Creating Parameterised CloudFormation Stacks

In this lesson, you will learn about stages and how to configure them through parameters.

WE'LL COVER THE FOLLOWING

Stages

In the stack template, you configured events for the `/hello-world` path. But the actual path in the API is slightly different, and it includes the `/Prod/` prefix. This is because SAM automatically creates a `Prod` stage for the implicit API.

In API Gateway terminology, a *stage* is a published version of the API configuration. Using stages allows API Gateway to serve a stable version of the API while users are adding or configuring resources to prepare for a new version. In the context of API Gateway, stages serve a very similar purpose to aliases of Lambda functions. You can even set up a 'canary' deployment where one version of the API gets just a percentage of traffic for a deployed stage.

An API needs at least one stage so that it becomes publicly accessible. In theory, you can set up different stages for development, testing, or production with a single API, but with SAM that's not convenient. It's a lot more common to create completely different stacks for different pipeline steps. The APIs you create using SAM will usually have a single stage.

`Prod`

The standard stage name, `Prod`, forces users to remember a mix of upper-case and lower-case letters. Mixed-case URLs are a lot more error-prone than if everything is only lower case, so they are not advised. SAM lets you configure the stage name for an API, but you can't set this through the `Globals` section. You'll have to create an API explicitly and configure it.

> ### CloudFormation API or SAM API?
>
> CloudFormation has two resources for managing Rest APIs, `AWS::ApiGateway::RestApi` and `AWS::ApiGatewayV2::Api`. SAM provides a convenient wrapper around lower-level resources with `AWS::Serverless::Api`. When you use the SAM wrapper, you don't need to set up security policies for Lambda event mappings and worry about connecting to the right Lambda version or alias. To achieve the same result with lower-level resources, you would need a lot more template code.
>
> You can use the reference to a SAM wrapper resource instead of lower-level resources in all the places where CloudFormation expects the identifier of a REST API. If SAM created an API implicitly, you can get its identifier using the automatically generated `ServerlessRestApi` reference.

Defining stack parameters

Having `/Prod/` in the URL might not feel right to you, but that's a matter of taste. Whatever you choose as a nice URL, other people might not like. You can make a template that works for everybody by introducing a *stack parameter*. Parameters are a way to create customisable stacks. Users can set the parameter value to provide their own configuration when deploying a stack. This makes it easy to publish reusable stacks and even connect the outputs of one stack to the inputs of another.

Let’s first create a parameter to hold the stage name. Define a top-level section called `Parameters` in your template. For example, add it immediately after the `Globals` section. Set up a parameter by providing its name as a key then list the expected type and default value as in the following listing:

```
Parameters:
  AppStage:
    Type: String
    Default: api
```

Line 10 to Line 13 of ch6/template-custom-stage.yaml

The default value setting is optional and just makes it slightly easier to deploy the stack without having to set all parameters. If you don’t configure a default value for a parameter and don’t provide a value during deployment, CloudFormation will refuse to set up the stack. For cases where you actually want the default to be undefined, use a blank string between quotes as the value (`Default: ''`).

Referencing parameter values

You can now create an API Gateway in the `Resources` section of the template. Add a new resource of type `AWS::Serverless::Api` (on the same indentation level as the `HelloWorldFunction`) and set up the `StageName` property. In the previous chapter, you used the `!Ref` function and turned the logical name of a CloudFormation resource into its physical name. The same function works on parameters. To use a parameter value directly in a YAML template, just put the parameter name after `!Ref`, as in the following listing:

```
WebApi:
  Type: AWS::Serverless::Api
  Properties:
    StageName: !Ref AppStage
```

Line 20 to Line 23 of code/ch6/template-custom-stage.yaml

The `AWS::Serverless::Api` resource allows you to quickly set up caching, canary deployments, authorisation, and a lot more. For a full list of properties you can configure using this resource, check out the AWS::Serverless::Api resource page in the SAM documentation.

resource page in the SAM documentation.

The logical identifiers of resources (such as `WebApi` and the `HelloWorldFunction`) are also references. This means that you can use `!Ref WebApi` in any place where CloudFormation expects an ID of an API Gateway. SAM events with the `Api` type have a third optional property, `RestApiId`, where you can specify a non-default API. Add this into both events of the template (lines 14 and 20 in the next code listing).

```
HelloWorldFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: hello-world/
    Handler: app.lambdaHandler
    Runtime: nodejs12.x
    AutoPublishAlias: live
    Events:
      HelloWorld:
        Type: Api
        Properties:
          Path: /hello
          Method: get
          RestApiId: !Ref WebApi
      SubmitForm:
        Type: Api
        Properties:
          Path: /hello
          Method: post
          RestApiId: !Ref WebApi
```

Line 24 to Line 43 of code/ch6/template-custom-stage.yaml

Finally, you need to somehow discover the ID of the newly created API so that you can use it to send requests. Let's change the value of the `HelloWorldApi` output in the template to use `WebApi` instead of the built-in `ServerlessRestApi`. While we're doing that, let's also change the hard-coded `Prod` stage to the `AppStage` parameter.

```
Outputs:
  HelloWorldApi:
    Description: "API Gateway endpoint URL"
    Value: !Sub "https://${WebApi}.execute-api.${AWS::Region}.amazonaws.com/${AppStage}/hello
```

Line 44 to Line 47 of code/ch6/template-custom-stage.yaml

!Ref or !Sub

> Notice how you used references on line 47 of the previous listing. Instead of `!Ref`, you use `!Sub`. Because of the exclamation mark, this is again a function call. The `Sub` function substitutes references enclosed within `${}` with related values. Use `!Ref` when you need the actual reference, and `!Sub` when you want to combine references with some other text, for example to construct a URL.

Now you'll build, package, and deploy your stack, then list stack outputs or find your stack in the AWS Web Console. You'll see that the API endpoint has a different URL:

```
$ aws cloudformation describe-stacks --stack-name sam-test-1 --query Stack
s[].Outputs
[
  [
    {
      "OutputKey": "HelloWorldApi",
      "OutputValue": "https://keqbc29e60.execute-api.us-east-1.amazonaws.c
om/api/hello/",
      "Description": "API Gateway endpoint URL"
    }
  ]
]
```

Environment Variables

Key:	Value:
AWS_ACCESS_KEY_ID	Not Specified...
AWS_SECRET_ACCE...	Not Specified...
BUCKET_NAME	Not Specified...
AWS_REGION	Not Specified...

```
{
  "body": "{\"message\": \"hello world\"}",
  "resource": "/{proxy+}",
  "path": "/path/to/resource",
  "httpMethod": "POST",
  "isBase64Encoded": false,
  "queryStringParameters": {
    "foo": "bar"
  },
  "pathParameters": {
    "proxy": "/path/to/resource"
  },
  "stageVariables": {
    "baz": "qux"
  },
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8",
    "Accept-Encoding": "gzip, deflate, sdch",
    "Accept-Language": "en-US,en;q=0.8",
    "Cache-Control": "max-age=0",
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Desktop-Viewer": "true",
    "CloudFront-Is-Mobile-Viewer": "false",
    "CloudFront-Is-SmartTV-Viewer": "false",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Viewer-Country": "US",
    "Host": "1234567890.execute-api.us-east-1.amazonaws.com",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Custom User Agent String",
    "Via": "1.1 08f323deadbeefa7af34d5feb414ce27.cloudfront.net (CloudFront)",
    "X-Amz-Cf-Id": "cDehVQoZnx43VYQb9j2-nvCh-9z396Uhbp027Y2JvkCPNLmGJHqlaA==",
    "X-Forwarded-For": "127.0.0.1, 127.0.0.2",
    "X-Forwarded-Port": "443",
    "X-Forwarded-Proto": "https"
  },
  "requestContext": {
    "accountId": "123456789012",
    "resourceId": "123456",
    "stage": "prod",
    "requestId": "c6af9ac6-7b61-11e6-9a41-93e8deadbeef",
    "requestTime": "09/Apr/2015:12:34:56 +0000",
    "requestTimeEpoch": 1428582896000,
    "identity": {
      "cognitoIdentityPoolId": null,
      "accountId": null,
      "cognitoIdentityId": null,
      "caller": null,
      "accessKey": null,
      "sourceIp": "127.0.0.1",
      "cognitoAuthenticationType": null,
      "cognitoAuthenticationProvider": null,
      "userArn": null,
      "userAgent": "Custom User Agent String",
      "user": null
    },
    "path": "/prod/path/to/resource",
    "resourcePath": "/{proxy+}",
    "httpMethod": "POST",
    "apiId": "1234567890",
    "protocol": "HTTP/1.1"
  }
}
```

Notice that the path prefix is now `api`, because that was the default value for the `AppStage` parameter in the template. The domain name is different as well, because SAM created a new API Gateway for us. API Gateway domain names are based on their ID, so when you stopped using the implicit API and created your own, the API ID changed.

Custom API domain names

Custom API domain names

You can set up an API to work with your own domain name instead of a generic AWS URL. For example, create an API that responds to `https://api.company.com`. That is relatively straightforward to do from the API Gateway Web Console: select the *Custom Domain Names* option in the left menu and follow the instructions on the screen. For detailed instructions, check out the Set up Custom Domain Name for an API page in the AWS API Gateway documentation.

Provide parameter values during deployment

If you don't like this choice for the stage name, you can modify it very easily. There is no need to build and package again. You can use the same template and just redeploy it with different parameter values. Add `--parameter-overrides` to the deployment command, and then put the parameter name and value separated by an equals sign. For example:

```
sam deploy --template-file output.yaml --stack-name sam-test-1 --capabilit
ies CAPABILITY_IAM --parameter-overrides AppStage=test
```

List the stack outputs after deployment, and you'll see that the stage is now called `test`:

```
$ aws cloudformation describe-stacks --stack-name=sam-test-1 --query Stack
s[].Outputs
[
  [
    {
      "OutputKey": "HelloWorldApi",
      "OutputValue": "https://kdlawk4vtf.execute-api.us-east-1.amazonaws.c
om/test/hello/",
      "Description": "API Gateway endpoint URL"
    }
  ]
]
```

If you want to provide multiple parameters, just separate them with spaces. For example:

```
sam deploy --parameter-overrides AppStage=api AppName=Demo
```

Making stack parameters more user-friendly

Although you can specify an API stage now, there's nothing really preventing people from entering invalid stage names and breaking the stack completely. The `Type` field can catch some silly errors such as trying to provide text for a numerical value, but you used a `String` here, and almost anything is a valid string. CloudFormation doesn't know how you intended to use the parameter, so it can't know if it should validate it. This means that detecting problems can take a long time, especially with complex stacks. CloudFormation will likely start creating resources until it hits an invalid one, and then explode. For complex templates, especially when you use the same parameter in multiple places, error messages might not even make a lot of sense to an average user.

Luckily, there are several ways to make template parameters more user-friendly. You can explain the purpose of a parameter using the `Description` field and even set up automatic validation. With strings, you can set up `MinLength` and `MaxLength` properties to control the allowed size, and provide a regular expression to validate the contents using `AllowedPattern`. For numbers, you can configure the applicable range using `MinValue` and `MaxValue`. If the users should only be able to choose from a predetermined set of options, you can use `AllowedValues` and provide a list. To help people understand potential validation errors more easily, explain the limits using `ConstraintDescription`.

Here is how you could potentially set up the stage values to only accept one to 10 letters:

```
Parameters:
  AppStage:
    Type: String
    Default: api
    Description: API Gateway stage, used as a prefix for the endpoint URLs
    AllowedPattern: ^[A-Za-z]+$
    MaxLength: 10
    MinLength: 1
    ConstraintDescription: "1-10 Latin letters"
```

Line 10 to Line 18 of ch6/template-custom-stage.yaml

Build, package, and deploy this stack now, but try providing a stage name with a number during deployment. CloudFormation should complain quickly and show you a nice message about how to fix the problem:

```
$ sam deploy --template-file output.yaml --stack-name sam-test-1 --capabil
ities CAPABILITY_IAM --parameter-overrides AppStage=12345

An error occurred (ValidationError) when calling the CreateChangeSet opera
tion: Parameter AppStage failed to satisfy constraint: 1-10 Latin letters
```

Check out the CloudFormation Parameters documentation page for more information on validation settings.

That’s it for this lesson. In the next lesson, you’ll have a look at some interesting experiments!