts that are installed under the user's home directory.

   - **Linux:** Adjusting Your Path on Linux (p. 54)
   - **Windows:** Adjusting Your Path on Windows (p. 54)
   - **macOS:** Adjusting Your Path on macOS (p. 55)
5. Verify that `sam` is installed.

   Restart or open a new terminal, and verify that the installation worked.

   ```
   # Restart current shell
   $ sam --version
   ```

Upgrading

You can upgrade `sam` by using pip:

```
$ pip install --user --upgrade aws-sam-cli
```

First uninstall previous versions of the AWS SAM CLI (0.2.11 or earlier). Then follow the earlier installation steps. To uninstall, use this command:

```
$ npm uninstall -g aws-sam-local
```

Building from Source

To build from source, first install Python (2.7 or 3.6) on your machine, and then run the following:

```
# Clone the repository
$ git clone git@github.com:awslabs/aws-sam-cli.git

# cd into the git
$ cd aws-sam-cli

# pip install the repository
$ pip install --user -e .
```

Installing with PyEnv

To install with PyEnv, run the following commands:

```
# Install PyEnv (https://github.com/pyenv/pyenv#installation)
$ brew update
$ brew install pyenv

# Initialize pyenv using bash_profile
$ echo -e 'if command -v pyenv 1>/dev/null 2>&1; then\n  eval "$(pyenv init -)"\nfi\nexport
 PATH="~/.pyenv/bin:$PATH"' >> ~/.bash_profile
# or using zshrc
$ echo -e 'if command -v pyenv 1>/dev/null 2>&1; then\n  eval "$(pyenv init -)"\nfi\nexport
 PATH="~/.pyenv/bin:$PATH"' >> ~/.zshrc

# restart the shell
$ exec "$SHELL"

# Install Python 2.7
$ pyenv install 2.7.14
$ pyenv local 2.7.14

# Install the CLI
$ pip install --user aws-sam-cli

# Verify your installation worked
$ sam --version
```

Updating on AWS Cloud9

If your AWS Cloud9 environment has an AWS SAM CLI version earlier than 0.3.0 installed, there are a few extra steps that you must do to upgrade to newer versions:

```
# Uninstall the older version of SAM Local
$ npm uninstall -g aws-sam-local

# Remove the symlink
$ rm -rf $(which sam)

# Install the CLI
$ pip install --user aws-sam-cli

# Create new symlink
$ ln -sf $(which sam) ~/.c9/bin/sam

# Reset the bash cache
$ hash -r

# Verify your installation worked
$ sam --version
```

Adjusting Your Path on Linux

This section describes how you can adjust your path. You have to do this as part of installing the AWS SAM CLI by using pip, as described in Installing Using Pip (p. 52).

On Linux systems, the command `python -m site --user-base` typically prints `~/.local` path, so you need to add `/bin` to obtain the script path.

> **Note**
> As explained in the Python Developer's Guide, the user's home directory (where the scripts are installed) is `~/.local/bin` for Linux.

```
# Find your Python User Base path (where Python --user will install packages/scripts)
$ USER_BASE_PATH=$(python -m site --user-base)

# Update your preferred shell configuration
-- Standard bash --> ~/.bash_profile
-- ZSH           --> ~/.zshrc
$ export PATH=$PATH:$USER_BASE_PATH/bin
```

Restart or open a new terminal, and verify that the installation worked:

```
# Restart current shell
$ exec "$SHELL"
$ sam --version
```

Adjusting Your Path on Windows

This section describes how you can adjust your path. You have to do this as part of installing the AWS SAM CLI by using pip, as described in Installing Using Pip (p. 52).

On Windows systems, the command `py -m site --user-site` typically prints `%APPDATA%\Roaming\Python<VERSION>\site-packages`, so you need to remove the last `\site-packages` folder, and replace it with the `\Scripts` folder.

```
$ python -m site --user-base
```

Using File Explorer, go to the folder that's indicated in the output, and look for the `Scripts` folder. Visually confirm that the `sam` application is inside this folder.

Copy the file path.

> **Note**
> As explained in the Python Developer's Guide, the user's home directory (where the scripts are installed) is `%APPDATA%\Python\Scripts` for Windows.

Search Windows for `Edit the system environment variables`.

Choose **Environment Variables**.

Under **System Variables**, choose **Path**.

Choose **New**, and enter the file path to the Python Scripts folder.

Adjusting Your Path on macOS

This section describes how you can adjust your path. You have to do this as part of installing the AWS SAM CLI by using pip, as described in Installing Using Pip (p. 52).

On macOS systems, the command `python -m site --user-base` typically prints `~/.local` path, so you need to add `/bin` to obtain the script path.

> **Note**
> As explained in the Python Developer's Guide, the user's home directory (where the scripts are installed) is `~/.local/bin` for macOS.

```
# Find your Python User Base path (where Python --user will install packages/scripts)
$ USER_BASE_PATH=$(python -m site --user-base)

# Update your preferred shell configuration
-- Standard bash --> ~/.bash_profile
-- ZSH           --> ~/.zshrc
$ export PATH=$PATH:$USER_BASE_PATH/bin
```

Restart or open a new terminal, and verify that the installation worked:

```
# Restart current shell
$ exec "$SHELL"
$ sam --version
```

AWS SAM CLI Command Reference

This section is the reference for the AWS SAM CLI commands (version 0.8.0).

Topics

- sam publish (p. 66)
- sam validate (p. 66)

sam build

Use this command to build your Lambda source code and generate deployment artifacts that target Lambda's execution environment. By doing this, the functions that you build locally run in a similar environment in the AWS Cloud.

The `sam build` command iterates through the functions in your application, looks for a manifest file (such as `requirements.txt`) that contains the dependencies, and automatically creates deployment artifacts that you can deploy to Lambda using the `sam package` and `sam deploy` commands. You that can also use `sam build` in combination with other commands like `sam local invoke` to test your application locally.

To use this command, update your AWS SAM template to specify the path to your function's source code in the resource's Code or CodeUri property.

Usage:

```
sam build [OPTIONS]
```

Examples:

```
To build on your workstation, run this command in folder containing
SAM template. Built artifacts will be written to .aws-sam/build folder
$ sam build

To build inside a AWS Lambda like Docker container
$ sam build --use-container

To build & run your functions locally
$ sam build && sam local invoke

To build and package for deployment
$ sam build && sam package --s3-bucket <bucketname>
```

Options:

Option	Description
-b, --build-dir DIRECTORY	The path to a folder where the built artifacts are stored.
-s, --base-dir DIRECTORY	Resolves relative paths to the function's source code with respect to this folder. Use this if the AWS SAM template and your source code aren't in the same enclosing folder. By default, relative paths are resolved with respect to the template's location.
-u, --use-container	If your functions depend on packages that have natively compiled dependencies, use this flag to build your function inside an AWS Lambda-like Docker container.
-m, --manifest PATH	The path to a custom dependency manifest (ex: package.json) to use instead of the default one.
-t, --template PATH	The AWS SAM template file [default: template.[yaml|yml]].

Option	Description
--parameter-overrides	Optional. A string that contains AWS CloudFormation parameter overrides, encoded as key-value pairs. Use the same format as the AWS CLI—for example, 'ParameterKey=KeyPairName, ParameterValue=MyKey ParameterKey=InstanceType, ParameterValue=t1.micro'.
--skip-pull-image	Specifies whether the command should skip pulling down the latest Docker image for Lambda runtime.
--docker-network TEXT	Specifies the name or id of an existing Docker network to Lambda Docker containers should connect to, along with the default bridge network. If not specified, the Lambda containers will only connect to the default bridge Docker network.
--profile TEXT	Selects a specific profile from your credential file to get AWS credentials.
--region TEXT	Sets the AWS Region of the service (for example, us-east-1).
--debug	Turns on debug logging to print debug message generated by the AWS SAM CLI.
--help	Shows this message and exits.

sam deploy

Deploys an AWS SAM application. This is an alias for `aws cloudformation deploy`.

Usage:

```
sam deploy [OPTIONS] [ARGS]...
```

Options:

Option	Description
--template-file PATH	Required. The path where your AWS SAM template file is located.
--stack-name TEXT	Required. The name of the AWS CloudFormation stack you're deploying to. If you specify an existing stack, the command updates the stack. If you specify a new stack, the command creates it.
--profile TEXT	Select a specific profile from your credential file to get AWS credentials.
--region TEXT	Sets the AWS Region of the service (for example, us-east-1).
--debug	Turns on debug logging.
--help	Shows this message and exits.

sam init

Initializes a serverless application with an AWS SAM template. The template provides a folder structure for your Lambda functions, and is connected to an event source such as APIs, S3 buckets, or DynamoDB tables. This application includes everything you need to get started and to eventually extend it into a production-scale application.

Usage:

```
sam init [OPTIONS]
```

Examples:

```
Initializes a new SAM project using Python 3.6 default template runtime

$ sam init --runtime python3.6

Initializes a new SAM project using custom template in a Git/Mercurial repository

# gh being expanded to github url
$ sam init --location gh:aws-samples/cookiecutter-aws-sam-python

$ sam init --location git+ssh://git@github.com/aws-samples/cookiecutter-aws-sam-python.git

$ sam init --location hg+ssh://hg@bitbucket.org/repo/template-name

$ sam init --location hg+ssh://hg@bitbucket.org/repo/template-name

Initializes a new SAM project using custom template in a Zipfile

$ sam init --location /path/to/template.zip

$ sam init --location https://example.com/path/to/template.zip

Initializes a new SAM project using custom template in a local path

$ sam init --location /path/to/template/folder
```

Options:

Option	Description
-l, --location TEXT	The template location (Git, Mercurial, HTTP/HTTPS, ZIP, path).
-r, --runtime [python3.7| python3.6| python2.7| python| ruby2.5| nodejs6.10| nodejs8.10| nodejs| dotnetcore2.0| dotnetcore2.1| dotnetcore1.0| dotnetcore| dotnet| go1.x| go| java8| java]	The Lambda runtime of your application.
-o, --output-dir PATH	The location where the initialized application is output.
-n, --name TEXT	The name of your project to be generated as a folder.
--no-input	Disables prompting and accepts default values that are defined in the template configuration.
--profile TEXT	Select a specific profile from your credential file to get AWS credentials.
--region TEXT	Sets the AWS Region of the service (for example, us-east-1).
--debug	Turns on debug logging.
-h, --help	Shows this message and exits.

sam local generate-event

Generates sample payloads from different event sources, such as Amazon S3, Amazon API Gateway, and Amazon SNS. These payloads contain the information that the event sources send to your Lambda functions.

Usage:

```
sam local generate-event [OPTIONS] COMMAND [ARGS]...
```

Examples:

```
Generate the event that S3 sends to your Lambda function when a new object is uploaded
$ sam local generate-event s3 [put/delete]

You can even customize the event by adding parameter flags. To find which flags apply to
 your command,
run:

$ sam local generate-event s3 [put/delete] --help

Then you can add in those flags that you wish to customize using

$ sam local generate-event s3 [put/delete] --bucket <bucket> --key <key>

After you generate a sample event, you can use it to test your Lambda function locally
$ sam local generate-event s3 [put/delete] --bucket <bucket> --key <key> | sam local invoke
 <function logical id>
```

Options:

Option	Description
--help	Shows this message and exits.

Commands:

- alexa-skills-kit
- alexa-smart-home
- apigateway
- batch
- cloudformation
- cloudfront
- cloudwatch
- codecommit
- codepipeline
- cognito
- config
- dynamodb
- kinesis
- lex
- rekognition

- s3
- ses
- sns
- sqs
- stepfunctions

sam local invoke

Invokes a local Lambda function once and quits after invocation completes.

This is useful for developing serverless functions that handle asynchronous events (such as Amazon S3 or Amazon Kinesis events). It can also be useful if you want to compose a script of test cases. The event body can be passed in either by `stdin` (default), or by using the `--event` parameter. The runtime output (logs etc) is output to `stderr`, and the Lambda function result is output to `stdout`.

Usage:

```
sam local invoke [OPTIONS] [FUNCTION_IDENTIFIER]
```

Options:

Option	Description
-e, --event PATH	The JSON file that contains event data that's passed to the Lambda function when it's invoked. If you don't specify this option, the default is reading the JSON from `stdin`.
--no-event	Invokes the function with an empty event.
-t, --template PATH	The AWS SAM template file [default: template.[yaml|yml]].
-n, --env-vars PATH	The JSON file that contains values for the Lambda function's environment variables.
--parameter-overrides	Optional. A string that contains AWS CloudFormation parameter overrides encoded as key-value pairs. Use the same format as the AWS CLI—for example, 'ParameterKey=KeyPairName, ParameterValue=MyKey ParameterKey=InstanceType,ParameterValue=t1.micro'.
-d, --debug-port TEXT	When specified, starts the Lambda function container in debug mode and exposes this port on the local host.
--debugger-path TEXT	The host path to a debugger that will be mounted into the Lambda container.
--debug-args TEXT	Additional arguments to be passed to the debugger.
-v, --docker-volume-basedir TEXT	The location of the base directory where the AWS SAM file exists. If Docker is running on a remote machine, you must mount the path where the AWS SAM file exists on the Docker machine, and modify this value to match the remote machine.
--docker-network TEXT	The name or ID of an existing Docker network that Lambda Docker containers should connect to, along with the default bridge network. If this isn't specified, the Lambda containers only connect to the default bridge Docker network.

Option	Description
-l, --log-file TEXT	The log file to send runtime logs to.
--layer-cache-basedir DIRECTORY	Specifies the location basedir where the layers your template uses are downloaded to.
--skip-pull-image	Specifies whether the CLI should skip pulling down the latest Docker image for the Lambda runtime.
--force-image-build	Specifies whether the CLI should rebuild the image used for invoking functions with layers.
--profile TEXT	The AWS credentials profile to use.
--region TEXT	Sets the AWS Region of the service (for example, us-east-1).
--debug	Turns on debug logging.
--help	Shows this message and exits.

sam local start-api

Allows you to run your serverless application locally for quick development and testing. When you run this command in a directory that contains your serverless functions and your AWS SAM template, it creates a local HTTP server that hosts all of your functions.

When it's accessed (through a browser, CLI, and so on), it starts a Docker container locally to invoke the function. It reads the CodeUri property of the AWS::Serverless::Function resource to find the path in your file system that contains the Lambda function code. This could be the project's root directory for interpreted languages like Node.js and Python, or a build directory that stores your compiled artifacts or a Java Archive (JAR) file.

If you're using an interpreted language, local changes are available immediately in the Docker container on every invoke. For more compiled languages or projects that require complex packing support, we recommend that you run your own building solution, and point AWS SAM to the directory or file that contains the build artifacts.

Usage:

```
sam local start-api [OPTIONS]
```

Options:

Option	Description
--host TEXT	The local hostname or IP address to bind to (default: '127.0.0.1').
-p, --port INTEGER	The local port number to listen on (default: '3000').
-s, --static-dir TEXT	Any static asset (for example, CSS/JavaScript/HTML) files located in this directory are presented at /.
-t, --template PATH	The AWS SAM template file [default: template.[yaml|yml]].
-n, --env-vars PATH	The JSON file that contains values for the Lambda function's environment variables.

Option	Description
--parameter-overrides	Optional. A string that contains AWS CloudFormation parameter overrides encoded as key-value pairs. Use the same format as the AWS CLI—for example, 'ParameterKey=KeyPairName, ParameterValue=MyKey ParameterKey=InstanceType,ParameterValue=t1.micro'.
-d, --debug-port TEXT	When specified, starts the Lambda function container in debug mode and exposes this port on the local host.
--debugger-path TEXT	The host path to a debugger that will be mounted into the Lambda container.
--debug-args TEXT	Additional arguments to be passed to the debugger.
-v, --docker-volume-basedir TEXT	The location of the base directory where the AWS SAM file exists. If Docker is running on a remote machine, you must mount the path where the AWS SAM file exists on the Docker machine, and modify this value to match the remote machine.
--docker-network TEXT	The name or ID of an existing Docker network that the Lambda Docker containers should connect to, along with the default bridge network. If this isn't specified, the Lambda containers only connect to the default bridge Docker network.
-l, --log-file TEXT	The log file to send runtime logs to.
--layer-cache-basedir DIRECTORY	Specifies the location basedir where the Layers your template uses are downloaded to.
--skip-pull-image	Specifies whether the CLI should skip pulling down the latest Docker image for the Lambda runtime.
--force-image-build	Specifies whether CLI should rebuild the image used for invoking functions with layers.
--profile TEXT	The AWS credentials profile to use.
--region TEXT	Sets the AWS Region of the service (for example, us-east-1).
--debug	Turns on debug logging.
--help	Shows this message and exits.

sam local start-lambda

Enables you to programmatically invoke your Lambda function locally by using the AWS CLI or SDKs. This command starts a local endpoint that emulates AWS Lambda. You can run your automated tests against this local Lambda endpoint. When you send an invoke to this endpoint using the AWS CLI or SDK, it locally executes the Lambda function that's specified in the request.

Usage:

```
sam local start-lambda [OPTIONS]
```

Examples:

```
SETUP
```

```
------
Start the local Lambda endpoint by running this command in the directory that contains your
 AWS SAM template.
$ sam local start-lambda

USING AWS CLI
-------------
Then, you can invoke your Lambda function locally using the AWS CLI
$ aws lambda invoke --function-name "HelloWorldFunction" --endpoint-url
 "http://127.0.0.1:3001" --no-verify-ssl out.txt

USING AWS SDK
-------------
You can also use the AWS SDK in your automated tests to invoke your functions
 programatically.
Here is a Python example:
    self.lambda_client = boto3.client('lambda',
                                      endpoint_url="http://127.0.0.1:3001",
                                      use_ssl=False,
                                      verify=False,
                                      config=Config(signature_version=UNSIGNED,
                                                    read_timeout=0,
                                                    retries={'max_attempts': 0}))
    self.lambda_client.invoke(FunctionName="HelloWorldFunction")
```

Options:

Option	Description
--host TEXT	The local hostname or IP address to bind to (default: '127.0.0.1').
-p, --port INTEGER	The local port number to listen on (default: '3001').
-t, --template PATH	The AWS SAM template file [default: template.[yaml|yml]].
-n, --env-vars PATH	The JSON file that contains values for the Lambda function's environment variables.
--parameter-overrides	Optional. A string that contains AWS CloudFormation parameter overrides encoded as key-value pairs. Use the same format as the AWS CLI—for example, 'ParameterKey=KeyPairName, ParameterValue=MyKey ParameterKey=InstanceType,ParameterValue=t1.micro'.
-d, --debug-port TEXT	When specified, starts the Lambda function container in debug mode, and exposes this port on the local host.
--debugger-path TEXT	The host path to a debugger to be mounted into the Lambda container.
--debug-args TEXT	Additional arguments to be passed to the debugger.
-v, --docker-volume-basedir TEXT	The location of the base directory where the AWS SAM file exists. If Docker is running on a remote machine, you must mount the path where the AWS SAM file exists on the Docker machine, and modify this value to match the remote machine.
--docker-network TEXT	The name or ID of an existing Docker network that Lambda Docker containers should connect to, along with the default bridge network. If this is specified, the Lambda containers only connect to the default bridge Docker network.
-l, --log-file TEXT	The log file to send runtime logs to.

Option	Description
--layer-cache-basedir DIRECTORY	Specifies the location basedir where the layers your template uses are downloaded to.
--skip-pull-image	Specifies whether the CLI should skip pulling down the latest Docker image for the Lambda runtime.
--force-image-build	Specify whether the CLI should rebuild the image used for invoking functions with layers.
--profile TEXT	The AWS credentials profile to use.
--region TEXT	Sets the AWS Region of the service (for example, us-east-1).
--debug	Turns on debug logging.
--help	Shows this message and exits.

sam logs

Fetches logs that are generated by your Lambda function.

When your functions are a part of an AWS CloudFormation stack, you can fetch logs by using the function's logical ID when you specify the stack name.

Usage:

```
sam logs [OPTIONS]
```

Examples:

```
$ sam logs -n HelloWorldFunction --stack-name mystack

Or, you can fetch logs using the function's name.
$ sam logs -n mystack-HelloWorldFunction-1FJ8PD36GML2Q

You can view logs for a specific time range using the -s (--start-time) and -e (--end-time)
 options
$ sam logs -n HelloWorldFunction --stack-name mystack -s '10min ago' -e '2min ago'

You can also add the --tail option to wait for new logs and see them as they arrive.
$ sam logs -n HelloWorldFunction --stack-name mystack --tail

Use the --filter option to quickly find logs that match terms, phrases or values in your
 log events.
$ sam logs -n HelloWorldFunction --stack-name mystack --filter "error"
```

Options:

Option	Description
-n, --name TEXT	The name of your Lambda function. If this function is part of an AWS CloudFormation stack, this can be the logical ID of the function resource in the AWS CloudFormation/AWS SAM template. [required]
--stack-name TEXT	The name of the AWS CloudFormation stack that the function is a part of.

Option	Description
--filter TEXT	Lets you specify an expression to quickly find logs that match terms, phrases, or values in your log events. This can be a simple keyword (for example, "error") or a pattern that's supported by Amazon CloudWatch Logs. For the syntax, see the Amazon CloudWatch Logs documentation.
-s, --start-time TEXT	Fetches logs starting at this time. The time can be relative values like '5mins ago', 'yesterday', or a formatted timestamp like '2018-01-01 10:10:10'. It defaults to '10mins ago'.
-e, --end-time TEXT	Fetches logs up to this time. The time can be relative values like '5mins ago', 'tomorrow', or a formatted timestamp like '2018-01-01 10:10:10'.
-t, --tail	Tails the log output. This ignores the end time argument and continues to fetch logs as they become available.
--debug	Turns on debug logging to print debug messages that are generated by the AWS SAM CLI.
--profile TEXT	Selects a specific profile from your credential file to get AWS credentials.
--region TEXT	Sets the AWS Region of the service (for example, us-east-1).
--debug	Turns on debug logging to print debug messages that are generated by the AWS SAM CLI.
--help	Shows this message and exits.

sam package

Packages an AWS SAM application. This is an alias for `aws cloudformation package`.

> **Note**
> If the AWS SAM template contains a `Metadata` section for ServerlessRepo, and the `LicenseUrl` or `ReadmeUrl` properties contain references to local files, you must update AWS CLI to version 1.16.77 or later. For more information about the `Metadata` section of AWS SAM templates and publishing applications with AWS SAM CLI, see Publishing Serverless Applications Using the AWS SAM CLI (p. 38).

Usage:

```
sam package [OPTIONS] [ARGS]...
```

Options:

Option	Description
--template-file PATH	The path where your AWS SAM template is located.
--s3-bucket BUCKET	The name of the S3 bucket where this command uploads the artifacts that are referenced in your template.
--output-template-file PATH	The path to the file where the command writes the packaged template. If you don't specify a path, the command writes the template to the standard output.

Option	Description
--profile TEXT	Select a specific profile from your credential file to get AWS credentials.
--region TEXT	Sets the AWS Region of the service (for example, us-east-1).
--debug	Turns on debug logging.
--help	Shows this message and exits.

sam publish

Publish an AWS SAM application to the AWS Serverless Application Repository. This command takes a packaged AWS SAM template and publishes the application to the specified region.

This command expects the AWS SAM template to include a `Metadata` containing application metadata required for publishing. Furthermore, these properties must include references to Amazon S3 buckets for `LicenseUrl` and `ReadmeUrl` values, and not references to local files. For more details about the `Metadata` section of the AWS SAM template, see Publishing Serverless Applications Using the AWS SAM CLI (p. 38).

This command creates the application as private by default, so you must share the application before other AWS accounts are allowed to view and deploy the application. For more information on sharing applications see Using Resource-Based Policies for the AWS Serverless Application Repository.

Usage:

```
sam publish [OPTIONS]
```

Examples:

```
To publish an application
$ sam publish --template packaged.yaml --region us-east-1
```

Options:

Option	Description
-t, --template PATH	AWS SAM template file [default: template.[yaml|yml]].
--semantic-version TEXT	Optional. The semantic version of the application provided by this parameter overrides SemanticVersion in the `Metadata` section of the template file. https://semver.org/
--profile TEXT	Select a specific profile from your credential file to get AWS credentials.
--region TEXT	Sets the AWS Region of the service (for example, us-east-1).
--debug	Turns on debug logging.
--help	Shows this message and exits.

sam validate

Validates an AWS SAM template.

Usage:

```
sam validate [OPTIONS]
```

Options:

Option	Description
-t, --template PATH	The AWS SAM template file [default: template.[yaml|yml]].
--profile TEXT	Selects a specific profile from your credential file to get AWS credentials.
--region TEXT	Sets the AWS Region of the service (for example, us-east-1).
--debug	Turns on debug logging.
--help	Shows this message and exits.

Document History for AWS Serverless Application Model

The following table describes the important changes in each release of the *AWS Serverless Application Model Developer Guide*. For notification about updates to this documentation, you can subscribe to an RSS feed by choosing the RSS button in the top menu panel.

- **Latest documentation update:** March 21, 2019

update-history-change	update-history-description	update-history-date
Controlling access to API Gateway APIs (p. 68)	Added support for controlling access to API Gateway APIs. For more information, see Controlling Access to API Gateway APIs.	March 21, 2019
Added sam publish to AWS SAM CLI (p. 68)	The new `sam publish` command in the AWS SAM CLI simplifies the process for publishing serverless applications in the AWS Serverless Application Repository. For more information, see Publishing an Applications Using AWS SAM CLI.	December 21, 2018
Nested applications and layers support (p. 68)	Added support for nested applications and layers. For more information, see Nested Applications and Working with Layers.	November 29, 2018
Added sam build to AWS SAM CLI (p. 68)	The new `sam build` command in the AWS SAM CLI simplifies the process for compiling serverless applications with dependencies, so that you can locally test and deploy these applications. For more information, see Building Applications with Dependencies.	November 19, 2018
Added new installation options for AWS SAM CLI (p. 68)	Added Linuxbrew (Linux), MSI (Windows), and Homebrew (macOS) installation options for the AWS SAM CLI. For more information, see Installing the AWS SAM CLI.	November 7, 2018

New guide (p. 68)	This is the first release of the *AWS Serverless Application Model Developer Guide*.	October 17, 2018