
CloudFormation Command Line Interface User Guide for Extension Development



CloudFormation Command Line Interface: User Guide for Extension Development

Copyright © 2020 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is the CloudFormation Command Line Interface?	1
Setting up your dev environment	1
Setting up your environment (macOS)	1
Upgrading to CFN-CLI 2.0	2
Creating resource types	4
Using the CloudFormation CLI to create resource type	4
Modeling resource types	5
Defining Property Attributes	6
How to Define a Minimal Resource	6
Resource type schema	6
Patterns for modeling resources	13
Developing resource types	15
Implementing resource handlers	15
Monitoring runtime logging for resource types	16
Testing resource types	16
Progress chaining, stabilization and callback pattern	30
Registering resource type	34
Resource type versions	35
Resource type scope	35
Resource type provisioning	35
Deregistering resource types and type versions	35
Walkthrough: Develop a resource type	36
Prerequisites	36
Create the resource type development project	36
Model the resource type	36
Implement the Resource Handlers	39
Test the resource type	55
Submit the resource type	60
Provision the resource in a CloudFormation stack	61
Resource type FAQ	62
Updates	62
Schema development	62
Permissions and authorization	62
Resource type development	63
Testing	63
Deployment	64
Developing modules	65
Module structure	65
Creating the template fragment	66
Generating the module schema	69
Develop a module	69
CloudFormation CLI reference	71
Global parameters	71
init	71
Description	71
Synopsis	71
Options	72
Output	72
generate	72
Description	72
Synopsis	72
Output	72
validate	72
Description	72

Synopsis	72
Output	72
invoke	72
Description	72
Synopsis	73
Options	73
Output	73
test	73
Description	73
Synopsis	74
Options	74
Output	74
submit	74
Description	74
Synopsis	75
Options	75
Output	76
Document History	77
AWS glossary	78

What is the CloudFormation Command Line Interface?

The CloudFormation Command Line Interface (CLI) is an open-source tool that enables you to develop and test AWS and third-party extensions, such as resource types or modules, and register them for use in AWS CloudFormation. The CloudFormation CLI provides a consistent way to model and provision both AWS and third-party extensions through CloudFormation. It includes commands to enable each step of creating your extensions.

An extension is an artifact, registered in the CloudFormation Registry, which augments the functionality of CloudFormation in a native manner. Extensions can be written by Amazon, APN partners, Marketplace sellers, and the developer community.

You can use the CloudFormation CLI to register extensions—both those you create yourself, as well as ones shared with you—with the CloudFormation registry. This enables you to use CloudFormation capabilities to create, provision, and manage these custom types in a safe and repeatable manner, just as you would any AWS resource. For more information on the CloudFormation registry, see [Using the CloudFormation registry](#) in the *CloudFormation User Guide*.

Setting up your environment for developing extensions

Before you can develop extensions, you'll need to set up your developer environment, including the CloudFormation CLI.

Currently, plugins are available for the following languages:

- Go
- Java
- Python

Or, if you're using another language, you can install the CloudFormation CLI directly.

Setting up your environment (macOS)

Prerequisites

- Python version 3.6 or above
- [AWS Command Line Interface](#) for access to `aws cloudformation` commands.
- Your choice of IDE

The [Walkthrough: Develop a resource type \(p. 36\)](#) walkthrough uses the Community Edition of the IntelliJ IDEA.

- [AWS Serverless Application Model Command Line Interface \(AWS SAM CLI\)](#)

Note

Installing the [AWS SAM CLI](#) requires Docker as a prerequisite for testing your resource type locally.

Complete the following steps:

1. Install Homebrew

First, install [Homebrew](#), an open-source package manager for macOS. You'll use Homebrew to install additional development requirements.

2. Next, use Homebrew to install Python and the AWS Command Line Interface (AWS CLI).

```
$ brew update
$ brew install python awscli
```

Installing the CloudFormation CLI and plugins (macOS)

Use the Python Package Index (PyPI) to install the development plugin for the language of your choice. Installing any of the plugins listed below also installs the CloudFormation CLI. For full installation instructions, refer to the appropriate plugin repository.

Available Language Plugins

Language	Plugin Status	GitHub Location	PyPI Installation
Go	General Availability	cloudformation-cli-go-plugin	cloudformation-cli-go-plugin
Java	General Availability	cloudformation-cli-java-plugin	cloudformation-cli-java-plugin
Python	General Availability	cloudformation-cli-python-plugin	cloudformation-cli-python-plugin

Upgrading to CFN-CLI 2.0

If you have developed resource types using the CFN-CLI 1.0, we recommend you update to CFN-CLI 2.0 and rebuild those types. Upgrading involves updating the CFN-CLI, as well as any language plugins you are using, but does not require any changes to your resource type solution itself.

Enhancements in CFN-CLI 2.0 include:

- Increased resource payload limit, from 8 kb to 6 mb.
- Increased resource stabilization time, from 12 hours to 36 hours, or 48 hours if you are using a stack role to consume the resource.
- Improved resource stability, with improved retry strategy and fail-fast.

To upgrade CFN-CLI 2.0 and the CloudFormation Provider Development Toolkit Go Plugin

1. Upgrade the Go Plugin using the following command:

```
pip3 install --upgrade cloudformation-cli-go-plugin
```

2. Update the Go plugin in the `go.mod` file.

```
go get -u github.com/aws-cloudformation/cloudformation-cli-go-plugin
```

3. To update a resource type to use the CFN-CLI 2.0, build and register a new version of the resource using the following command:

```
make  
cfn submit --set-default
```

To upgrade CFN-CLI 2.0 and the CloudFormation Provider Development Toolkit Java Plugin

1. Upgrade the Java Plugin using the following command:

```
pip3 install --upgrade cloudformation-cli-java-plugin
```

2. Update Java plugin in maven `pom.xml` to 2.0.0

```
<dependency>  
  <groupId>software.amazon.cloudformation</groupId>  
  <artifactId>aws-cloudformation-rpdk-java-plugin</artifactId>  
  <version>2.0.0</version>  
</dependency>
```

3. To update a resource type to use the CFN-CLI 2.0, build and register a new version of the resource using the following command:

```
mvn package  
cfn submit --set-default
```

To upgrade CFN-CLI 2.0 and the CloudFormation Provider Development Toolkit Python Plugin

1. Upgrade the Python Plugin using the following command:

```
pip3 install cloudformation-cli-python-plugin
```

2. To update a resource type to use the CFN-CLI 2.0, build and register a new version of the resource using the following command:

```
cfn submit --set-default
```

Creating resource types

If you use third-party resources in your infrastructure and applications, you can now model and automate those resources by developing them as *resource types* for use within CloudFormation. A resource type includes a resource type specification, and handlers that control API interactions with the underlying AWS or third-party services. These interactions include create, read, update, delete and list (CRUDL) operations for resources. Use resource types to model and provision resources using CloudFormation.

Resource types are treated as first-class citizens within CloudFormation; you can use CloudFormation capabilities to create, provision, and manage these custom resources in a safe and repeatable manner, just as you would any AWS resource. Using resource types for third-party resources provides you a way to reliably manage these resources using a single tool, without having to resort to time-consuming and error-prone methods like manual configuration or custom scripts.

You can create resource types and make them available for use within the AWS account in which they are registered.

Using the CloudFormation CLI to create resource types

Use the [AWS CloudFormation Command Line Interface \(CLI\)](#) to develop your resource types. The CloudFormation CLI is an open-source project that provides a consistent way to model and provision both AWS and third-party resources using CloudFormation. It includes commands to enable each step of creating your resource types.

There are three major steps in developing a resource types:

- Model

Create and validate a schema that serves as the canonical definition of your resource type.

Use the `init` (p. 71) command to generate your resource project, including an example resource schema. Edit the example schema to define the actual model of your resource type. This includes resource properties and their attributes, as well specifying resource event handlers and any permissions needed for each.

As you iterate on your resource model, you can use the `validate` (p. 72) command to validate your schema against the [Resource type definition schema](#) and fix any issues.

- Develop

Add logic that controls what happens to the resource at each stage in its lifecycle, and then test the resource locally to ensure it works as expected.

Implement the resource provisioning actions that the CloudFormation CLI stubbed out when you initially generated your resource project.

If you make changes to your resource schema, use the `generate` (p. 72) command to generate the language-specific data model, contract test, and unit test stubs based on the current state of

the resource schema. (If you use the Java add-in for the CloudFormation CLI, this is done for you automatically.)

When you're ready to test the resource behavior, the CloudFormation CLI provides two commands for testing:

- Use the `invoke` command to test a single handler.
- Use the `test` (p. 73) command to run the entire suite of resource contract tests locally, using the AWS SAM Command Line Interface (SAM CLI), to make sure the handlers you've written comply with expected handler behavior at each stage of the resource lifecycle.
- Register

Register the resource type with the CloudFormation registry in order to make it available for use in CloudFormation templates.

Use the `submit` (p. 74) command to register the resource type with CloudFormation and make it available for use in CloudFormation operations. Registration includes:

- Validating the resource schema.
- Packaging up the resource project files and uploading them to CloudFormation.
- Registering the resource definition in your account, in the specified region.

You can register multiple versions of a resource type, and specify which version you want users to use by default.

Modeling resource types for use in AWS CloudFormation

The first step in creating a resource type is *modeling* that resource, which involves crafting a schema that defines the resource, its properties, and their attributes. When you initially create your resource type project using the CloudFormation CLI `init` (p. 71) command, one of the files created is an example resource schema. Use this schema file as a starting point for defining the shape and semantics of your resource type.

In order to be considered valid, your resource type's schema must adhere to the [Resource type definition schema](#). This meta-schema provides a means of validating your resource specification during resource development.

The Resource Type Definition Schema is a *meta-schema* that extends [draft-07](#) of the [JSON Schema](#). To simplify authoring resource specifications, the Resource Type Definition Schema constrains the scope of the full JSON Schema standard in terms of how certain validations can be expressed, and encourages consistent modelling for all resource schemas. (For full details on how the Resource Type Definition Schema differs from the full JSON schema, see [Divergence From JSON Schema](#).)

Once you have defined your resource schema, you can use the CloudFormation CLI `validate` (p. 72) command to verify that the resource schema is valid.

In terms of testing, the resource schema also determines:

- What unit test stubs are generated in your resource package, and what contract tests are appropriate to run for the resource. When you run the CloudFormation CLI `generate` (p. 72) command, the CloudFormation CLI generates empty unit tests based on the properties of the resource and their attributes.
- Which contract tests are appropriate for CloudFormation CLI to run for your resources. When you run the `test` (p. 73) command, the CloudFormation CLI runs the appropriate contract tests, based on which handlers are included in your resource schema.

Defining Property Attributes

Certain properties of a resource may have special meaning when used in different contexts. For example, a given resource property may be read-only when read back for state changes, but can be specified when used as the target of a `$ref` from a related resource. Because of this semantic difference in how this property metadata should be interpreted, certain property attributes are defined at the resource level, rather than at a property level.

These attributes include:

- `primaryIdentifier`
- `additionalIdentifiers`
- `createOnlyProperties`
- `readOnlyProperties`
- `writeOnlyProperties`

For reference information on resource schema elements, see [Resource type schema \(p. 6\)](#).

How to Define a Minimal Resource

The example below displays a minimal resource type definition. In this case, the resource consists of a single optional property, `Name`, which is also specified as its primary (and only) identifier.

Note that this resource schema would require a `handlers` section with the create, read, and update handlers specified in order for the resource to actually be provisioned within a CloudFormation account.

```
{
  "typeName": "myORG::myService::myResource",
  "properties": {
    "Name": {
      "description": "The name of the resource.",
      "type": "string",
      "pattern": "^[a-zA-Z0-9_-]{0,64}$",
      "maxLength": 64
    }
  },
  "createOnlyProperties": [
    "/properties/Name"
  ],
  "identifiers": [
    [
      "/properties/Name"
    ]
  ],
  "additionalProperties": false
}
```

Resource type schema

In order to be considered valid, your resource type's schema must adhere to the [Resource Provider Definition Schema](#). This meta-schema provides a means of validating your resource specification during resource development.

Syntax

Below is the structure for a typical resource type schema. For the complete meta-schema definition, see the [Resource Provider Definition Schema](#) on [GitHub](#).

```
{
  "typeName": "string",
  "description": "string",
  "sourceUrl": "string",
  "documentationUrl": "string",
  "definitions": {
    "definitionName": {
      . . .
    }
  },
  "properties": {
    "propertyName": {
      "description": "string",
      "type": "string",
      . . .
    },
  },
  "required": [
    "propertyName"
  ],
  "readOnlyProperties": [
    "/properties/propertyName"
  ],
  "writeOnlyProperties": [
    "/properties/propertyName"
  ],
  "primaryIdentifier": [
    "/properties/propertyName"
  ],
  "createOnlyProperties": [
    "/properties/propertyName"
  ],
  "deprecatedProperties": [
    "/properties/propertyName"
  ],
  "additionalIdentifiers": [
    [
      "/properties/propertyName"
    ]
  ],
  "handlers": {
    "create": {
      "permissions": [
        "permission"
      ]
    },
    "read": {
      "permissions": [
        "permission"
      ]
    },
    "update": {
      "permissions": [
        "permission"
      ]
    },
    "delete": {
      "permissions": [
        "permission"
      ]
    },
    "list": {
      "permissions": [
```

```
    "permission"  
  ]  
}  
}
```

Properties

typeName

The unique name for your resource. Specifies a three-part namespace for your resource, with a recommended pattern of `Organization::Service::Resource`.

Note

The following organization namespaces are reserved and cannot be used in your resource type names:

- Alexa
- AMZN
- Amazon
- ASK
- AWS
- Custom
- Dev

Required: Yes

Pattern: `^[a-zA-Z0-9]{2,64}:[a-zA-Z0-9]{2,64}:[a-zA-Z0-9]{2,64}$`

description

A short description of the resource. This will be displayed in the AWS CloudFormation console.

Required: Yes

sourceUrl

The URL of the source code for this resource, if public.

documentationUrl

The URL of a page providing detailed documentation for this resource.

Note

While the resource schema itself should include complete and accurate property descriptions, the `documentationUrl` property enables you to provide users with documentation that describes and explains the resource in more detail, including examples, use cases, and other detailed information.

definitions

Use the `definitions` block to provide shared resource property schemas.

It is considered a best practice is to use the `definitions` section to define schema elements that may be used at multiple points in your resource type schema. You can then use a JSON pointer to reference that element at the appropriate places in your resource type schema.

properties

The properties of the resource.

All properties of a resource must be expressed in the schema. Arbitrary inputs are not allowed. A resource must contain at least one property.

Nested properties are not allowed. Instead, define any nested properties in the `definitions` element, and use a `$ref` pointer to reference them in the desired property.

Required: Yes

propertyName

insertionOrder

For properties of type `array`, set to `true` to specify that the order in which array items are specified must be honored, and that changing the order of the array will indicate a change in the property.

The default is `false`.

readOnly

Reserved for CloudFormation use.

writeOnly

Reserved for CloudFormation use.

dependencies

Any properties that are required if this property is specified.

patternProperties

Use to specify a specification for key-value pairs.

```
"type": "object",  
"propertyNames": {  
  "format": "regex"  
}
```

properties

Minimum: 1

patternProperties

Pattern: `^[A-Za-z0-9]{1,64}$`

Specifies a pattern that properties must match to be valid.

allOf

The property must contain all of the data structures define here.

Contains a single schema. A list of schemas is not allowed.

Minimum: 1

anyOf

The property can contain any number of the data structures define here.

Contains a single schema. A list of schemas is not allowed.

Minimum: 1

oneOf

The property must contain only one of the data structures define here.

Contains a single schema. A list of schemas is not allowed.

Minimum: 1

items

For properties of type array, defines the data structure of each array item.

Contains a single schema. A list of schemas is not allowed.

In addition, the following elements, defined in [draft-07](#) of the [JSON Schema](#), are allowed in the `properties` object:

- `$ref`
- `$comment`
- `title`
- `description`
- `examples`
- `default`
- `multipleOf`
- `maximum`
- `exclusiveMaximum`
- `minimum`
- `exclusiveMinimum`
- `exclusiveMinimum`
- `minLength`
- `pattern`
- `maxItems`
- `minItems`
- `uniqueItems`
- `contains`
- `maxProperties`
- `maxProperties`
- `required`
- `const`
- `enum`
- `type`
- `format`

remote

Reserved for CloudFormation use.

readOnlyProperties

Resource properties that can be returned by a `read` or `list` request, but cannot be set by the user.

Type: List of JSON pointers

writeOnlyProperties

Resource properties that can be specified by the user, but cannot be returned by a `read` or `list` request. Write-only properties are often used to contain passwords, secrets, or other sensitive data.

Type: List of JSON pointers

createOnlyProperties

Resource properties that can be specified by the user only during resource creation.

Note

Any property not explicitly listed in the `createOnlyProperties` element can be specified by the user during a resource update operation.

Type: List of JSON pointers

deprecatedProperties

Resource properties that have been deprecated by the underlying service provider. These properties are still accepted in create and update operations. However they may be ignored, or converted to a consistent model on application. Deprecated properties are not guaranteed to be returned by read operations.

Type: List of JSON pointers

primaryIdentifier

The uniquely identifier for an instance of this resource type. An identifier is a non-zero-length list of JSON pointers to properties that form a single key. An identifier can be a single or multiple properties to support composite-key identifiers.

Type: List of JSON pointers

Required: Yes

additionalIdentifiers

An optional list of supplementary identifiers, each of which uniquely identifies an instance of this resource type. An identifier is a non-zero-length list of JSON pointers to properties that form a single key. An identifier can be a single property, or multiple properties to construct composite-key identifiers.

Type: List of JSON pointers

Minimum: 1

handlers

Specifies the provisioning operations which can be performed on this resource type. The handlers specified determine what provisioning actions CloudFormation takes with respect to the resource during various stack operations.

- If the resource type does not contain `create`, `read`, and `delete` handlers, CloudFormation cannot actually provision the resource.
- If the resource type does not contain an `update` handler, CloudFormation cannot update the resource during stack update operations, and will instead replace it.

If your resource type calls AWS APIs in any of its handlers, you must create an *IAM execution role* that includes the necessary permissions to call those AWS APIs, and provision that execution role in your account. For more information, see [Accessing AWS APIs from a Resource Type](#).

create

permissions

A string array that specifies the AWS permissions needed to invoke the create handler.

You must specify at least one permission for each handler.

Required: Yes

read

permissions

A string array that specifies the AWS permissions needed to invoke the read handler.

You must specify at least one permission for each handler.

Required: Yes

update

permissions

A string array that specifies the AWS permissions needed to invoke the update handler.

You must specify at least one permission for each handler.

Required: Yes

delete

permissions

A string array that specifies the AWS permissions needed to invoke the delete handler.

You must specify at least one permission for each handler.

Required: Yes

list

The `list` handler must at least return the resource's primary identifier.

permissions

A string array that specifies the AWS permissions needed to invoke the list handler.

You must specify at least one permission for each handler.

Required: Yes

allOf

The resource must contain all of the data structures defined here.

Minimum: 1

anyOf

The resource can contain any number of the data structures define here.

Minimum: 1

oneOf

The resource must contain only one of the data structures define here.

Minimum: 1

In addition, the following element, defined in [draft-07](#) of the [JSON Schema](#), is allowed:

- required

Patterns for modeling your resource types

Use the following patterns to model the data structures of your resource types using the Resource Provider Schema.

How to Specify a Property as Dependent on Another

Use the `dependencies` element to specify if a property is required in order for another property to be specified. In the following example, if the user specifies a value for the `ResponseCode` property, they must also specify a value for `ResponsePagePath`, and vice versa. (Note that, as a best practice, this is also called out in the description of each property.)

```
"properties": {
  "CustomErrorResponse": {
    "additionalProperties": false,
    "dependencies": {
      "ResponseCode": [
        "ResponsePagePath"
      ],
      "ResponsePagePath": [
        "ResponseCode"
      ]
    },
    "properties": {
      "ResponseCode": {
        "description": "The HTTP status code that you want CloudFront to return to the
viewer along with the custom error page. If you specify a value for ResponseCode, you must
also specify a value for ResponsePagePath.",
        "type": "integer"
      },
      "ResponsePagePath": {
        "description": "The path to the custom error page that you want CloudFront
to return to a viewer when your origin returns the HTTP status code specified by
ErrorCode. If you specify a value for ResponsePagePath, you must also specify a value for
ResponseCode.",
        "type": "string"
      }
      . . .
    },
    "type": "object"
  },
},
. . .
```

How to Define Nested Properties

It is considered a best practice is to use the `definitions` section to define schema elements that may be used at multiple points in your resource type schema. You can then use a JSON pointer to reference that element at the appropriate places in your resource type schema.

For example, define the reused element in the `definitions` section:

```
"definitions": {
  "AccountId": {
    "pattern": "^[0-9]{12}$",
    "type": "string"
  },
},
```

```
},  
...  
}
```

And then reference that definition where appropriate:

```
"AwsAccountNumber": {  
  "description": "An AWS account that is included in the TrustedSigners complex type for  
  this distribution.",  
  "$ref": "#/definitions/AccountId"  
},  
...  
}
```

Advanced: How to Encapsulate Complex Logic

Use the `allOf`, `oneOf`, or `anyOf` elements to encapsulate complex logic in your resource type schema.

In the example below, if `whitelist` is specified for the `Forward` property in your resource, then the `WhitelistedNames` property must also be specified.

```
"properties": {  
  "Cookies": {  
    "oneOf": [  
      {  
        "additionalProperties": false,  
        "properties": {  
          "Forward": {  
            "description": "Specifies which cookies to forward to the origin for  
this cache behavior.",  
            "enum": [  
              "all",  
              "none"  
            ],  
            "type": "string"  
          }  
        },  
        "required": [  
          "Forward"  
        ]  
      },  
      {  
        "additionalProperties": false,  
        "properties": {  
          "Forward": {  
            "description": "Specifies which cookies to forward to the origin for  
this cache behavior.",  
            "enum": [  
              "whitelist"  
            ],  
            "type": "string"  
          },  
          "WhitelistedNames": {  
            "description": "Required if you specify whitelist for the value of  
Forward.",  
            "items": {  
              "type": "string"  
            },  
            "minItems": 1,  
            "type": "array"  
          }  
        },  
        "required": [  
          "WhitelistedNames"  
        ]  
      }  
    ]  
  }  
}
```

```
        "Forward",  
        "WhitelistedNames"  
    ]  
    }  
    ,  
    "type": "object"  
},  
},  
. . .
```

Developing resource types for use in AWS CloudFormation Templates

Once you've modeled your resource type, and validated its schema, the next step is to develop the resource. Developing the resource consists of two main steps:

- Implementing the appropriate event handlers for your resource.
- Testing the resource locally to ensure it works as expected.

Implementing resource handlers

When you [generate](#) (p. 72) your resource package, the CloudFormation CLI stubs out empty handler functions, each of which each corresponds to a specific event in the resource lifecycle. You add logic to these handlers to control what happens to your resource type at each stage of its lifecycle.

- `create`: CloudFormation invokes this handler when the resource is initially created during stack create operations.
- `read`: CloudFormation invokes this handler as part of a stack update operation when detailed information about the resource's current state is required.
- `update`: CloudFormation invokes this handler when the resource is updated as part of a stack update operation.
- `delete`: CloudFormation invokes this handler when the resource is deleted, either when the resource is deleted from the stack as part of a stack update operation, or the stack itself is deleted.
- `list`: CloudFormation invokes this handler when summary information about multiple resources of this resource type is required.

You can only specify a single handler for each event.

Which handlers you implement for a resource determine what provisioning actions CloudFormation takes with respect to the resource during various stack operations:

- If the resource type contains both `create` and `update` handlers, CloudFormation invokes the appropriate handler during stack create and update operation.
- If the resource type does not contain an `update` handler, CloudFormation cannot update the resource during stack update operations, and will instead replace it. CloudFormation invokes the `create` handler to create a new resource, then deletes the old resource by invoking the `delete` handler.
- If the resource type does not contain `create`, `read`, and `delete` handlers, CloudFormation cannot actually provision the resource.

Use the resource schema to specify which handlers you have implemented. If you choose not to implement a specific handler, remove it from the `handlers` section of the resource schema.

Accessing AWS APIs from a resource type

If your resource type calls AWS APIs in any of its handlers, you must create an *IAM execution role* that includes the necessary permissions to call those AWS APIs, and provision that execution role in your account. CloudFormation then assumes that execution role to provide your resource type with the appropriate credentials.

When you call `generate`, the CloudFormation CLI automatically generates an execution role template, `resource-role.yaml`, as part of generating the code files for the resource type package. This template is based on the permissions specified for each handler in the `handlers` section of the *Resource type schema*. When you use `submit` to register the resource type, the CloudFormation CLI attempts to create or update an execution role based on the template, and then passes this execution role to CloudFormation as part of the registration.

For more information on the permissions available per AWS service, see *Actions, Resources, and Condition Keys for AWS Services* in the *AWS Identity and Access Management User Guide*.

Monitoring runtime logging for resource types

When you register a resource type using `cf submit`, CloudFormation creates a CloudWatch log group for the resource type in your account. This enables you to access the logs for your resource to help you diagnose any faults. The log group is named according to the following pattern:

```
/my-resource-type-stack-ResourceHandler-string
```

Where:

- *my-resource-type* is the three-part resource type name
- *string* is a unique string generated by CloudFormation

Now, when you execute stack operations for stacks that contain the resource type, CloudFormation delivers log events emitted by the resource type to this log group.

Testing resource types using contract tests

As you model and develop your resource type, you can have the CloudFormation CLI perform tests to ensure the resource type is behaving as expected during each event in the resource lifecycle. The CloudFormation CLI performs a suite of tests, each written to test a requirement contained in the resource type handler contract.

Testing resource types locally using SAM

Once you've implemented the desired handlers for your resource, you can also test the resource locally using the AWS SAM command line interface (CLI), to make sure your resource behaves as expected, debug what's wrong, and fix any issues.

To start testing, use the SAM CLI to start the Local Lambda Service. Run the following command in a terminal separate from your resource type workspace, or as a background process.

```
$ sam local start-lambda
```

If you have functions defined in your SAM template, it will provide an endpoint to invoke these functions locally. This is especially helpful because it allows for remote debugging to step through resource type invocations in real time.

```
Starting the Local Lambda Service. You can now invoke your Lambda Functions defined in your
template through the endpoint.
2020-01-15 15:27:19 * Running on http://127.0.0.1:3001/ (Press CTRL+C to quit)
```

Alternatively, you can also specify using the public Lambda Service and invoke functions deployed in your account. Be aware, however, that using the local service allows for more iteration. To specify a debug port for remote debugging, use the `-d` option:

```
sam local start-lambda -d PORT
```

Once you have the Lambda service started, use the `test` command to perform contract tests:

```
cfn test
```

The CloudFormation CLI selects the appropriate contract tests to execute, based on the handlers specified in your resource type schema. If a test fail, the CloudFormation CLI outputs a detailed trace of the failure, including the related assertion failure and mismatched values.

For more information on testing using AWS SAM CLI, see [Testing and Debugging Serverless Applications](#) in the *AWS Serverless Application Model Developer Guide*.

How the CloudFormation CLI constructs and executes contract tests

The CloudFormation CLI uses [PyTest](#), an open-source testing framework, to execute the contract tests.

The tests themselves are located in the `suite` folder of the CloudFormation CLI repository on GitHub. Each test is adorned with the appropriate `pytest` markers. For example, tests applicable to the `create` handler are adorned with the `@pytest.mark.create` marker. This enables the CloudFormation CLI to execute only those tests appropriate for a resource type, based on the handlers specified in the resource type's schema. For example, suppose a resource type's schema specified `create`, `read`, and `delete` handlers. In this case, the CloudFormation CLI would not perform any test marked with only the `@pytest.mark.update` or `@pytest.mark.list`, since those handlers were not implemented.

To test `create` and `update` handlers, the CloudFormation CLI uses the resource type's schema and [Hypothesis](#), an open-source Python library used to generate testing strategies. The resource type schema is walked to generate a strategy for valid resource models, and the strategy is used to generate models for tests.

Tests create, update, and delete resources to test various aspects of the resource handler contract during handler operations. The CloudFormation CLI uses PyTest `fixtures` to decrease the amount of time the contract tests take to perform. Using fixtures enable the tests within a test module to share resources, rather than have to create a new resource for each test. Currently, the contract tests employ the following fixtures:

- `created_resource` in the `handler_create` test module
- `updated_resource` in the `handler_update` test module
- `deleted_resource` in `handler_delete` test module

Specifying input data for use in contract tests

By default, the CloudFormation CLI performs resource contract tests using input properties generated from the patterns you define in your resource type schema. However, most resources are complex enough that the input properties for creating or updating those resources requires an understanding of

the resource being provisioned. To address this, you can specify the input the CloudFormation CLI uses when performing its contract tests.

The CloudFormation CLI offers two ways for you to specify the input data for it to use when performing contract tests:

- Overrides file

Using an `overrides` file provides a light-weight way of specifying input data for certain specific properties for the CloudFormation CLI to use during both create and update operations testing.

- Input files

You can also use multiple `input` files to specify contract test input data if:

- You want or need to specify different input data for create and update operations, or invalid data with which to test.
- You want to specify multiple different input data sets.

Specifying input data using an override file

Using an override file enables you to overwrite input values for specific resource properties. Input values specified in the override file are used in contract testing for both create and update operations. You can only specify a single override file, and only specify a single input value for each resource property. For any properties for which you do not specify values, the CloudFormation CLI uses generated input.

Because the input data specified in the `overrides.json` file is used by the CloudFormation CLI during testing of create and update operations, you cannot include input values for create-only properties in the file, as this would lead to contract test failures during update operations. For more information, see [createOnlyProperties](#).

To override the input data used for specific properties during contract testing, add an `overrides.json` file to the root directory of your resource type project. The `overrides.json` file should contain only the resource properties to be used in testing. Use the following syntax:

```
{
  "CREATE": {
    "/property_name": "property_value" # optional_comment
  }
}
```

For example:

```
{
  "CREATE": {
    "/SubnetId": "subnet-0bc6136e" # This should be a real subnet that exists in the
    account you're testing against.
  }
}
```

You can also use output values from other stacks when specifying input data. For example, suppose you had a stack that contained an export value named `SubnetExport`:

```
Resources:
  VPC:
    Type: "AWS::EC2::VPC"
    Properties:
      CidrBlock: "10.0.0.0/16"
  Subnet:
```

```
Type: "AWS::EC2::Subnet"
Properties:
  CidrBlock: "10.0.0.0/24"
  VpcId: !Ref VPC
Outputs:
  SubnetId:
    Value: !Ref Subnet
  Export:
    Name: SubnetExport
```

You could then specify that export value as input data using the export value name using the following syntax:

```
{
  "CREATE": {
    "/SubnetId": "{{SubnetExport}}"
  }
}
```

For more information, see [Outputs](#).

Specifying input data using input files

Use `input` files to specify different kinds of input data for the CloudFormation CLI to use: create input, update input, and invalid input. Each kind of data is specified in a separate file. You can also specify multiple sets of input data for contract tests.

To specify `input` files for the CloudFormation CLI to use in contract testing, add an `inputs` folder to the root directory of your resource type project. Then add your input files.

Specify which kind of input data a file contains by using the following naming conventions, where `n` is an integer:

- `inputs_n_create.json`: Use files with `_create.json` for specifying inputs for creating the resource. This includes input values for create-only properties. For more information, see [createOnlyProperties](#).
- `inputs_n_update.json`: Use files with `_update.json` for specifying inputs for updating the resource.
- `inputs_n_invalid.json`: Use files with `_invalid.json` for specifying invalid inputs to test when creating or updating the resource.

To specify multiple sets of input data for contract tests, increment the integer in the file names to order your input data sets. For example, your first set of input files should be named `inputs_1_create.json`, `inputs_1_update.json`, and `inputs_1_invalid.json`. Your next set would be named `inputs_2_create.json`, `inputs_2_update.json`, and `inputs_2_invalid.json`, and so on.

Each input file is a JSON file containing only the resource properties to be used in testing. Below is an example of an input file data set.

```
{
  "AlarmName": "Name",
  "AlarmDescription": "TestAlarmDimensions Description",
  "Namespace": "CloudWatchNamespace",
  "MetricName": "Fault",
  "Dimensions": [
    {
      "Name": "MethodName",
```

```
        "Value": "Value"
    }
  ],
  "Statistic": "Average",
  "Period": 60,
  "EvaluationPeriods": 5,
  "Threshold": 0.01,
  "ComparisonOperator": "GreaterThanOrEqualToThreshold"
}
```

If you specify an `inputs` folder, the CloudFormation CLI uses only the input data included in that folder. Therefore, you must specify `create`, `update`, and `invalid` data files for the CloudFormation CLI to successfully complete the contract tests.

If you specify both input files and an `overrides` files, the CloudFormation CLI ignores the `overrides` file and just uses the input data specified in the `inputs` folder.

You can also use output values from other stacks when specifying input data. For example, suppose you had a stack that contained an export value named `SubnetExport`:

```
Resources:
  VPC:
    Type: "AWS::EC2::VPC"
    Properties:
      CidrBlock: "10.0.0.0/16"
  Subnet:
    Type: "AWS::EC2::Subnet"
    Properties:
      CidrBlock: "10.0.0.0/24"
      VpcId: !Ref VPC
Outputs:
  SubnetId:
    Value: !Ref Subnet
    Export:
      Name: SubnetExport
```

You could then specify that export value as input data using the export value name using the following syntax:

```
{
  "SubnetId": "{{SubnetExport}}",
  . . .
}
```

For more information, see [Outputs](#).

Resource type handler contracts

The resource type handler contract specifies the expected and required behavior to which a resource must adhere in each given event handler. It defines a set of specific, unambiguous rules with which `create`, `read`, `update`, `delete` and `list` resource handlers must comply. Following the contract will allow customers to interact with all resource types under a uniform set of behaviors and expectations, and prevents creation of unintended or duplicate resources.

A resource implementation **MUST** pass all resource contract tests in order to be registered.

Assuming no other concurrent interaction on the resource, the handlers **MUST** comply with the following contract.

All terminology in the handler contract requirements adheres to the [RFC 2119 specification](#).

Create handlers

Input assumptions

The `create` handler can make the following assumptions about input submitted to it:

- The input to a `create` handler MUST be valid against the resource schema.

Output requirements

The `create` handler must adhere to the following requirements regarding its output:

- A `create` handler MUST always return a `ProgressEvent` object within 60 seconds. For more information, see [ProgressEvent Object Schema](#).

In every `ProgressEvent` object, the `create` handler MUST return a model which conforms to the shape of the resource schema. For more information, see [Returned models must conform to the shape of the schema](#).

Every model MUST include the `primaryIdentifier`. The only exception is if the first progress event is `FAILED`, and the resource has not yet been created. In this case, a subsequent `read` call MUST return `NotFound`.

- A `create` handler MUST NOT return `SUCCESS` until it has applied all properties included in the `create` request. For more information, see [Update, create, and delete handlers must satisfy desired-state stabilization](#).
- A `create` handler MUST return `IN_PROGRESS` if it has not yet reached the desired-state.

A `create` handler SHOULD return a model containing all properties set so far and nothing more during each `IN_PROGRESS` event.

- A `create` handler MUST return `FAILED` progress event if it cannot reach the desired-state within the timeout specified in the resource schema.

The progress event MUST return an error message and the most applicable error code. For more information, see [Handler error codes](#).

- A `create` handler MAY return `SUCCESS` once it reaches the desired-state.

Once the desired state has been reached, a `create` handler MAY perform runtime-state stabilization. For more information, see [Update and create handlers should satisfy runtime-state stabilization](#).

When the `create` handler returns `SUCCESS`, it MUST return a `ProgressEvent` object containing a model that satisfies the following requirements:

- All properties specified in the `create` request MUST be present in the model returned, and they MUST match exactly, with the exception of properties defined as `writeOnlyProperties` in the resource schema.
- The model MUST contain all properties that have values, including any properties that have default values, and any `readOnlyProperties` as defined in the resource schema.
- The model MUST NOT return any properties that are null or do not have values.
- After a `create` operation returns `SUCCESS`, a subsequent `read` request MUST succeed when passed in the `primaryIdentifier` or any `additionalIdentifiers` associated with the provisioned resource instance.
- After a `create` operation returns `SUCCESS`, a subsequent `list` operation MUST return the `primaryIdentifier` associated with the provisioned resource instance.

If the `list` operation is paginated, the entire `list` operation is defined as all `list` requests until the `nextToken` is `null`.

- A `create` handler MUST be idempotent. A `create` handler MUST NOT create multiple resources given the same idempotency token.
- A `create` handler MUST return FAILED with an `AlreadyExists` error code if the resource already existed prior to the create request.

Update handlers

Input assumptions

The `update` handler can make the following assumptions about input submitted to it:

- The input to an `update` handler MUST be valid against the resource schema.
- Any `createOnlyProperties` specified in update handler input MUST NOT be different from their previous state.
- The input to a `update` handler MUST contain either the `primaryIdentifier` or an `additionalIdentifier`.

Output requirements

The `update` handler must adhere to the following requirements:

- An `update` handler MUST always return a `ProgressEvent` object within 60 seconds. For more information, see [ProgressEvent Object Schema](#).

In every `ProgressEvent` object, the `update` handler MUST return a model which conforms to the shape of the resource schema. For more information, see [Returned models must conform to the shape of the schema](#).

Every model MUST include the `primaryIdentifier`.

The `primaryIdentifier` returned in every progress event must match the `primaryIdentifier` passed into the request.

- An `update` handler MUST NOT return `SUCCESS` until it has applied all properties included in the `update` request. For more information, see [Update, create, and delete handlers must satisfy desired-state stabilization](#).
- An `update` handler MUST return `IN_PROGRESS` if it has not yet reached the desired-state.

An `update` handler SHOULD return a model containing all properties set so far and nothing more during each `IN_PROGRESS` event.

- An `update` handler MUST return `FAILED` progress event if it cannot reach the desired-state within the timeout specified in the resource schema.

The progress event MUST return an error message and the most applicable error code. For more information, see [Handler error codes](#).

- An `update` handler MAY return `SUCCESS` once it reaches the desired-state.

Once the desired state has been reached, an `update` handler MAY perform runtime-state stabilization. For more information, see [Update and create handlers should satisfy runtime-state stabilization](#).

When the `update` handler returns `SUCCESS`, it MUST return a `ProgressEvent` object containing a model that satisfies the following requirements:

- All properties specified in the `update` request MUST be present in the model returned, and they MUST match exactly, with the exception of properties defined as `writeOnlyProperties` in the resource schema.

- The model **MUST** contain all properties that have values, including any properties that have default values, and any `readOnlyProperties` as defined in the resource schema.
- The model **MUST NOT** return any properties that are null or do not have values.

All list or collection properties **MUST** be applied in full. The successful outcome **MUST** be replacement of the previous properties, if any.

- An `update` handler **MUST** return `FAILED` with a `NotFound` error code if the resource did not exist prior to the update request.
- An `update` handler **MUST NOT** create a new physical resource.

Delete handlers

Input assumptions

The `delete` handler can make the following assumptions about input submitted to it:

- The input to a `delete` handler **MUST** contain either the `primaryIdentifier` or an `additionalIdentifier`. Any other properties **MAY NOT** be included in the request.

Output requirements

The `delete` handler must adhere to the following requirements:

- A `delete` handler **MUST** always return a `ProgressEvent` object within 60 seconds. For more information, see [ProgressEvent Object Schema](#).
- A `delete` handler **MUST NOT** return `SUCCESS` until the resource has reached the desired state for deletion. For more information, see [Update, create, and delete handlers must satisfy desired-state stabilization](#).
 - A `delete` handler **MUST** return `IN_PROGRESS` if it has not yet reached the desired state.
 - A `delete` handler **MUST** return `FAILED` progress event if it cannot reach the desired-state within the timeout specified in the resource schema.

The progress event **MUST** return an error message and the most applicable error code. For more information, see [Handler error codes](#).

- A `delete` handler **MUST** return `SUCCESS` once it reaches the desired state. (This is because there is no runtime-state stabilization for delete requests.)

When the `delete` handler returns `SUCCESS`, the `ProgressEvent` object **MUST NOT** contain a model.

- A `delete` handler **MUST** return `FAILED` with a `NotFound` error code if the if the resource did not exist prior to the delete request.
- Once a `delete` operation successfully completes, any subsequent `update`, `delete`, or `read` request for the deleted resource instance **MUST** return `FAILED` with a `NotFound` error code.
- Once a `delete` operation successfully completes, any subsequent `list` operation **MUST NOT** return the `primaryIdentifier` associated with the deleted resource instance.

If the `list` operation is paginated, the 'list operation' is defined as all `list` calls until the `nextToken` is `null`.

- Once a `delete` operation successfully completes, a subsequent `create` request with the same `primaryIdentifier` or `additionalIdentifiers` **MUST NOT** return `FAILED` with an `AlreadyExists` error code.
- Once a `delete` operation successfully completes, the resource **SHOULD NOT** be billable to the customer.

Read handlers

Input assumptions

The read handler can make the following assumptions about input submitted to it:

- The input to a read handler MUST contain either the `primaryIdentifier` or an `additionalIdentifier`. Any other properties MAY NOT be included in the request.

Output requirements

The read handler must adhere to the following requirements regarding its output:

- A read handler MUST always return a `ProgressEvent` object within 30 seconds. For more information, see [ProgressEvent Object Schema](#).

A read handler MUST always return a status of `SUCCESS` or `FAILED`; it MUST NOT return a status of `IN_PROGRESS`.

- A read handler MUST return a model representation that conforms to the shape of the resource schema.
 - The model MUST contain all properties that have values, including any properties that have default values and any `readOnlyProperties` as defined in the resource schema.
 - The model MUST NOT return any properties that are null or do not have values.
- A read handler MUST return `FAILED` with a `NotFound` error code if the resource does not exist.

List handlers

- A list handler MUST always return a `ProgressEvent` object within 30 seconds. For more information, see [ProgressEvent Object Schema](#).

A list handler MUST always return a status of `SUCCESS` or `FAILED`; it MUST NOT return a status of `IN_PROGRESS`.

- A list handler MUST return an array of primary identifiers.

When passed in a read request, each `primaryIdentifier` MUST NOT return `FAILED` with `NotFound` error code.

- A list request MUST support pagination by returning a `NextToken`.

The `NextToken` returned MUST be able to be used in a subsequent list request to retrieve the next set of results from the service.

The `NextToken` MUST be null when all results have been returned.

- A list request MUST return an empty array if there are no resources found.
- A list handler MAY accept a set of properties conforming to the shape of the resource schema as filter criteria.

The filter should use `AND(&)` when multiple properties are passed in.

Additional requirements

The following requirements also apply to resource handlers.

Returned models must conform to the shape of the schema

A model returned in a [ProgressEvent](#) object MUST always conform to the shape of the resource schema. This means that each property that is returned MUST adhere to its own individual restrictions: correct data type, regex, length, etc. However, the model returned MAY NOT contain all properties defined as required in the json-schema.

More specifically, contract tests validate models based on json-schema [Validation Keywords](#).

- ALL Validation Keywords for the following MUST be observed:
 - Any Instance Type (Section 6.1)
 - Numeric Instances (Section 6.2)
 - Strings (Section 6.3)
 - Arrays (Section 6.4)
- All Validation Keywords for Objects (Section 6.5) MUST be observed EXCEPT for:
 - required (Section 6.5.3)
 - dependencies (Section 6.5.7)
 - propertyName (Section 6.5.8)
- Contract tests will NOT validate Validation Keywords for:
 - Applying Subschemas Conditionally (Section 6.6)
 - Applying Subschemas With Boolean Logic (Section 6.7)

Update, create, and delete handlers must satisfy desired-state stabilization

Stabilization is the process of waiting for a resource to be in a particular state. Note that reaching the desired-state is mandatory for all handlers before returning SUCCESS.

Create and update handlers

For Create and Update handlers, desired-state stabilization is satisfied when all properties specified in the request are applied as requested. This is verified by calling the Read handler.

In many cases, the desired-state is reached immediately upon completion of a Create/Update API call. However, in some cases, multiple API calls and/or wait periods may be required in order to reach this state.

Eventual consistency in desired-state stabilization

Eventual consistency means that the result of an API command you run might not be immediately visible to all subsequent commands you run. Handling API eventual consistency is required as part of desired-state stabilization. This is because a subsequent Read call might fail with a NotFound error code.

AWS EC2 resources are a great example of this. For more information, see [Eventual Consistency](#) in the *Amazon Elastic Compute Cloud API Reference*.

Examples of desired-state stabilization

For a simple example of desired-state stabilization, consider the implementation of the `create` handler for the `AWS::Logs::MetricFilter` resource: immediately after the handler code completes the call to the `PutMetricFilter` method, the `AWS::Logs::MetricFilter` has achieved its desired state. You can examine the code for this resource in its open-source repository at github.com/aws-cloudformation/aws-cloudformation-resource-providers-logs.

A more complex example is the implementation of the `update` handler for the `AWS::Kinesis::Stream` resource. The update handler must make multiple API calls during

an update, including `AddTagsToStream` or `RemoveTagsFromStream`, `UpdateShardCount`, `IncreaseRetentionPeriod` or `DecreaseRetentionPeriod`, and `StartStreamEncryption` or `StopStreamEncryption`. Meanwhile, each API call will set the `StreamStatus` to `UPDATING`, during which time other API calls cannot be performed or the API will throw a `ResourceInUseException`. Therefore, in order to reach the desired state, the handler will need to wait for the `StreamStatus` to become `ACTIVE` in between each API call.

Delete handlers

In most cases, the definition of 'deleted' is obvious. A `delete` API call will result in the resource being purged from the database, and the resource is no longer describable to the user.

However, in some cases, a deletion will result in the resource leaving an *audit trail*, in which the resource can still be described by service APIs, but can no longer be interacted with by the user. For example, when you delete a CloudFormation stack, it is assigned a status of `DELETE_COMPLETE`, but it can still be returned from a `DescribeStacks` API call. For resources like this, the desired-state for deletion is when the resource has reached a *terminal, inoperable, and irrecoverable state*. If the resource can continue to be mutated by the user through another API call, then it is not *deleted*, it is *updated*.

Note that there is no difference between desired-state stabilization and runtime-state stabilization for a delete handler. By definition, once a resource has reached the desired-state for deletion, a subsequent read call **MUST** return `FAILED` with a `NotFound` error code, and a subsequent `create` call with the same `primaryIdentifier` or `additionalIdentifiers` **MUST NOT** return `FAILED` with an `AlreadyExists` error code. Additional restrictions are defined in the contract above.

So in the case of a CloudFormation stack, a `read` handler **MUST** return `FAILED` with a `NotFound` error code if the stack is `DELETE_COMPLETE`, even though it's audit trail can still be accessed by the `DescribeStacks` API.

Update and create handlers should satisfy runtime-state stabilization

Runtime-state stabilization is a process of waiting for the resource to be "ready" to use. Generally, runtime-state stabilization is done by continually describing the resource until it reaches a particular state, though it can take many forms.

Runtime-state stabilization can mean different things for different resources, but the following are common requirements:

- *Additional mutating API calls can be made on the resource*

Some resources cannot be modified while they are in a particular state

- *Dependent resources can consume the resource*

There may be other resources which need to consume the resource in some way, but can't until it is in a particular state.

- *Users can interact with the resource*

Customers may not be able to use the resource until it is in a particular status. This usually overlaps with the dependent resources requirement, although there could be different qualifications, depending on the resources.

Note that while desired-state stabilization is mandatory, runtime-state stabilization is optional but encouraged. Users have come to expect that once a resource is `COMPLETE`, they will be able to use it.

Examples of run-time stabilization

For a simple example of run-time stabilization, consider the implementation of the `create` handler for the `AWS::KinesisFirehose::DeliveryStream` resource. The `create` handler invokes only a single

API, `CreateDeliveryStream`, in order for the resource to reach its desired state. Immediately after this API call is made, a `read` request will return the correct desired state. However, the resource still has not reached run-time stabilization because it cannot be used by the customer or downstream resources until the `DeliveryStreamStatus` is `ACTIVE`.

For a more complex example, consider the implementation of the `update` handler for the `AWS::Kinesis::Stream` resource once again. Once the `update` handler has made its final call, to `StartStreamEncryption` or `StopStreamEncryption` as described in [Examples of desired-state stabilization](#), the resource has reached its desired state. However, like the other API calls on the Kinesis resource, the `StreamStatus` will again be set to `UPDATING`. During this period, it has reached its desired state, and customers can even continue using the stream. But it has not yet achieved runtime-stabilization, because additional API calls cannot be made on the resource until the `StreamStatus` gets set to `ACTIVE`.

Handlers must not leak resources

Resource leaking refers to when a handler loses track of the existence of a resource. This happens most often in the following cases:

- A `create` handler is not idempotent. Re-invoking the handler with the same `idempotencyToken` will cause another resource to be created, and the handler is only tracking a single resource.
- A `create` handler creates the resource, but is unable to communicate an identifier for that resource back to CloudFormation. A subsequent `delete` call does not have enough information to delete the resource.
- A bug in the `delete` handler causes the resource to not actually be deleted, but the `delete` handler reports that the resource was successfully deleted.

Handler error codes

One of the following error codes **MUST** be returned from the handler whenever there is a [progress event](#) with an operation status of `FAILED`.

- `AccessDenied`

The customer has insufficient permissions to perform this request.

Type: Terminal

- `AlreadyExists`

The specified resource already existed prior to the execution of this handler. This error is applicable to `create` handlers only.

Type: Terminal

- `GeneralServiceException`

The downstream service generated an error that does not map to any other handler error code.

Type: Terminal

- `InternalFailure`

An unexpected error occurred within the handler.

Type: Terminal

- `InvalidCredentials`

The credentials provided by the user are invalid.

Type: Terminal

- `InvalidRequest`

Invalid input from the user has generated a generic exception.

Type: Terminal

- `NetworkFailure`

The request could not be completed due to networking issues, such as a failure to receive a response from the server.

Type: Retriable

- `NotFound`

The specified resource does not exist, or is in a terminal, inoperable, and irrecoverable state.

Type: Terminal

- `NotStabilized`

The downstream resource failed to complete all of its ready-state checks.

Type: Retriable

- `NotUpdatable`

The user has requested an update to a property defined in the resource type schema as a [create-only property](#). This error is applicable to update handlers only.

Type: Terminal

- `ResourceConflict`

The resource is temporarily unavailable to be acted upon. For example, if the resource is currently undergoing an operation and cannot be acted upon until that operation is finished.

Type: Retriable

- `ServiceInternalError`

The downstream service returned an internal error, typically with a 5XX HTTP Status code.

Type: Retriable

- `ServiceLimitExceeded`

A non-transient resource limit was reached on the service side.

Type: Terminal

- `Throttling`

The request was throttled by a downstream service. Retriable.

Type: Retriable

ProgressEvent object schema

A `ProgressEvent` is a JSON object which represents the current operation status of the handler, the current live state of the resource, and any additional resource information the handler wishes to communicate to the CloudFormation CLI. Each handler **MUST** communicate a progress event to the

CloudFormation CLI under certain circumstances, and SHOULD communicate a progress event under others. For more information, see [Handler Communication Contract](#).

A handler MAY use progress events on a re-invocation to continue work from where it left off. For a detailed discussion of this, see [Progress Chaining, Stabilization and Callback Pattern](#).

Syntax

Below is the syntax for the ProgressEvent object.

```
{
  "OperationStatus": "string",
  "HandlerErrorCode": "string",
  "Message": "string",
  "CallbackContext": "string",
  "CallbackDelaySeconds": "string",
  "ResourceModel": "string",
  "ResourceModels": [
    "string"
  ],
  "NextToken": "string",
}
```

Properties

OperationStatus

Indicates whether the handler has reached a terminal state or is still computing and requires more time to complete.

Values: PENDING | IN_PROGRESS | SUCCESS | FAILED

Required: No

HandlerErrorCode

A handler error code should be provided when the event operation status is FAILED or IN_PROGRESS.

For a list of handler error codes, see [Handler Error Codes](#).

Required: Conditional. A handler error codes MUST be returned from the handler whenever there is a progress event with an operation status of FAILED.

Message

Information which can be shown to users to indicate the nature of a progress transition or callback delay.

Required: No

CallbackContext

Arbitrary information which the handler can return in an event with operation status of IN_PROGRESS, to allow the passing through of additional state or metadata between subsequent retries. For example, to pass through a resource identifier which can be used to continue polling for stabilization.

For more detailed examples, see [Progress Chaining, Stabilization and Callback Pattern](#).

Required: No

CallbackDelaySeconds

A callback will be scheduled with an initial delay of no less than the number of seconds specified.

Set this value to less than 0 to indicate no callback should be made.

Required: No

ResourceModel

Resource model returned by a `read` or `list` operation response for synchronous results, or for final response validation/confirmation by `create`, `update`, and `delete` operations.

Required: No

ResourceModels

List of resource models returned by a `list` operation response for synchronous results.

Required: Conditional. Required for List handlers.

NextToken

Token used to request additional pages of resources from a `list` operation response.

Required: Conditional. Required for List handlers.

Progress chaining, stabilization and callback pattern

Often when you develop CloudFormation resource types, when interacting with web service APIs you need to chain them in sequence to apply the desired state. CloudFormation provides a framework to write these chain patterns. The framework does a lot of the heavy lifting needed to handle error conditions, throttle when calling downstream API, and more. The framework provides callbacks that the handler can use to inspect and change the behavior when making these service calls.

Most web service API calls follows a typical pattern:

1. Initiate the call context for the API.
2. Transform the incoming resource model properties to the underlying service API request.
3. Make the service call.
4. Handle errors. (Optional)
5. Handle stabilization. (Optional, if you need resource to be in a specific state before you apply the next state.)
6. Finalize progress to the next part of the call chain, or indicate successful completion.

In writing the handler, you do not need to do anything special with replay/continuation semantics. The framework ensures that the call chain is effectively resumed from where it was halted. This is essentially useful when the wait time for resource stabilization runs into minutes or even hours.

Sample: Kinesis Stream integration

Here is an example integration against AWS service APIs for a Kinesis Stream. A snippet of the Kinesis resource model is shown below:

```
{  
  "typeName": "AWS::Kinesis::Stream",
```

```
"description": "Resource Type definition for AWS::Kinesis::Stream",
"definitions": {
  ...
},
"properties": {
  "Arn": {
    "type": "string"
  },
  "Name": {
    "type": "string",
    "pattern": "[a-zA-Z0-9_-.]+"
  },
  "RetentionPeriodHours": {
    "type": "integer",
    "minimum": 24,
    "maximum": 168
  },
  "ShardCount": {
    "type": "integer",
    "minimum": 1,
    "maximum": 100000
  },
  "StreamEncryption": {
    "$ref": "#/definitions/AWSKinesisStreamStreamEncryption"
  },
  "Tags": {
    "type": "array",
    "uniqueItems": true,
    "items": {
      "$ref": "#/definitions/Tag"
    },
    "maximum": 50
  }
}
...
}
```

And a sample CloudFormation template for creating this resource in a stack:

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: AWS MetricFilter
Resources:
  KinesisStream:
    Type: AWS::Kinesis::Stream
    Properties:
      ShardCount: 100
      RetentionPeriodHours: 36
      Tags:
        - Key: '1'
          Value: one
        - Key: '2'
          Value: two
      StreamEncryption:
        EncryptionType: KMS
        KeyId: alias/KinesisEncryption
```

For Kinesis, the stream must first be created with a name and shard count, then tags can be applied, followed by encryption. After creating a stream, but before any other configuration can be applied, the stream must be in an ACTIVE state.

Here is the example of the using the progress-chaining and callback pattern to apply state consistently. Note that much of the error handling is delegated to the framework. The CloudFormation CLI provides

some error handling on interpreting errors that can be retried after a delay. The framework provides a fluent API that guides the developer with the right set of calls with strong typing and code completion capabilities in IDEs.

```
public class CreateHandler extends BaseKinesisHandler {
    //
    // The handler is provide with a AmazonWebServicesClientProxy that provides
    // the framework for making calls that returns a ProgressEvent,
    // which can then be chained to perform the next task.
    //
    protected ProgressEvent<ResourceModel, CallbackContext>
        handleRequest(final AmazonWebServicesClientProxy proxy,
                      final ResourceHandlerRequest<ResourceModel> request,
                      final CallbackContext callbackContext,
                      final ProxyClient<KinesisClient> client,
                      final Logger logger) {

        ResourceModel model = request.getDesiredResourceState();
        if (model.getName() == null || model.getName().isEmpty()) {
            model.setName(
                IdentifierUtils.generateResourceIdentifier(
                    "stream-", request.getClientRequestToken(), 128));
        }
        //
        // 1) initiate the call context, we are making createStream API call
        //
        return proxy.initiate(
            "kinesis:CreateStream", client, model, callbackContext)

            //
            // 2) transform Resource model properties to CreateStreamRequest API
            //
            .request((m) ->
                CreateStreamRequest.builder()
                    .streamName(m.getName()).shardCount(m.getShardCount()).build())

            //
            // 3) Make a service call. Handler does not worry about credentials, they
            //     are auto injected
            //
            .call((r, c) ->
                c.injectCredentialsAndInvokeV2(r, c.client()::createStream))

            //
            // provide stabilization callback. The callback is provided with
            // the following parameters
            // a. CreateStreamRequest the we transformed in request()
            // b. CreateStreamResponse that the service returned with successful call
            // c. ProxyClient<Kinesis>, we provided in initiate call
            // d. ResourceModel we provided in initiate call
            // f. CallbackContext callback context.
            //
            .stabilize((_request, _response, _client, _model, _context) ->
                isStreamActive(client1, _model, context))

            //
            // Once ACTIVE return progress
            //
            .progress()

            //
            // we then chain to next state, setting tags on the resource.
            // we receive ProgressEvent object from .progress().
            //
    }
}
```

```
.then(r -> {
    Set<Tag> tags = model.getTags();
    if (tags != null && !tags.isEmpty()) {
        return setTags(proxy, client, model, callbackContext, false, logger);
    }
    return r;
})

//
// we then setRetention...
//
.then(r -> {
    Integer retention = model.getRetentionPeriodHours();
    if (retention != null) {
        return handleRetention(proxy, client, model, DEFAULT_RETENTION,
retention, callbackContext, logger);
    }
    return r;
})

... // other steps

//
// finally we wait for Kinesis stream to be ACTIVE
//
.then((r) -> waitForActive(proxy, client, model, callbackContext))

//
// we then delete to ReadHandler to read the live state and send
// back successful response.
//
.then((r) -> new ReadHandler()
    .handleRequest(proxy, request, callbackContext, client, logger));
}
}
```

How to make other calls

The same pattern shown here for CreateStreamRequest is followed with others as well. Here is code for handleRetention:

```
protected ProgressEvent<ResourceModel, CallbackContext>
    handleRetention(final AmazonWebServicesClientProxy proxy,
        final ProxyClient<KinesisClient> client,
        final ResourceModel model,
        final int previous,
        final int current,
        final CallbackContext callbackContext,
        final Logger logger) {

    if (current > previous) {
        //
        // 1) initiate the call context, we are making IncreaseRetentionPeriod API
call
        //
        return proxy.initiate(
            "kinesis:IncreaseRetentionPeriod:" + getClass().getSimpleName(),
            client, model, callbackContext)
            //
            // 2) transform Resource model properties to
IncreaseStreamRetentionPeriodRequest API
            //
            .request((m) ->
                IncreaseStreamRetentionPeriodRequest.builder()

```

```
        .retentionPeriodHours(current)
        .streamName(m.getName()).build()
    //
    // 3) Make a service call. Handler does not worry about credentials, they
    // are auto injected

    // Add important comments like shown below
    // https://docs.aws.amazon.com/kinesis/latest/APIReference/
API_IncreaseStreamRetentionPeriod.html
    // When applying change if stream is not ACTIVE, we get ResourceInUse.
    // We filter this expectation back off and then re-try to set this.

    //
    // set new retention period
    //
    .call((r, c) -> c.injectCredentialsAndInvokeV2(r,
c.client()::increaseStreamRetentionPeriod))

    //
    // Filter ResourceInUse or LimitExceeded.
    // Currently LimitExceeded is issued even for throttles
    //
    .exceptFilter(this::filterException).progress();
} else {
    return proxy.initiate("kinesis:DecreaseRetentionPeriod:" +
getClass().getSimpleName(), client, model, callbackContext)
    //
    // convert to API model
    //
    .request(m ->
DecreaseStreamRetentionPeriodRequest.builder().retentionPeriodHours(current).streamName(m.getName())
        .build())
    ... // snipped for brevity
    .exceptFilter(this::filterException).progress();
}
}

protected boolean filterException(AwsRequest request,
    Exception e,
    ProxyClient<KinesisClient> client,
    ResourceModel model,
    CallbackContext context) {
    return e instanceof ResourceInUseException ||
        e instanceof LimitExceededException;
}
```

Registering resource types for use in AWS CloudFormation templates

Once you've completed developing your resource type, you'll need to *register* it with CloudFormation in order to make it available for use in CloudFormation operations. From the CloudFormation CLI, use the [submit \(p. 74\)](#) command to register your resource with CloudFormation. The `submit` command does the following:

- Validates the resource schema.
- Packages up the resource project files and uploads them to CloudFormation.

This includes the source code for your resource handlers. These resource handlers run within the CloudFormation service account.

- Runs the unit and contract tests defined in the resource project.
- Determines which handlers have been specified for the resource, to determine how CloudFormation provisions the resource.
- Returns a *registration token* that you can use with the [DescribeTypeRegistration](#) action to track the status of the registration request.

To validate and package your resource project, but not register it with CloudFormation, use the `--dry-run` option for the `submit` command.

You can also register your resource directly using the [RegisterType](#) action.

You must register a resource in each region in which you want to use it.

Use the [ListTypes](#) action for summary information about types that have been registered with CloudFormation, and the [DescribeType](#) action for detailed information about specific registered resource type or resource type version.

Resource type versions

When you register a resource type, you are actually registering a specific *version* of that resource type. You can register multiple versions of a resource type, and specify which version you want to use.

Use the [SetTypeDefaultVersion](#) action to specify the default version of a type. The default version of a resource type will be used in CloudFormation operations.

Resource type scope

Any resource type you register is only visible and usable within the account(s) in which you register it.

Resource type provisioning

During registration, CloudFormation examines which resource handlers have been implemented for the resource. The handlers implemented determine what provisioning actions CloudFormation takes with respect to the resource during various stack operations.

- If the resource type does not contain `create`, `read`, and `delete` handlers, CloudFormation cannot actually provision the resource.
- If the resource type does not contain an `update` handler, CloudFormation cannot update the resource during stack update operations, and will instead replace it.

Deregistering resource types and type versions

To remove a resource type or type version from active use in CloudFormation, you must *deregister* it using the [DeregisterType](#) action. If a type or type version is deregistered, it can no longer be used in CloudFormation operations.

You can deregister a specific resource type version, or the resource type as a whole. To deregister a type, you must individually deregister all registered versions of that type. If a type has only a single registered version, deregistering that version results in the type itself being deregistered. You cannot deregister the default version of a type, unless it is the only registered version of that type, in which case the type itself is deregistered as well.

Deregistering a resource type or resource type version deregisters it in all regions.

You cannot deregister a resource using the CloudFormation console.

Walkthrough: Develop a resource type

In this walkthrough, we'll use the CloudFormation CLI to create a sample resource type, `Example::Testing::WordPress`. This includes modeling the schema, developing the handlers to testing those handlers, all the way to performing a dry run to get the resource type ready to submit to the CloudFormation registry. We'll be coding our new resource type in Java, and using the `us-west-2` region.

Prerequisites

For purposes of this walkthrough, it is assumed you have already set up the CloudFormation CLI and associated tooling for your Java development environment:

[Setting up your environment for developing extensions \(p. 1\)](#)

Create the resource type development project

Before we can actually design and implement our resource type, we'll need to generate a new resource type project, and then import it into our IDE.

Note

This walkthrough uses the Community Edition of the [IntelliJ IDEA](#).

Initiate the project

1. Use the `init` command to create your resource type project and generate the files it requires.

```
$ cfn init
Initializing new project
```

2. The `init` command launches a wizard that walks you through setting up the project, including specifying the resource name. For this walkthrough, specify `Example::Testing::WordPress`.

```
Enter resource type identifier (Organization::Service::Resource):
Example::Testing::WordPress
```

The wizard then enables you to select the appropriate language plugin. Currently, the only language plugin available is for Java:

```
One language plugin found, defaulting to java
```

3. Finally, specify the package name. For this walkthrough, use `com.example.testing.wordpress`

```
Enter a package name (empty for default 'com.example.testing.wordpress'):
com.example.testing.wordpress
Initialized a new project in /workplace/tobflem/example-testing-wordpress
```

Initiating the project includes generating the files needed to develop the resource type. For example:

```
$ ls -l
README.md
example-testing-wordpress.json
lombok.config
pom.xml
rpdk.log
```



```
src
target
template.yml
```

Import the project into your IDE

In order to guarantee that any project dependencies are correctly resolved, you must import the generated project into your IDE with Maven support.

For example, if you are using IntelliJ IDEA, you would need to do the following:

1. From the **File** menu, choose **New**, then choose **Project From Existing Sources**.
2. Navigate to the project directory
3. In the **Import Project** dialog box, choose **Import project from external model** and then choose **Maven**.
4. Choose **Next** and accept any defaults to complete importing the project.

Model the resource type

When you initiate the resource type project, an example resource type schema file is included to help start you modeling your resource type. This is a JSON file named for your resource, and contains an example of a typical resource type schema. In the case of our example resource, the schema file is named `example-testing-wordpress.json`.

1. In your IDE, open `example-testing-wordpress.json`.
2. Paste the following schema in place of the default example schema currently in the file.

This schema defines a resource, `Example::Testing::WordPress`, that provisions a WordPress site. The resource itself contains four properties, only two of which can be set by users: `Name`, and `SubnetId`. The other two properties, `InstanceId` and `PublicIp`, are read-only, meaning they cannot be set by users, but will be assigned during resource creation. Both of these properties also serve as identifiers for the resource when it is provisioned.

As we'll see later in the walkthrough, creating a WordPress site actually requires more information than represented in our resource model. However, we'll be handling that information on behalf of the user in the code for the resource `create` handler.

```
{
  "typeName": "Example::Testing::WordPress",
  "description": "An example resource that creates a website based on WordPress 5.2.2.",
  "sourceUrl": "https://docs.aws.amazon.com/cloudformation-cli/latest/userguide/resource-type-walkthrough.html",
  "properties": {
    "Name": {
      "description": "A name associated with the website.",
      "type": "string",
      "pattern": "^[a-zA-Z0-9]{1,219}\\Z",
      "minLength": 1,
      "maxLength": 219
    },
    "SubnetId": {
      "description": "A subnet in which to host the website.",
      "pattern": "^(subnet-[a-f0-9]{13})|(subnet-[a-f0-9]{8})\\Z",
      "type": "string"
    },
    "InstanceId": {
      "description": "The ID of the instance that backs the WordPress site.",
      "type": "string"
    }
  }
}
```

```
    },
    "PublicIp": {
      "description": "The public IP for the WordPress site.",
      "type": "string"
    }
  },
  "required": [
    "Name",
    "SubnetId"
  ],
  "handlers": {
    "create": {
      "permissions": [
        "ec2:AuthorizeSecurityGroupIngress",
        "ec2:CreateSecurityGroup",
        "ec2>DeleteSecurityGroup",
        "ec2:DescribeInstances",
        "ec2:DescribeSubnets",
        "ec2:CreateTags",
        "ec2:RunInstances"
      ]
    },
    "read": {
      "permissions": [
        "ec2:DescribeInstances"
      ]
    },
    "delete": {
      "permissions": [
        "ec2>DeleteSecurityGroup",
        "ec2:DescribeInstances",
        "ec2:TerminateInstances"
      ]
    }
  },
  "additionalProperties": false,
  "primaryIdentifier": [
    "/properties/PublicIp",
    "/properties/InstanceId"
  ],
  "readOnlyProperties": [
    "/properties/PublicIp",
    "/properties/InstanceId"
  ]
}
```

3. Update the auto-generated files in the resource type package so that they reflect the changes we've made to the resource type schema.

When we first initiated the resource type project, the CloudFormation CLI generated supporting files and code for our resource type. Since we've made changes to the resource type schema, we'll need to regenerate that code to ensure that it reflects the updated schema. To do this, we use the `generate` command:

```
$ cfn generate
Generated files for Example::Testing::WordPress
```

Note

When using Maven, as part of the build process the `generate` command is automatically run before the code is compiled. So your changes will never get out of sync with the generated code.

Be aware the CloudFormation CLI must be in a location Maven/the system can find. For more information, see [Setting up your environment for developing extensions \(p. 1\)](#).

Implement the Resource Handlers

Now that we have our resource type schema specified, we can start implementing the behavior we want the resource type to exhibit during each resource operation. To do this, we'll have to implement the various event handlers, including:

- Adding any necessary dependencies
- Writing code to implement the various resource operation handlers.

Add Dependencies

To actually make WordPress handlers that call the associated EC2 APIs, we need to declare the AWS EC2 SDK as a dependency in Maven's pom.xml file. To enable this, we need to add a dependency on the [AWS SDK for Java](#) to the project.

1. In your IDE, open the project's pom.xml file.
2. Add the following dependency in the dependencies section.

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-java-sdk-ec2</artifactId>
  <version>1.11.606</version>
</dependency>
```

This artifact will be added by Maven from the [Maven Repository](#).

For more information on how to add dependencies, see the [Maven documentation](#).

Note

Depending on your IDE, you may have to take additional steps for your IDE to include the new dependency.

In IntelliJ IDEA, a dialog should appear to enable you to import these changes. We recommend allowing automatic importing.

Implement the Create Handler

With the necessary dependency specified, we can now start writing the handlers that actually implement the resource's functionality. For our example resource, we'll implement just the `create` and `delete` operation handlers.

To create a WordPress site, our resource `create` handler will have to accomplish the following:

- Gather and define inputs that we'll need to create the underlying AWS resources on behalf of the user. These are details we're managing for them, since this is a very high-level resource type.
- Create an EC2 instance using a special AMI vended by Bitnami from the AMI Marketplace that bootstraps WordPress.
- Create a security group that the instance will belong to so you can access the WordPress site from your browser.
- Change the security group rules to dictate what the networking rules are for web access to the WordPress site.
- If something goes wrong with creating the resource, attempt to delete the security group.

Define the CallbackContext

Because our create handler is more complex than simply calling a single API, it takes some time to complete. However, each handler times out after one minute. To work around this issue, we'll write our handlers as state machines. A handler can exit with one of three states: SUCCESS, IN_PROGRESS, and FAILED. To wait on stabilization of underlying resources, we can return an IN_PROGRESS state with a CallbackContext. The CallbackContext will hold details about the current state of the execution. When we return an IN_PROGRESS state and a CallbackContext, CloudFormation will re-invoke the handler and pass the CallbackContext in with the request. You can then make decisions based on what is included in the context.

The CallbackContext is modeled as a POJO so you can define what information you want to pass between state transitions explicitly.

1. In your IDE, open the `CallbackContext.java` file, located in the `src/main/java/com/example/testing/wordpress` folder.
2. Replace the entire contents of the `CallbackContext.java` file with the following code.

```
package com.example.testing.wordpress;

import com.amazonaws.services.ec2.model.Instance;

import java.util.List;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Builder(toBuilder = true)
@Data
@NoArgsConstructor
@AllArgsConstructor
public class CallbackContext {
    private Instance instance;
    private Integer stabilizationRetriesRemaining;
    private List<String> instanceSecurityGroups;
}
```

Code the Create Handler

1. In your IDE, open the `CreateHandler.java` file, located in the `src/main/java/com/example/testing/wordpress/CreateHandler.java` folder.
2. Replace the entire contents of the `CreateHandler.java` file with the following code.

```
package com.example.testing.wordpress;

import software.amazon.cloudformation.proxy.AmazonWebServicesClientProxy;
import software.amazon.cloudformation.proxy.Logger;
import software.amazon.cloudformation.proxy.OperationStatus;
import software.amazon.cloudformation.proxy.ProgressEvent;
import software.amazon.cloudformation.proxy.ResourceHandlerRequest;
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.AuthorizeSecurityGroupIngressRequest;
import com.amazonaws.services.ec2.model.CreateSecurityGroupRequest;
import com.amazonaws.services.ec2.model.DeleteSecurityGroupRequest;
import com.amazonaws.services.ec2.model.DescribeInstancesRequest;
import com.amazonaws.services.ec2.model.DescribeInstancesResult;
```

CloudFormation Command Line Interface
User Guide for Extension Development
Implement the Resource Handlers

```
import com.amazonaws.services.ec2.model.DescribeSubnetsRequest;
import com.amazonaws.services.ec2.model.DescribeSubnetsResult;
import com.amazonaws.services.ec2.model.Instance;
import com.amazonaws.services.ec2.model.InstanceNetworkInterfaceSpecification;
import com.amazonaws.services.ec2.model.IpPermission;
import com.amazonaws.services.ec2.model.IpRange;
import com.amazonaws.services.ec2.model.Reservation;
import com.amazonaws.services.ec2.model.RunInstancesRequest;
import com.amazonaws.services.ec2.model.Subnet;
import com.amazonaws.services.ec2.model.Tag;
import com.amazonaws.services.ec2.model.TagSpecification;

import java.util.List;
import java.util.UUID;

public class CreateHandler extends BaseHandler<CallbackContext> {
    private static final String SUPPORTED_REGION = "us-west-2";
    private static final String WORDPRESS_AMI_ID = "ami-04fb0368671b6f138";
    private static final String INSTANCE_TYPE = "m4.large";
    private static final String SITE_NAME_TAG_KEY = "Name";
    private static final String AVAILABLE_INSTANCE_STATE = "running";
    private static final int NUMBER_OF_STATE_POLL_RETRIES = 60;
    private static final int POLL_RETRY_DELAY_IN_MS = 5000;
    private static final String TIMED_OUT_MESSAGE = "Timed out waiting for instance to
become available.";

    private AmazonWebServicesClientProxy clientProxy;
    private AmazonEC2 ec2Client;

    @Override
    public ProgressEvent<ResourceModel, CallbackContext> handleRequest(
        final AmazonWebServicesClientProxy proxy,
        final ResourceHandlerRequest<ResourceModel> request,
        final CallbackContext callbackContext,
        final Logger logger) {

        final ResourceModel model = request.getDesiredResourceState();

        clientProxy = proxy;
        ec2Client =
AmazonEC2ClientBuilder.standard().withRegion(SUPPORTED_REGION).build();
        final CallbackContext currentContext = callbackContext == null ?

CallbackContext.builder().stabilizationRetriesRemaining(NUMBER_OF_STATE_POLL_RETRIES).build() :
            callbackContext;

        // This Lambda will continually be re-invoked with the current state of the
instance, finally succeeding when state stabilizes.
        return createInstanceAndUpdateProgress(model, currentContext);
    }

    private ProgressEvent<ResourceModel, CallbackContext>
createInstanceAndUpdateProgress(ResourceModel model, CallbackContext callbackContext) {
        // This Lambda will continually be re-invoked with the current state of the
instance, finally succeeding when state stabilizes.
        final Instance instanceStateSoFar = callbackContext.getInstance();

        if (callbackContext.getStabilizationRetriesRemaining() == 0) {
            throw new RuntimeException(TIMED_OUT_MESSAGE);
        }

        if (instanceStateSoFar == null) {
            return ProgressEvent.<ResourceModel, CallbackContext>builder()
                .resourceModel(model)
                .status(OperationStatus.IN_PROGRESS)
                .callbackContext(CallbackContext.builder()
```

CloudFormation Command Line Interface
User Guide for Extension Development
Implement the Resource Handlers

```
                .instance(createEC2Instance(model))
        .stabilizationRetriesRemaining(NUMBER_OF_STATE_POLL_RETRIES)
            .build()
        } else if
        (instanceStateSoFar.getState().getName().equals(AVAILABLE_INSTANCE_STATE)) {
            model.setInstanceId(instanceStateSoFar.getInstanceId());
            model.setPublicIp(instanceStateSoFar.getPublicIpAddress());
            return ProgressEvent.<ResourceModel, CallbackContext>builder()
                .resourceModel(model)
                .status(OperationStatus.SUCCESS)
                .build();
        } else {
            try {
                Thread.sleep(POLL_RETRY_DELAY_IN_MS);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            return ProgressEvent.<ResourceModel, CallbackContext>builder()
                .resourceModel(model)
                .status(OperationStatus.IN_PROGRESS)
                .callbackContext(CallbackContext.builder())
                .instance(updatedInstanceProgress(instanceStateSoFar.getInstanceId()))
                .stabilizationRetriesRemaining(callbackContext.getStabilizationRetriesRemaining() - 1)
                    .build()
            .build();
        }
    }

    private Instance createEC2Instance(ResourceModel model) {
        final String securityGroupId = createSecurityGroupForInstance(model);
        final RunInstancesRequest runInstancesRequest = new RunInstancesRequest()
            .withInstanceType(INSTANCE_TYPE)
            .withImageId(WORDPRESS_AMI_ID)
            .withNetworkInterfaces(new InstanceNetworkInterfaceSpecification()
                .withAssociatePublicIpAddress(true)
                .withDeviceIndex(0)
                .withGroups(securityGroupId)
                .withSubnetId(model.getSubnetId()))
            .withMaxCount(1)
            .withMinCount(1)
            .withTagSpecifications(buildTagFromSiteName(model.getName()));

        try {
            return clientProxy.injectCredentialsAndInvoke(runInstancesRequest,
                ec2Client::runInstances)
                .getReservation()
                .getInstances()
                .stream()
                .findFirst()
                .orElse(new Instance());
        } catch (Throwable e) {
            attemptToCleanUpSecurityGroup(securityGroupId);
            throw new RuntimeException(e);
        }
    }

    private String createSecurityGroupForInstance(ResourceModel model) {
        String vpcId;
        try {
            vpcId = getVpcIdFromSubnetId(model.getSubnetId());
        } catch (Throwable e) {
```

CloudFormation Command Line Interface
User Guide for Extension Development
Implement the Resource Handlers

```
        throw new RuntimeException(e);
    }

    final String securityGroupName = model.getName() + "-" +
UUID.randomUUID().toString();

    final CreateSecurityGroupRequest createSecurityGroupRequest = new
CreateSecurityGroupRequest()
        .withGroupName(securityGroupName)
        .withDescription("Created for the test WordPress blog: " + model.getName())
        .withVpcId(vpcId);

    final String securityGroupId =
        clientProxy.injectCredentialsAndInvoke(createSecurityGroupRequest,
ec2Client::createSecurityGroup)
            .getGroupId();

    final AuthorizeSecurityGroupIngressRequest authorizeSecurityGroupIngressRequest =
new AuthorizeSecurityGroupIngressRequest()
        .withGroupId(securityGroupId)
        .withIpPermissions(openHTTP(), openHTTPS());

    clientProxy.injectCredentialsAndInvoke(authorizeSecurityGroupIngressRequest,
ec2Client::authorizeSecurityGroupIngress);

    return securityGroupId;
}

private String getVpcIdFromSubnetId(String subnetId) throws Throwable {
    final DescribeSubnetsRequest describeSubnetsRequest = new
DescribeSubnetsRequest()
        .withSubnetIds(subnetId);

    final DescribeSubnetsResult describeSubnetsResult =
        clientProxy.injectCredentialsAndInvoke(describeSubnetsRequest,
ec2Client::describeSubnets);

    return describeSubnetsResult.getSubnets()
        .stream()
        .map(Subnet::getVpcId)
        .findFirst()
        .orElseThrow(() -> {
            throw new RuntimeException("Subnet " + subnetId +
" not found");
        });
}

private IpPermission openHTTP() {
    return new IpPermission().withIpProtocol("tcp")
        .withFromPort(80)
        .withToPort(80)
        .withIpv4Ranges(new IpRange().withCidrIp("0.0.0.0/0"));
}

private IpPermission openHTTPS() {
    return new IpPermission().withIpProtocol("tcp")
        .withFromPort(443)
        .withToPort(443)
        .withIpv4Ranges(new IpRange().withCidrIp("0.0.0.0/0"));
}

private TagSpecification buildTagFromSiteName(String siteName) {
    return new TagSpecification()
        .withResourceType("instance")
        .withTags(new Tag().withKey(SITE_NAME_TAG_KEY).withValue(siteName));
}
}
```

```
private Instance updatedInstanceProgress(String instanceId) {
    DescribeInstancesRequest describeInstancesRequest;
    DescribeInstancesResult describeInstancesResult;

    describeInstancesRequest = new
DescribeInstancesRequest().withInstanceIds(instanceId);
    describeInstancesResult =
clientProxy.injectCredentialsAndInvoke(describeInstancesRequest,
ec2Client::describeInstances);
    return describeInstancesResult.getReservations()
        .stream()
        .map(Reservation::getInstances)
        .flatMap(List::stream)
        .findFirst()
        .orElse(new Instance());
}

private void attemptToCleanUpSecurityGroup(String securityGroupId) {
    final DeleteSecurityGroupRequest deleteSecurityGroupRequest = new
DeleteSecurityGroupRequest().withGroupId(securityGroupId);
    clientProxy.injectCredentialsAndInvoke(deleteSecurityGroupRequest,
ec2Client::deleteSecurityGroup);
}
}
```

Update the Create Handler Unit Test

Since our resource type is a high-level abstraction, a lot of implementation behavior isn't apparent by the name alone. As such, we'll need to make some additions to our unit tests so that we're not calling the live APIs that are necessary to create the WordPress site.

1. In your IDE, open the `CreateHandlerTest.java` file, located in the `src/test/java/com/example/testing/wordpress` folder.
2. Replace the entire contents of the `CreateHandlerTest.java` file with the following code.

```
package com.example.testing.wordpress;

import software.amazon.cloudformation.proxy.AmazonWebServicesClientProxy;
import software.amazon.cloudformation.proxy.Logger;
import software.amazon.cloudformation.proxy.OperationStatus;
import software.amazon.cloudformation.proxy.ProgressEvent;
import software.amazon.cloudformation.proxy.ResourceHandlerRequest;
import com.amazonaws.services.ec2.model.AuthorizeSecurityGroupIngressRequest;
import com.amazonaws.services.ec2.model.AuthorizeSecurityGroupIngressResult;
import com.amazonaws.services.ec2.model.CreateSecurityGroupRequest;
import com.amazonaws.services.ec2.model.CreateSecurityGroupResult;
import com.amazonaws.services.ec2.model.DescribeInstancesRequest;
import com.amazonaws.services.ec2.model.DescribeInstancesResult;
import com.amazonaws.services.ec2.model.DescribeSubnetsRequest;
import com.amazonaws.services.ec2.model.DescribeSubnetsResult;
import com.amazonaws.services.ec2.model.GroupIdentifier;
import com.amazonaws.services.ec2.model.Instance;
import com.amazonaws.services.ec2.model.InstanceState;
import com.amazonaws.services.ec2.model.Reservation;
import com.amazonaws.services.ec2.model.RunInstancesRequest;
import com.amazonaws.services.ec2.model.RunInstancesResult;
import com.amazonaws.services.ec2.model.Subnet;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
```


CloudFormation Command Line Interface
User Guide for Extension Development
Implement the Resource Handlers

```
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.doReturn;
import static org.mockito.Mockito.mock;

@ExtendWith(MockitoExtension.class)
public class CreateHandlerTest {
    private static String EXPECTED_TIMEOUT_MESSAGE = "Timed out waiting for instance to
    become available.";

    @Mock
    private AmazonWebServicesClientProxy proxy;

    @Mock
    private Logger logger;

    @BeforeEach
    public void setup() {
        proxy = mock(AmazonWebServicesClientProxy.class);
        logger = mock(Logger.class);
    }

    @Test
    public void testSuccessState() {
        final InstanceState inProgressState = new InstanceState().withName("running");
        final GroupIdentifier group = new GroupIdentifier().withGroupId("sg-1234");
        final Instance instance = new
Instance().withInstanceId("i-1234").withState(inProgressState).withPublicIpAddress("54.0.0.0").with

        final CreateHandler handler = new CreateHandler();

        final ResourceModel model = ResourceModel.builder()
            .name("MyWordPressSite")
            .subnetId("subnet-1234")
            .build();

        final ResourceModel desiredOutputModel = ResourceModel.builder()
            .instanceId("i-1234")
            .publicIp("54.0.0.0")
            .name("MyWordPressSite")
            .subnetId("subnet-1234")
            .build();

        final ResourceHandlerRequest<ResourceModel> request =
ResourceHandlerRequest.<ResourceModel>builder()
            .desiredResourceState(model)
            .build();

        final CallbackContext context = CallbackContext.builder()
            .stabilizationRetriesRemaining(1)
            .instance(instance)
            .build();

        final ProgressEvent<ResourceModel, CallbackContext> response
            = handler.handleRequest(proxy, request, context, logger);

        assertThat(response).isNotNull();
        assertThat(response.getStatus()).isEqualTo(OperationStatus.SUCCESS);
        assertThat(response.getCallbackContext()).isNull();
        assertThat(response.getCallbackDelaySeconds()).isEqualTo(0);
        assertThat(response.getResourceModel()).isEqualTo(desiredOutputModel);
        assertThat(response.getResourceModels()).isNull();
    }
}
```

CloudFormation Command Line Interface
User Guide for Extension Development
Implement the Resource Handlers

```
        assertThat(response.getMessage()).isNull();
        assertThat(response.getErrorCode()).isNull();
    }

    @Test
    public void testInProgressStateInstanceCreationNotInvoked() {
        final InstanceState inProgressState = new InstanceState().withName("in-
progress");
        final GroupIdentifier group = new GroupIdentifier().withGroupId("sg-1234");
        final Instance instance = new
Instance().withState(inProgressState).withPublicIpAddress("54.0.0.0").withSecurityGroups(group);
        doReturn(new DescribeSubnetsResult().withSubnets(new
Subnet().withVpcId("vpc-1234")))
        .when(proxy).injectCredentialsAndInvoke(any(DescribeSubnetsRequest.class),
any());
        doReturn(new RunInstancesResult().withReservation(new
Reservation().withInstances(instance)))
        .when(proxy).injectCredentialsAndInvoke(any(RunInstancesRequest.class),
any());
        doReturn(new
CreateSecurityGroupResult().withGroupId("sg-1234"))
        .when(proxy).injectCredentialsAndInvoke(any(CreateSecurityGroupRequest.class),
any());
        doReturn(new
AuthorizeSecurityGroupIngressResult())
        .when(proxy).injectCredentialsAndInvoke(any(AuthorizeSecurityGroupIngressRequest.class),
any());

        final CreateHandler handler = new CreateHandler();

        final ResourceModel model =
ResourceModel.builder().name("MyWordPressSite").subnetId("subnet-1234").build();

        final ResourceHandlerRequest<ResourceModel> request =
ResourceHandlerRequest.<ResourceModel>builder()
            .desiredResourceState(model)
            .build();

        final ProgressEvent<ResourceModel, CallbackContext> response
            = handler.handleRequest(proxy, request, null, logger);

        final CallbackContext desiredOutputContext = CallbackContext.builder()
            .stabilizationRetriesRemaining(60)
            .instance(instance)
            .build();

        assertThat(response).isNotNull();
        assertThat(response.getStatus()).isEqualTo(OperationStatus.IN_PROGRESS);

        assertThat(response.getCallbackContext()).isEqualTo(ComparingFieldByField(desiredOutputContext));
        assertThat(response.getCallbackDelaySeconds()).isEqualTo(0);

        assertThat(response.getResourceModel()).isEqualTo(request.getDesiredResourceState());
        assertThat(response.getResourceModels()).isNull();
        assertThat(response.getMessage()).isNull();
        assertThat(response.getErrorCode()).isNull();
    }

    @Test
    public void testInProgressStateInstanceCreationInvoked() {
        final InstanceState inProgressState = new InstanceState().withName("in-
progress");
        final GroupIdentifier group = new GroupIdentifier().withGroupId("sg-1234");
        final Instance instance = new
Instance().withState(inProgressState).withPublicIpAddress("54.0.0.0").withSecurityGroups(group);
        final DescribeInstancesResult describeInstancesResult =
            new DescribeInstancesResult().withReservations(new
Reservation().withInstances(instance));
```

CloudFormation Command Line Interface
User Guide for Extension Development
Implement the Resource Handlers

```
doReturn(describeInstancesResult).when(proxy).injectCredentialsAndInvoke(any(DescribeInstancesRequest.class), any());

    final CreateHandler handler = new CreateHandler();

    final ResourceModel model =
ResourceModel.builder().name("MyWordPressSite").subnetId("subnet-1234").build();

    final ResourceHandlerRequest<ResourceModel> request =
ResourceHandlerRequest.<ResourceModel>builder()
    .desiredResourceState(model)
    .build();

    final CallbackContext context = CallbackContext.builder()
    .stabilizationRetriesRemaining(60)
    .instance(instance)
    .build();

    final ProgressEvent<ResourceModel, CallbackContext> response
    = handler.handleRequest(proxy, request, context, logger);

    final CallbackContext desiredOutputContext = CallbackContext.builder()
.stabilizationRetriesRemaining(59)
    .instance(instance)
    .build();

    assertThat(response).isNotNull();
    assertThat(response.getStatus()).isEqualTo(OperationStatus.IN_PROGRESS);

    assertThat(response.getCallbackContext()).isEqualToComparingFieldByField(desiredOutputContext);
    assertThat(response.getCallbackDelaySeconds()).isEqualTo(0);

    assertThat(response.getResourceModel()).isEqualTo(request.getDesiredResourceState());
    assertThat(response.getResourceModels()).isNull();
    assertThat(response.getMessage()).isNull();
    assertThat(response.getErrorCode()).isNull();
}

@Test
public void testStabilizationTimeout() {
    final CreateHandler handler = new CreateHandler();

    final ResourceModel model =
ResourceModel.builder().name("MyWordPressSite").subnetId("subnet-1234").build();

    final ResourceHandlerRequest<ResourceModel> request =
ResourceHandlerRequest.<ResourceModel>builder()
    .desiredResourceState(model)
    .build();

    final CallbackContext context = CallbackContext.builder()
    .stabilizationRetriesRemaining(0)
    .instance(new
Instance().withState(new InstanceState().withName("in-progress")))
    .build();

    try {
        handler.handleRequest(proxy, request, context, logger);
    } catch (RuntimeException e) {
        assertThat(e.getMessage()).isEqualTo(EXPECTED_TIMEOUT_MESSAGE);
    }
}
}
```

Implement the Delete Handler

We'll also need to implement a delete handler. At a high level, the delete handler needs to accomplish the following:

1. Find the security groups attached to the EC2 instance that is hosting the WordPress page.
2. Delete the instance.
3. Delete the security groups.

Again, we'll implement the delete handler as a state machine.

Code the Delete Handler

1. In your IDE, open the `DeleteHandler.java` file, located in the `src/main/java/com/example/testing/wordpress` folder.
2. Replace the entire contents of the `DeleteHandler.java` file with the following code.

```
package com.example.testing.wordpress;

import software.amazon.cloudformation.proxy.AmazonWebServicesClientProxy;
import software.amazon.cloudformation.proxy.HandlerErrorCode;
import software.amazon.cloudformation.proxy.Logger;
import software.amazon.cloudformation.proxy.OperationStatus;
import software.amazon.cloudformation.proxy.ProgressEvent;
import software.amazon.cloudformation.proxy.ResourceHandlerRequest;
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.DeleteSecurityGroupRequest;
import com.amazonaws.services.ec2.model.DescribeInstancesRequest;
import com.amazonaws.services.ec2.model.DescribeInstancesResult;
import com.amazonaws.services.ec2.model.GroupIdentifier;
import com.amazonaws.services.ec2.model.Instance;
import com.amazonaws.services.ec2.model.Reservation;
import com.amazonaws.services.ec2.model.TerminateInstancesRequest;

import java.util.List;
import java.util.stream.Collectors;

public class DeleteHandler extends BaseHandler<CallbackContext> {
    private static final String SUPPORTED_REGION = "us-west-2";
    private static final String DELETED_INSTANCE_STATE = "terminated";
    private static final int NUMBER_OF_STATE_POLL_RETRIES = 60;
    private static final int POLL_RETRY_DELAY_IN_MS = 5000;
    private static final String TIMED_OUT_MESSAGE = "Timed out waiting for instance to
    terminate.";

    private AmazonWebServicesClientProxy clientProxy;
    private AmazonEC2 ec2Client;

    @Override
    public ProgressEvent<ResourceModel, CallbackContext> handleRequest (
        final AmazonWebServicesClientProxy proxy,
        final ResourceHandlerRequest<ResourceModel> request,
        final CallbackContext callbackContext,
        final Logger logger) {

        final ResourceModel model = request.getDesiredResourceState();

        clientProxy = proxy;
        ec2Client =
            AmazonEC2ClientBuilder.standard().withRegion(SUPPORTED_REGION).build();
```

CloudFormation Command Line Interface
User Guide for Extension Development
Implement the Resource Handlers

```
        final CallbackContext currentContext = callbackContext == null ?
CallbackContext.builder().stabilizationRetriesRemaining(NUMBER_OF_STATE_POLL_RETRIES).build() :
                                callbackContext;

        // This Lambda will continually be re-invoked with the current state of the
instance, finally succeeding when state stabilizes.
        return deleteInstanceAndUpdateProgress(model, currentContext);
    }

    private ProgressEvent<ResourceModel, CallbackContext>
deleteInstanceAndUpdateProgress(ResourceModel model, CallbackContext callbackContext) {

        if (callbackContext.getStabilizationRetriesRemaining() == 0) {
            throw new RuntimeException(TIMED_OUT_MESSAGE);
        }

        if (callbackContext.getInstanceSecurityGroups() == null) {
            final Instance currentInstanceState =
currentInstanceState(model.getInstanceId());

            if (DELETED_INSTANCE_STATE.equals(currentInstanceState.getState().getName()))
{
                return ProgressEvent.<ResourceModel, CallbackContext>builder()
                    .status(OperationStatus.FAILED)
                    .errorCode(HandlerErrorCode.NotFound)
                    .build();
            }

            final List<String> instanceSecurityGroups = currentInstanceState
                .getSecurityGroups()
                .stream()
                .map(GroupIdentifier::getGroupId)
                .collect(Collectors.toList());

            return ProgressEvent.<ResourceModel, CallbackContext>builder()
                .resourceModel(model)
                .status(OperationStatus.IN_PROGRESS)
                .callbackContext(CallbackContext.builder()

.stabilizationRetriesRemaining(NUMBER_OF_STATE_POLL_RETRIES)

.instanceSecurityGroups(instanceSecurityGroups)
                                .build())
                .build();
        }

        if (callbackContext.getInstance() == null) {
            return ProgressEvent.<ResourceModel, CallbackContext>builder()
                .resourceModel(model)
                .status(OperationStatus.IN_PROGRESS)
                .callbackContext(CallbackContext.builder()

.instance(deleteInstance(model.getInstanceId()))

.instanceSecurityGroups(callbackContext.getInstanceSecurityGroups())

.stabilizationRetriesRemaining(NUMBER_OF_STATE_POLL_RETRIES)
                                .build())
                .build();
        } else if
(callbackContext.getInstance().getState().getName().equals(DELETED_INSTANCE_STATE)) {
            callbackContext.getInstanceSecurityGroups().forEach(this::deleteSecurityGroup);
            return ProgressEvent.<ResourceModel, CallbackContext>builder()
                .resourceModel(model)
```

CloudFormation Command Line Interface
User Guide for Extension Development
Implement the Resource Handlers

```
        .status(OperationStatus.SUCCESS)
        .build();
    } else {
        try {
            Thread.sleep(POLL_RETRY_DELAY_IN_MS);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        return ProgressEvent.<ResourceModel, CallbackContext>builder()
            .resourceModel(model)
            .status(OperationStatus.IN_PROGRESS)
            .callbackContext(CallbackContext.builder()

.instance(currentInstanceState(model.getInstanceId()))

.instanceSecurityGroups(callbackContext.getInstanceSecurityGroups())

.stabilizationRetriesRemaining(callbackContext.getStabilizationRetriesRemaining() - 1)
            .build())
            .build();
    }
}

private Instance deleteInstance(String instanceId) {
    final TerminateInstancesRequest terminateInstancesRequest = new
    TerminateInstancesRequest().withInstanceIds(instanceId);
    return clientProxy.injectCredentialsAndInvoke(terminateInstancesRequest,
    ec2Client::terminateInstances)
        .getTerminatingInstances()
        .stream()
        .map(instance -> new
    Instance().withState(instance.getCurrentState()).withInstanceId(instance.getInstanceId()))
        .findFirst()
        .orElse(new Instance());
}

private Instance currentInstanceState(String instanceId) {
    DescribeInstancesRequest describeInstancesRequest;
    DescribeInstancesResult describeInstancesResult;

    describeInstancesRequest = new
    DescribeInstancesRequest().withInstanceIds(instanceId);
    describeInstancesResult =
    clientProxy.injectCredentialsAndInvoke(describeInstancesRequest,
    ec2Client::describeInstances);
    return describeInstancesResult.getReservations()
        .stream()
        .map(Reservation::getInstances)
        .flatMap(List::stream)
        .findFirst()
        .orElse(new Instance());
}

private void deleteSecurityGroup(String securityGroupId) {
    final DeleteSecurityGroupRequest deleteSecurityGroupRequest = new
    DeleteSecurityGroupRequest().withGroupId(securityGroupId);
    clientProxy.injectCredentialsAndInvoke(deleteSecurityGroupRequest,
    ec2Client::deleteSecurityGroup);
}
}
```

Update the Delete Handler Unit Test

We'll also need to update the unit test for the delete handler.

1. In your IDE, open the `DeleteHandlerTest.java` file, located in the `src/test/java/com/example/testing/wordpress` folder.
2. Replace the entire contents of the `DeleteHandlerTest.java` file with the following code.

```
package com.example.testing.wordpress;

import software.amazon.cloudformation.proxy.AmazonWebServicesClientProxy;
import software.amazon.cloudformation.proxy.HandlerErrorCode;
import software.amazon.cloudformation.proxy.Logger;
import software.amazon.cloudformation.proxy.OperationStatus;
import software.amazon.cloudformation.proxy.ProgressEvent;
import software.amazon.cloudformation.proxy.ResourceHandlerRequest;
import com.amazonaws.services.ec2.model.DeleteSecurityGroupRequest;
import com.amazonaws.services.ec2.model.DeleteSecurityGroupResult;
import com.amazonaws.services.ec2.model.DescribeInstancesRequest;
import com.amazonaws.services.ec2.model.DescribeInstancesResult;
import com.amazonaws.services.ec2.model.GroupIdentifier;
import com.amazonaws.services.ec2.model.Instance;
import com.amazonaws.services.ec2.model.InstanceState;
import com.amazonaws.services.ec2.model.InstanceStateChange;
import com.amazonaws.services.ec2.model.Reservation;
import com.amazonaws.services.ec2.model.TerminateInstancesRequest;
import com.amazonaws.services.ec2.model.TerminateInstancesResult;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.Arrays;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.doReturn;
import static org.mockito.Mockito.mock;

@ExtendWith(MockitoExtension.class)
public class DeleteHandlerTest {
    private static String EXPECTED_TIMEOUT_MESSAGE = "Timed out waiting for instance to
    terminate.";

    @Mock
    private AmazonWebServicesClientProxy proxy;

    @Mock
    private Logger logger;

    @BeforeEach
    public void setup() {
        proxy = mock(AmazonWebServicesClientProxy.class);
        logger = mock(Logger.class);
    }

    @Test
    public void testSuccessState() {
        final DeleteSecurityGroupResult deleteSecurityGroupResult = new
        DeleteSecurityGroupResult();
    }
}
```

CloudFormation Command Line Interface
User Guide for Extension Development
Implement the Resource Handlers

```
doReturn(deleteSecurityGroupResult).when(proxy).injectCredentialsAndInvoke(any(DeleteSecurityGroupRequest.class), any());

    final DeleteHandler handler = new DeleteHandler();

    final ResourceModel model = ResourceModel.builder().instanceId("i-1234").build();

    final ResourceHandlerRequest<ResourceModel> request =
ResourceHandlerRequest.<ResourceModel>builder()
    .desiredResourceState(model)
    .build();

    final CallbackContext context = CallbackContext.builder()
                                                .stabilizationRetriesRemaining(1)

.instantiateSecurityGroups(Arrays.asList("sg-1234"))
                                                .instance(new
Instance().withState(new InstanceState().withName("terminated")))
                                                .build());

    final ProgressEvent<ResourceModel, CallbackContext> response
        = handler.handleRequest(proxy, request, context, logger);

    assertThat(response).isNotNull();
    assertThat(response.getStatus()).isEqualTo(OperationStatus.SUCCESS);
    assertThat(response.getCallbackContext()).isNull();
    assertThat(response.getCallbackDelaySeconds()).isEqualTo(0);

assertThat(response.getResourceModel()).isEqualTo(request.getDesiredResourceState());
    assertThat(response.getResourceModels()).isNull();
    assertThat(response.getMessage()).isNull();
    assertThat(response.getErrorCode()).isNull();
}

@Test
public void testHandlerInvokedWhenInstanceIsAlreadyTerminated() {
    final DescribeInstancesResult describeInstancesResult =
        new DescribeInstancesResult().withReservations(new
Reservation().withInstances(new Instance().withState(new
InstanceState().withName("terminated"))

        .withSecurityGroups(new GroupIdentifier().withGroupId("sg-1234"))));

doReturn(describeInstancesResult).when(proxy).injectCredentialsAndInvoke(any(DescribeInstancesRequest.class), any());

    final DeleteHandler handler = new DeleteHandler();

    final ResourceModel model = ResourceModel.builder().instanceId("i-1234").build();

    final ResourceHandlerRequest<ResourceModel> request =
ResourceHandlerRequest.<ResourceModel>builder()
    .desiredResourceState(model)
    .build();

    final ProgressEvent<ResourceModel, CallbackContext> response
        = handler.handleRequest(proxy, request, null, logger);

    assertThat(response).isNotNull();
    assertThat(response.getStatus()).isEqualTo(OperationStatus.FAILED);
    assertThat(response.getCallbackContext()).isNull();
    assertThat(response.getCallbackDelaySeconds()).isEqualTo(0);
    assertThat(response.getResourceModel()).isNull();
    assertThat(response.getResourceModels()).isNull();
    assertThat(response.getMessage()).isNull();
}
```


CloudFormation Command Line Interface
User Guide for Extension Development
Implement the Resource Handlers

```
        assertThat(response.getErrorCode()).isEqualTo(HandlerErrorCode.NotFound);
    }

    @Test
    public void testInProgressStateSecurityGroupsNotGathered() {
        final DescribeInstancesResult describeInstancesResult =
            new DescribeInstancesResult().withReservations(new
Reservation().withInstances(new Instance().withState(new
InstanceState().withName("running")))

                .withSecurityGroups(new GroupIdentifier().withGroupId("sg-1234"))));

doReturn(describeInstancesResult).when(proxy).injectCredentialsAndInvoke(any(DescribeInstancesRequest.class),
any());

        final DeleteHandler handler = new DeleteHandler();

        final ResourceModel model = ResourceModel.builder().instanceId("i-1234").build();

        final ResourceHandlerRequest<ResourceModel> request =
ResourceHandlerRequest.<ResourceModel>builder()
            .desiredResourceState(model)
            .build();

        final ProgressEvent<ResourceModel, CallbackContext> response
            = handler.handleRequest(proxy, request, null, logger);

        final CallbackContext desiredOutputContext = CallbackContext.builder()

.stabilizationRetriesRemaining(60)

.instanceSecurityGroups(Arrays.asList("sg-1234"))

                .build();

        assertThat(response).isNotNull();
        assertThat(response.getStatus()).isEqualTo(OperationStatus.IN_PROGRESS);

assertThat(response.getCallbackContext()).isEqualTo(ComparingFieldByFieldComparer.comparing(desiredOutputContext));
        assertThat(response.getCallbackDelaySeconds()).isEqualTo(0);

assertThat(response.getResourceModel()).isEqualTo(request.getDesiredResourceState());
        assertThat(response.getResourceModels()).isNull();
        assertThat(response.getMessage()).isNull();
        assertThat(response.getErrorCode()).isNull();
    }

    @Test
    public void testInProgressStateSecurityGroupsGathered() {
        final InstanceState inProgressState = new InstanceState().withName("in-
progress");
        final TerminateInstancesResult terminateInstancesResult =
            new TerminateInstancesResult().withTerminatingInstances(new
InstanceStateChange().withCurrentState(inProgressState));

doReturn(terminateInstancesResult).when(proxy).injectCredentialsAndInvoke(any(TerminateInstancesRequest.class),
any());

        final DeleteHandler handler = new DeleteHandler();

        final ResourceModel model = ResourceModel.builder().instanceId("i-1234").build();

        final ResourceHandlerRequest<ResourceModel> request =
ResourceHandlerRequest.<ResourceModel>builder()
            .desiredResourceState(model)
            .build();

        final CallbackContext context = CallbackContext.builder()
```

CloudFormation Command Line Interface
User Guide for Extension Development
Implement the Resource Handlers

```

                                                                                   .stabilizationRetriesRemaining(60)
.instanceSecurityGroups(Arrays.asList("sg-1234"))
                                                                                   .build();

    final ProgressEvent<ResourceModel, CallbackContext> response
        = handler.handleRequest(proxy, request, context, logger);

    final CallbackContext desiredOutputContext = CallbackContext.builder()
.stabilizationRetriesRemaining(60)
.instanceSecurityGroups(context.getInstanceSecurityGroups())
Instance().withState(inProgressState)
                                                                                   .instance(new
                                                                                   .build());

    assertThat(response).isNotNull();
    assertThat(response.getStatus()).isEqualTo(OperationStatus.IN_PROGRESS);

assertThat(response.getCallbackContext()).isEqualToComparingFieldByField(desiredOutputContext);
    assertThat(response.getCallbackDelaySeconds()).isEqualTo(0);

assertThat(response.getResourceModel()).isEqualTo(request.getDesiredResourceState());
    assertThat(response.getResourceModels()).isNull();
    assertThat(response.getMessage()).isNull();
    assertThat(response.getErrorCode()).isNull();
}

@Test
public void testInProgressStateInstanceTerminationInvoked() {
    final InstanceState inProgressState = new InstanceState().withName("in-
progress");
    final GroupIdentifier group = new GroupIdentifier().withGroupId("sg-1234");
    final Instance instance = new
Instance().withState(inProgressState).withSecurityGroups(group);
    final DescribeInstancesResult describeInstancesResult =
        new DescribeInstancesResult().withReservations(new
Reservation().withInstances(instance));

doReturn(describeInstancesResult).when(proxy).injectCredentialsAndInvoke(any(DescribeInstancesRequest.class),
any());

    final DeleteHandler handler = new DeleteHandler();

    final ResourceModel model = ResourceModel.builder().instanceId("i-1234").build();

    final ResourceHandlerRequest<ResourceModel> request =
ResourceHandlerRequest.<ResourceModel>builder()
        .desiredResourceState(model)
        .build();

    final CallbackContext context = CallbackContext.builder()
                                                                                   .stabilizationRetriesRemaining(60)
                                                                                   .instance(new
Instance().withState(inProgressState).withSecurityGroups(group))
.instanceSecurityGroups(Arrays.asList("sg-1234"))
                                                                                   .build();

    final ProgressEvent<ResourceModel, CallbackContext> response
        = handler.handleRequest(proxy, request, context, logger);

    final CallbackContext desiredOutputContext = CallbackContext.builder()
.stabilizationRetriesRemaining(59)

```

```
.instanceSecurityGroups(context.getInstanceSecurityGroups())
Instance().withState(InProgressState).withSecurityGroups(group))
    .instance(new
    .build());

    assertThat(response).isNotNull();
    assertThat(response.getStatus()).isEqualTo(OperationStatus.IN_PROGRESS);

assertThat(response.getCallbackContext()).isEqualToComparingFieldByField(desiredOutputContext);
    assertThat(response.getCallbackDelaySeconds()).isEqualTo(0);

assertThat(response.getResourceModel()).isEqualTo(request.getDesiredResourceState());
    assertThat(response.getResourceModels()).isNull();
    assertThat(response.getMessage()).isNull();
    assertThat(response.getErrorCode()).isNull();
}

@Test
public void testStabilizationTimeout() {
    final DeleteHandler handler = new DeleteHandler();

    final ResourceModel model = ResourceModel.builder().instanceId("i-1234").build();

    final ResourceHandlerRequest<ResourceModel> request =
ResourceHandlerRequest.<ResourceModel>builder()
    .desiredResourceState(model)
    .build();

    final CallbackContext context = CallbackContext.builder()
    .stabilizationRetriesRemaining(0)

    .instanceSecurityGroups(Arrays.asList("sg-1234"))
    .instance(new
Instance().withState(new InstanceState().withName("terminated")))
    .build());

    try {
        handler.handleRequest(proxy, request, context, logger);
    } catch (RuntimeException e) {
        assertThat(e.getMessage()).isEqualTo(EXPECTED_TIMEOUT_MESSAGE);
    }
}
}
```

Test the resource type

Next, we'll use the AWS SAM CLI to test locally that our resource will work as expected once we submit it to the CloudFormation registry. To do this, we'll need to define tests for the SAM to run against our create and delete handlers.

Create the SAM test files

1. Create two files:

- `package-root/sam-tests/create.json`
- `package-root/sam-tests/delete.json`

Where `package-root` is the root of the resource project. For our walkthrough example, the files would be:

- `example-testing-wordpress/sam-tests/create.json`

- `example-testing-wordpress/sam-tests/delete.json`

2. In `example-testing-wordpress/sam-tests/create.json`, paste the following test.

Note

Add the necessary information, such as credentials and log group name, and remove any comments in the file before testing.

To generate temporary credentials, you can use `aws sts get-session-token`.

```
{
  "credentials": {
    # Real STS credentials need to go here.
    "accessKeyId": "",
    "secretAccessKey": "",
    "sessionToken": ""
  },
  "action": "CREATE",
  "request": {
    "clientRequestToken": "4b90a7e4-b790-456b-a937-0cfdfa211dfe", # Can be any UUID.
    "desiredResourceState": {
      "Name": "MyBlog",
      "SubnetId": "subnet-0bc6136e" # This should be a real subnet that exists in
the account you're testing against.
    },
    "logicalResourceIdentifier": "MyResource"
  },
  "callbackContext": null
}
```

3. In `example-testing-wordpress/sam-tests/delete.json`, paste the following test.

Note

Add the necessary information, such as credentials and log group name, and remove any comments in the file before testing.

To generate temporary credentials, you can use `aws sts get-session-token`.

```
{
  "credentials": {
    # Real STS credentials need to go here.
    "accessKeyId": "",
    "secretAccessKey": "",
    "sessionToken": ""
  },
  "action": "DELETE",
  "request": {
    "clientRequestToken": "4b90a7e4-b790-456b-a937-0cfdfa211dfe", # Can be any UUID.
    "desiredResourceState": {
      "Name": "MyBlog",
      "InstanceId": "i-0167b19dd4c1efbf3", # This should be the instance ID that
was created in the "create.json" test.
      "SubnetId": "subnet-0bc6136e" # This should be a real subnet that exists in
the account you're testing against.
    },
    "logicalResourceIdentifier": "MyResource"
  },
  "callbackContext": null
}
```

Test the Create Handler

Once you've created the `example-testing-wordpress/sam-tests/create.json` test file, you can use it to test your create handler.

Ensure Docker is running on your computer.

1. Invoke the SAM function from the resource package root directory using the following commands.

```
sam local invoke TestEntrypoint --event sam-tests/create.json
```

Note

Occasionally these tests will fail with a retry-able error. In such a case, run the tests again to determine whether the issue was transient.

Because the create handler was written as a state machine, invoking the tests will return an output that represents a state. For example:

```
{
  "callbackDelaySeconds": 0,
  "resourceModel": {
    "SubnetId": "subnet-0bc6136e",
    "Name": "MyBlog"
  },
  "callbackContext": {
    "instance": {
      "subnetId": "subnet-0bc6136e",
      "virtualizationType": "hvm",
      "capacityReservationSpecification": {
        "capacityReservationPreference": "open"
      },
      "amiLaunchIndex": 0,
      "elasticInferenceAcceleratorAssociations": [],
      "sourceDestCheck": true,
      "stateReason": {
        "code": "pending",
        "message": "pending"
      },
      "instanceId": "i-0b6978477c0e9d358",
      "vpcId": "vpc-eb80788e",
      "hypervisor": "xen",
      "rootDeviceName": "/dev/sda1",
      "productCodes": [],
      "state": {
        "code": 0,
        "name": "pending"
      },
      "architecture": "x86_64",
      "ebsOptimized": false,
      "imageId": "ami-04fb0368671b6f138",
      "blockDeviceMappings": [],
      "stateTransitionReason": "",
      "clientToken": "207dc686-e95c-4df9-8fcb-ee22bbdde963",
      "instanceType": "m4.large",
      "cpuOptions": {
        "threadsPerCore": 2,
        "coreCount": 1
      },
      "monitoring": {
        "state": "disabled"
      },
      "publicDnsName": "",
      "privateIpAddress": "172.0.0.133",
      "rootDeviceType": "ebs",
      "tags": [
        {
          "value": "MyBlog",
          "key": "Name"
        }
      ]
    }
  }
}
```

```
    }
  ],
  "launchTime": 1567718644000,
  "elasticGpuAssociations": [],
  "licenses": [],
  "networkInterfaces": [
    {
      "networkInterfaceId": "eni-0e450b35a159b60fe",
      "privateIpAddresses": [
        {
          "privateIpAddress": "172.0.0.133",
          "primary": true
        }
      ],
      "subnetId": "subnet-0bc6136e",
      "description": "",
      "groups": [
        {
          "groupName": "MyBlog-cbb70fca-4704-430b-b67b-7d6d550e0592",
          "groupId": "sg-063679dc7681610c3"
        }
      ],
      "ipv6Addresses": [],
      "ownerId": "671472782477",
      "sourceDestCheck": true,
      "privateIpAddress": "172.0.0.133",
      "interfaceType": "interface",
      "macAddress": "02:e1:4b:d1:f7:40",
      "attachment": {
        "attachmentId": "eni-attach-0a01c63e4b45c4a5d",
        "deleteOnTermination": true,
        "deviceIndex": 0,
        "attachTime": 1567718644000,
        "status": "attaching"
      },
      "vpcId": "vpc-eb80788e",
      "status": "in-use"
    }
  ],
  "privateDnsName": "ip-172-0-0-133.us-west-2.compute.internal",
  "securityGroups": [
    {
      "groupName": "MyBlog-cbb70fca-4704-430b-b67b-7d6d550e0592",
      "groupId": "sg-063679dc7681610c3"
    }
  ],
  "placement": {
    "groupName": "",
    "tenancy": "default",
    "availabilityZone": "us-west-2b"
  }
},
"stabilizationRetriesRemaining": 60
},
"status": "IN_PROGRESS"
}
```

2. From the test response, copy the contents of the callbackContext, and paste it into the callbackContext section of the example-testing-wordpress/sam-tests/create.json file.
3. Invoke the TestEntrypoint function again.

```
sam local invoke TestEntrypoint --event sam-tests/create.json
```

If the resource has yet to complete provisioning, the test returns a response with a status of `IN_PROGRESS`. Once the resource has completed provisioning, the test returns a response with a status of `SUCCESS`. For example:

```
{
  "callbackDelaySeconds": 0,
  "resourceModel": {
    "InstanceId": "i-0b6978477c0e9d358",
    "PublicIp": "34.211.69.121",
    "SubnetId": "subnet-0bc6136e",
    "Name": "MyBlog"
  },
  "status": "SUCCESS"
}
```

4. Repeat the previous two steps until the resource has completed.

When the resource completes provisioning, the test response contains both its `PublicIp` and `InstanceId`:

- You can use the `PublicIp` value to navigate to the WordPress site.
- You can use the `InstanceId` value to test the delete handler, as described below.

Test the Delete Handler

Once you've created the `example-testing-wordpress/sam-tests/delete.json` test file, you can use it to test your delete handler.

Ensure Docker is running on your computer.

1. Invoke the `TestEntrypoint` function from the resource package root directory using the following commands.

```
sam local invoke TestEntrypoint --event sam-tests/delete.json
```

Note

Occasionally these tests will fail with a retry-able error. In such a case, run the tests again to determine whether the issue was transient.

As with the create handler, the delete handler was written as a state machine, so invoking the test will return an output that represents a state.

2. From the test response, copy the contents of the `callbackContext`, and paste it into the `callbackContext` section of the `example-testing-wordpress/sam-tests/delete.json` file.
3. Invoke the `TestEntrypoint` function again.

```
sam local invoke TestEntrypoint --event sam-tests/delete.json
```

If the resource has yet to complete provisioning, the test returns a response with a status of `IN_PROGRESS`. Once the resource has completed provisioning, the test returns a response with a status of `SUCCESS`.

4. Repeat the previous two steps until the resource has completed.

Performing resource contract tests

Resource contract tests verify that the resource type schema you've defined properly catches property values that will fail when passed to the underlying APIs called from within your resource handlers. This provides a way of validating user input before passing it to the resource handlers. For example, in the `Example::Testing::WordPress` resource type provide schema (in the `example-testing-wordpress.json` file), we specified regex patterns for the `Name` and `SubnetId` properties, and set the maximum length of `Name` as 219 characters. Contract tests are intended to stress and validate those input definitions.

Specify Resource Contract Test Override Values

The CloudFormation CLI performs resource contract tests using input that is generated from the patterns you define in your resource's property definitions. However, some inputs can't be randomly generated. For example, the `Example::Testing::WordPress` resource requires an actual subnet ID for testing, not just a string that matches the appearance of a subnet ID. So in order to test this property, we need to include a file with actual values for the resource contract tests to use. an `overrides.json` at the root of our project that looks like this:

1. Navigate to the root of your project.
2. Create a file named `overrides.json`.
3. Include the following override, specifying an actual subnet ID to use when performing resource contract tests.

```
{
  "CREATE": {
    "/SubnetId": "subnet-0bc6136e" # This should be a real subnet that exists in the
    account you're testing against.
  }
}
```

Run the Resource Contract Tests

To run resource contract tests, you'll need two shell sessions.

1. In a new session, run the following command:

```
$ sam local start-lambda
```

2. From the resource package root directory, in a session that is aware of the CloudFormation CLI, run the test command:

```
$ cfn test
```

The session that is running `sam local start-lambda` will display information about the status of your tests.

Submit the resource type

Once you have completed implementing and testing your resource provided, the final step is to submit it to the CloudFormation registry. This makes it available for use in stack operations in the account and region in which it was submitted.

- In a terminal, run the `submit` command to register the resource type in the `us-west-2` region.


```
$ cfn submit -v --region us-west-2
```

The CloudFormation CLI validates the included resource type schema, packages your resource provide project and uploads it to the CloudFormation registry, and then returns a registration token.

```
Validating your resource specification...
Packaging Java project
Creating managed upload infrastructure stack
Managed upload infrastructure stack was successfully created
Registration in progress with token: 3c27b9e6-dca4-4892-ba4e-3c0example
```

Resource type registration is an asynchronous operation. You can use the supplied registration token to track the progress of your type registration request using the [DescribeTypeRegistration](#) action of the CloudFormation API.

Note

If you update your resource type, you can submit a new version of that resource type. Every time you submit your resource type, CloudFormation generates a new version of that resource type. To set the default version of a resource type, use [SetTypeDefaultVersion](#). For example:

```
aws cloudformation set-type-default-version --type "RESOURCE" --type-name
"Example::Testing::WordPress" --version-id "00000002"
```

To retrieve information about the versions of a resource type, use [ListTypeVersions](#). For example:

```
aws cloudformation list-type-versions --type "RESOURCE" --type-name
"Example::Testing::WordPress"
```

Provision the resource in a CloudFormation stack

Once the registration request for your resource type has completed successfully, you can create a stack including resources of that type.

Note

Use [DescribeTypeRegistration](#) to determine when your resource type is successfully registration registered with a status of COMPLETE. You should also see your new resource type listed in the [CloudFormation console](#).

1. Save the following JSON as a stack template, with the name `stack.json`.

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "WordPress stack",
  "Resources": {
    "MyWordPressSite": {
      "Type": "Example::Testing::WordPress",
      "Properties": {
        "SubnetId": "subnet-0bc6136e", ## Note that this should be replaced with
a subnet that exists in your account.
        "Name": "MyWebsite"
      }
    }
  }
}
```

2. Use the template to create a stack.

Note

This resource uses an official WordPress image on AWS Marketplace. In order to create the stack, you'll first need to visit the [AWS Marketplace](#) and accept the terms and subscribe.

Navigate to the folder in which you saved the `stack.json` file, and create a stack named `wordpress`.

```
aws cloudformation create-stack --region us-west-2 \  
--template-body "file://stack.json" \  
--stack-name "wordpress"
```

As CloudFormation creates the stack, it should invoke your resource type create handler to provision a resource of type `Example::Testing::WordPress` as part of the `wordpress` stack.

As a final test of the resource type delete handler, you can delete the `wordpress` stack.

```
aws cloudformation delete-stack --region us-west-2 \  
--stack-name wordpress
```

Resource type FAQ

Below are some Frequently Asked Questions about resource type development.

Updates

- **Q: My service API implements Update actions as an Upsert, can I implement my CloudFormation resource type in this way?**

A: No, CloudFormation requires that update actions to a non-existing resource always throw a `ResourceNotFoundException`.

Schema development

- **Q: How can I re-use existing schemas, or establish relationships to other resource types in my schema?**

A: Relationships and re-use are established using JSON Pointers. These are implemented using the `$ref` keyword in your resource type schema. Refer to [Modeling resource types for use in AWS CloudFormation \(p. 5\)](#) for more information.

Permissions and authorization

- **Q: Why am I getting an `AccessDeniedException` for my AWS API calls?**

A: If you are seeing errors in your logs related to `AccessDeniedException` for a Lambda Execution Role like

```
A downstream service error occurred in a CREATE action on a  
AWS::MyService::MyResource: com.amazonaws.services.logs.model.AWSLogsException: User:  
arn:aws:sts::123456789012:assumed-role/  
UluruResourceHandlerLambdaExecutionRole-123456789012pdx/AWS-MYService-MyResource-  
Handler-1h344teffe is not authorized to perform: some:ApiCall on resource: some-resource
```

```
(Service: AWSLogs; Status Code: 400; Error Code: AccessDeniedException; Request ID: 36af0cec-a96a-11e9-b204-ddabexample)
```

This is an indication that you are attempting to create and invoke AWS APIs using the default client, which is injected with environment credentials.

For resource types, you should use the passed-in `AmazonWebServicesClientProxy` object to make AWS API calls, as in the following example.

```
SesClient client = ClientBuilder.getClient();
final CreateConfigurationSetRequest createConfigurationSetRequest =
    CreateConfigurationSetRequest.builder()
        .configurationSet(ConfigurationSet.builder()
            .name(model.getName())
            .build())
        .build();
proxy.injectCredentialsAndInvokeV2(createConfigurationSetRequest,
    client::createConfigurationSet);
```

- **Q: How do I specify credentials for non-AWS API calls?**

A: For non-AWS API calls which require authentication and authorization, you should create properties in your resource type which contain the credentials. Define these properties in the resource type schema as `writeOnlyProperties`.

Users can then provide their own credentials through their CloudFormation templates. We encourage the use of [dynamic references](#) in CloudFormation templates, which can use AWS Secrets Manager to fetch credentials at runtime.

Resource type development

- **Q: Can I share functionality between resource by adding common functionality to the `BaseHandler`?**

A: Because the `BaseHandler` is code-generated, it cannot be edited.

- **Q: For Java development, is there a way to include multiple resources in a single maven project?**

A: Not currently. For security and manageability, the CloudFormation Registry registers each resource type as a separate, versioned, type. You could still share code through a shared package. Ideally, the wrapper layer does most of the boilerplate. If you see a need for more boilerplate, we would like to know how we can improve for that use case rather than combine types in a package, so please reach out to the team.

- **Q: Will `software.amazon.cloudformation.proxy.Logger` have `debug/info/warning/error` levels/?**

A: Currently, all log messages are emitted to AWS CloudWatch, which has no built-in concept of log levels.

Testing

- **Q: For testing, when should I use `sam local invoke`, `cfn test` and `mvn test`?**

A: Use the various test capabilities for the test scenarios described below.

- `sam local invoke`: Creating custom integration test by passing in custom CloudFormation payloads and isolate specific handlers.

- `cfm test`: Contract tests meant to cycle through CRUDL actions and ideally leave in a clean state (if tests pass).
- `mvn test`: Used for Unit testing. The goal is to confirm that each method/unit of the resource performs as intended. It also helps to ensure that you have enough code coverage. We expect unit tests to mock dependencies and not create real resources.

Deployment

- **Q: Is the resource type interface guaranteed to be stable?**

A: The communication protocol between CloudFormation and your resource type package is subject to change. However this will be done in a backwards-compatible way, using versioned interfaces. This will be invisible to you as a developer and is managed as part of the CloudFormation managed platform.

The interface that your handlers implement inside your package is expected to be stable. We may introduce improvements, such as security fixes or other changes to the package, but these will be done with versioned dependency or CloudFormation CLI updates. You are not required to upgrade your packages in order to publish them, only to incorporate these improvements.

Developing modules

Modules are a way for you to package resource configurations for inclusion across stack templates, in a transparent, manageable, and repeatable way. Use modules to encapsulate common service configurations and best practices as modular, customizable building blocks that users can then take and include in their stack templates. Modules enable you to capture and disseminate resource configurations that incorporate best practices, expert domain knowledge, and accepted guidelines (for areas such as security, compliance, governance, and industry regulations). Users can then include the module in their template without having to acquire deep knowledge of the intricacies of the resource implementation.

For example, a domain expert in networking could create a module that contains built-in security groups and ingress/egress rules that adhere to security guidelines. A user could then include that module in their template to provision secure networking infrastructure in their stack--without having to spend time figuring out how VPCs, subnets, security groups, and gateways work. And because modules are versioned, if security guidelines change over time, the module author can create a new version of the module that incorporates those changes.

A module can contain:

- Template sections, including resources to be provisioned from the module, along with any associated data, such as outputs or conditions. Modules can also contain other modules.
- Any *module parameters*, which enable you to specify custom values whenever the module is used.

Characteristics of modules include:

- *Predictability*: Because a module must adhere to its schema, the resources and other outputs provisioned from the module are predictable.
- *Reusability*: Develop a module once, then reuse it across multiple templates and accounts
- *Traceability*: CloudFormation retains knowledge of which resources in a stack were provisioned from a module, enabling users to trace the source of resource changes.
- *Manageability*: Once you've registered a module, you can manage it through the CloudFormation registry, including versioning and account and region availability.

Users are able to register modules as private types in the CloudFormation registry for use in their accounts.

To use a module, users include it in their template as they would an individual resource, including specifying any necessary parameters for the module. When users initiate a stack operation, CloudFormation generates a processed template that resolves any included modules into the appropriate resources.

Users can use [change sets](#) to preview the resources to be added or updated prior to actually executing the stack operation.

For more information on using a module in a template, see [Using modules to encapsulate and reuse resource configurations](#) in the *CloudFormation User Guide*.

Module structure

A module consists of two main peices:

- A *template fragment*, which defines the resources and associated information you want to provision through use of the module, including any module parameters you define.
- A *module schema* that you generate based on the template fragment. The module schema declares the contract you defined in the template fragment, and is viewable to users in the CloudFormation registry.

Creating the module template fragment

The starting point for developing a module is the template fragment. The template fragment is a file that contains the information that defines the resources for CloudFormation to provision during stack operations, including:

- A **Resources** section that defines the resources to be provisioned.

The **Resources** section is required.

- Additional other template sections related for the provisioning of the resources as necessary, such as **Outputs** and **Conditions**.
- A **Parameters** section for any optional module-level parameters you want to define.

Much like **template parameters**, module parameters enable the user to input custom values to a module from the template (or module) that contains it. The module can then use these values to set properties of the resources it contains.

Currently, CloudFormation supports template fragments written in JSON.

For example, the following template fragment creates an S3 bucket resource, and sets the **AccessControl** property to **Private** and the resource **DeletionPolicy** to **Retain**. In addition, the template fragment defines a module-level parameter, **VersioningConfigurationParam**, whose values is used to set the **VersioningConfiguration** status of the S3 bucket.

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "A sample S3 Bucket module (AWS::SampleS3::Bucket::MODULE)",
  "Parameters": {
    "VersioningConfigurationParam": {
      "Description": "Versioning configuration",
      "Type": "String",
      "AllowedValues": [
        "Enabled",
        "Suspended"
      ]
    }
  },
  "Resources": {
    "S3BucketName": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "AccessControl": "Private",
        "VersioningConfiguration": {
          "Status": {
            "Ref": "VersioningConfigurationParam"
          }
        }
      }
    },
    "DeletionPolicy": "Retain"
  }
}
```

You can author template fragments manually, or use any tool that generates CloudFormation templates. For example, you can use the AWS Cloud Development Kit (AWS CDK) to synthesize one or more CDK [constructs](#) to produce a CloudFormation template, and then use that template as the basis for a module. For more information on the CDK, see the [AWS Cloud Development Kit](#).

Note

Be aware that regardless of the method you use to create a module's template fragment, it must adhere to the restrictions on what can be included in a template fragment for a module.

Considerations when authoring the template fragment

Keep in mind the following considerations when developing modules:

- Modules are, by design, predictable and transparent. Because of this, you cannot include features which can potentially result in external information or resources being imported into the module. These features include:
 - [Importing](#) stack values, using `Fn::ImportValue` intrinsic function.
 - [Exporting](#) stack values, using the `Export` field in the [Outputs](#) template section. (Use of the `Outputs` section is supported.)
 - [Macros](#), including use of the [Transform](#) template section or the `Fn::Transform` function.

This includes transforms provided by CloudFormation, such as [AWS::Include](#) and [AWS::Serverless](#).

- [Nested stacks](#), which are represented in the template by the `AWS::CloudFormation::Stack` resource.
- [Stack sets](#), which are represented in the template by the `AWS::CloudFormation::StackSet` resource.
- Tags cannot be specified at the module level. However:
 - You can assign tags to individual resources within the module, as you would assign tags to any resource.
 - You can use module parameters to set tag values.

Create the module parameter, and then have the tag you've assigned to individual resources within the module reference that module parameter. For more information, see [Using parameters to specify module values](#) in the *CloudFormation User Guide*.

- Tags you specify at the *stack* level are assigned to the individual resources derived from the module.
- Helper scripts specified at the module level do not propagate to the individual resources contained in the module when CloudFormation processes the template.
- Outputs specified in the module are propagated to outputs at the template level.

Each output will be assigned a logical ID that is a concatenation of the module logical name and the output name as defined in the module. For more information on outputs, see [Outputs](#) in the *CloudFormation User Guide*.

- Parameters specified in the module are not propagated to parameters at the template level.

However, you can create template-level parameters that reference module-level parameters. For more information, see [Using parameters to specify module values](#) in the *CloudFormation User Guide*.

Nesting modules

Modules can contain other modules. You can nest modules up to three levels deep. To include a module in your module, reference it in the `Resources` section of your template fragment, as you would any other resource. For an example, see [Specifying properties on resources in a child module from the parent module](#) in the *CloudFormation User Guide*.

Macros and modules

CloudFormation does not support inclusion of modules in macros. A module cannot contain a macro.

For more information on macros, see [Using macros to perform custom processing](#) in the *CloudFormation User Guide*.

Defining parameters in a module

Much like template parameters, module parameters enable the user to input custom values to a module from the template (or module) that contains it. The module can then use these values to set properties of the resources it contains.

You define a module parameter as you would a template parameter. For detailed information about parameter requirements and definition, see [Parameters](#) in the *CloudFormation User Guide*.

[Dynamic references](#) are not resolved when the module is processed by CloudFormation, but when the individual resources are created or updated during stack operations.

Module parameters do not count toward the parameter maximum for template parameters. For information on template parameters and their limits, see [Parameters](#) in the *CloudFormation User Guide*.

Parameters specified in the module are not propagated to parameters at the template level. However, you can create template-level parameters that reference module-level parameters.

For information on how users can specify parameter values in modules, see [Using parameters to specify module values](#) in the *CloudFormation User Guide*.

Specifying constraints for module parameters

Module parameters do not support [Type](#) or [Constraint](#) enforcement. To perform type or constraint checking on a module parameter, create a template parameter with the desired constraints, then reference that template parameter in your module parameter.

Specifying policies on resources contained in a module

If you specify the following resource policy attributes at the module level, CloudFormation applies the policy attribute to *all* resources contained in the module:

- `DeletionPolicy`
- `UpdateReplacePolicy`

This does not include specifying the `Snapshot` option for `UpdateReplacePolicy`. Specify this option on the resource directly.

Policy attributes specified at a resource level override any specified at the module level.

You cannot specify the following resource policy attributes at the module level:

- `CreationPolicy`
- `UpdatePolicy`

If you use a `DependsOn` attribute to specify that a resource in your template depends on a module, CloudFormation will finish provisioning *all* resources in the module before provisioning the dependant resource.

For more information on resource policies, see [Resource attribute reference](#) in the *CloudFormation User Guide*.

Generating the module schema

The module schema is generated from the template fragment, and defines the contract to which the module adheres, including defining the input it accepts and the possible resources it resolves to when included in a template.

To generate the module schema, use the `validate` command once you've authored your template fragment.

For example, suppose you created a module package and used the template fragment above. The `validate` command would result in the following module schema:

```
{
  "typeName": "AWS::SampleS3::Bucket::MODULE",
  "properties": {
    "VersioningConfigurationParam": {
      "description": "Version Configuration",
      "type": "string"
    }
  },
  "resources": {
    "type": "object",
    "properties": {
      "S3Bucket": {
        "$ref": "aws-s3-bucket.json"
      }
    }
  },
  "additionalProperties": false
}
```

For more information, see [Develop a module](#).

Develop a module using the CFN-CLI

Follow these basic steps to develop and register a module project.

1. In the CFN-CLI, use the `init` command to create a new project. The `init` command creates a `fragments` folder containing a sample fragment file named `sample.json`.

Follow the prompts. Specify that you want to create a **module(m)**, and enter the module name.

```
cfn init
Initializing new project
Do you want to develop a new resource(r) or a module(m)?.
>> m
What is the name of your module type?
(<Organization>::<Service>::<Name>::MODULE)
>> My::S5::SampleBucket::MODULE
```

2. Include your template fragment in the project.

In the `fragments` folder in the project, you should find a file named `sample.json`. This is the template fragment file. Author your template fragment in this file and save.

You can rename this file as necessary. The folder can only contain a single file.

For more information, see [Creating a module template fragment](#).

3. Use `validate` to validate your project. Fix any issues reported.

The `validate` command regenerates the module schema, based on the template fragment you included in the `fragments` folder. The module schema is located in the root folder, and named `schema.json`.

4. Use `submit` to register the module with CloudFormation, in the specified region. Registering a module makes it available for inclusion in CloudFormation templates.

Note

When you register your module using `submit`, CloudFormation re-generates your module schema based on the template fragment in your project. You cannot specify a schema file directly. To specify a module schema file when registering a module, use `RegisterType` in the CloudFormation API.

For information on using modules in CloudFormation templates, see [Using modules](#) in the *CloudFormation Users Guide*.

CloudFormation CLI command reference

The following commands are available through the CloudFormation Command Line Interface (CLI).

Topics

- [Global parameters \(p. 71\)](#)
- [init \(p. 71\)](#)
- [generate \(p. 72\)](#)
- [validate \(p. 72\)](#)
- [invoke \(p. 72\)](#)
- [test \(p. 73\)](#)
- [submit \(p. 74\)](#)

Global parameters

The following parameters can be used with any CloudFormation CLI command.

`-h, --help`

Show the help message and exit.

`-v, --verbose`

Increase the output verbosity. Can be specified multiple times.

init

Description

Generates a new extension project with stub source files.

While the specific folder structure and files generated varies by language, in general the project includes:

- Resource schema file
- Handler function source files
- Unit test files
- IDE and build files for the specified language

By default, `init` generates the extension project in the current directory.

Synopsis

```
cfm init  
[--force]
```

Options

`--force`

Force project files to be overwritten.

Output

The `init` command launches a wizard that walks you through setting up the project, including specifying the extension name.

generate

Description

Generates code based on the project and resource type schema.

Synopsis

```
cfn generate
```

Output

```
Generated files for <type_name>
```

validate

Description

Validates a project's resource specification against the [Resource Type Definition Schema](#).

Synopsis

```
cfn validate
```

Output

invoke

Description

Performs contract tests on the specified handler of a resource type.

Synopsis

```
cfn invoke
[--endpoint <value>]
[--function-name <value>]
[--region <value>]
[--max-reinvoke <value>]
action
request
```

Options

`--endpoint <value>`

The endpoint at which the type can be invoked. Alternately, you can also specify an actual Lambda endpoint and function name in your AWS account.

Default: `http://127.0.0.1:3001`

`--function-name <value>`

The logical lambda function name in the SAM template. Alternately, you can also specify an actual Lambda endpoint and function name in your AWS account.

Default: `TestEntrypoint`

`--region <value>`

The region to configure the client to interact with.

Default: `us-east-1`

`--max-reinvoke <value>`

Maximum number of IN_PROGRESS re-invoations allowed before exiting. If not specified, will continue to re- invoke until terminal status is reached.

`action`

Which single handler to invoke.

Values: `CREATE | READ | UPDATE | DELETE | LIST`

`request`

File path to a JSON file containing the request with which to invoke the function.

Output

test

Description

Performs contract tests on the handlers of a resource type.

Synopsis

```
cfn test
[--endpoint <value>]
[--function-name <value>]
[--region <value>]
[--role-arn <value>]
```

Options

`--endpoint <value>`

The endpoint at which the type can be invoked. Alternately, you can also specify an actual Lambda endpoint and function name in your AWS account.

Default: `http://127.0.0.1:3001`

`--function-name <value>`

The logical lambda function name in the SAM template. Alternately, you can also specify an actual Lambda endpoint and function name in your AWS account.

Default: `TestEntrypoint`

`--region <value>`

The region to use for temporary credentials.

Default: `us-east-1`

`--role-arn <value>`

The Amazon Resource Name (ARN) of the IAM execution role for the contract tests to assume and use when performing operations.

If you do not specify an execution role, the contract tests use the environment credentials or the credentials specified in the [Boto 3 credentials chain](#).

Output

submit

Description

Registers the extension with CloudFormation, in the specified region. Registering an extension makes it available for use in CloudFormation operations.

Registering includes:

- Validating the resource schema.
- Packaging up the resource project files and uploading them to CloudFormation.

This includes the source code for your resource handlers. These resource handlers run within the CloudFormation account.

- Determining which handlers have been specified for the resource, and running the appropriate contract tests.
- Uploading the resource handlers as functions that CloudFormation calls at the appropriate times in a resource's lifecycle.
- Returning a *registration token* that you can use with the [DescribeTypeRegistration](#) action to track the status of the registration request.

Note

The user registering the extension must be able to access the schema handler package in the S3 bucket. That is, the user needs to have [GetObject](#) permissions for the schema handler package. For more information, see [Actions, Resources, and Condition Keys for Amazon S3](#) in the *AWS Identity and Access Management User Guide*.

Synopsis

```
cfn submit
[--dry-run]
[--endpoint-url <value>]
[--region <value>]
[--role-arn <value>]
[--no-role]
[--set-default]
```

Options

`--dry-run`

Validate the schema and package up the project files, but do not register the extension with CloudFormation.

`--endpoint-url <value>`

The CloudFormation endpoint to use.

`--region <value>`

The AWS region in which to register the extension. If no region is specified, the extension is registered in the default region.

`--role-arn <value>`

A specific IAM role to use when invoking handler operations.

If you do not specify an IAM role, the CloudFormation CLI attempts to create or update an execution role based on the execution role template derived from the resource type's schema, and then passes this execution role to CloudFormation. For more information, see [Accessing AWS APIs from a Resource Type](#).

You cannot specify both `--role-arn` and `--no-role` arguments.

`--no-role`

Prevent the CloudFormation CLI from passing an execution role to CloudFormation.

If your resource type calls AWS APIs in any of its handlers, you must either specify a role arn, or have the CloudFormation CLI create or update an execution role and pass that execution role to CloudFormation. For more information, see [Accessing AWS APIs from a Resource Type](#).

You cannot specify both `--role-arn` and `--no-role` arguments.

`--set-default`

Upon successful registration of the type version, sets the current type version as the default version.

Output

Extension registration is an asynchronous operation. You can use the supplied registration token to track the progress of your extension registration request using the [DescribeTypeRegistration](#) action of the CloudFormation API.

Document History for User Guide

The following table describes the documentation for this release of the CloudFormation Command Line Interface (CLI).

- **Latest documentation update:** November 14, 2019

update-history-change	update-history-description	update-history-date
General Availability (p. 77)	Initial publication of the CloudFormation CLI documentation.	November 14, 2019

AWS glossary

For the latest AWS terminology, see the [AWS glossary](#) in the *AWS General Reference*.