Further Sections of 'template.yaml'

In this lesson, you'll see some further sections from 'template.yaml' and learn what they represent.

WE'LL COVER THE FOLLOWING

Description

After the transformation comes the template description:

```
Description: >
  sam-app

  Sample SAM Template for sam-app
```

The `Description` section contains a free-form explanation of the template. This section is useful if you want to publish the template or give it to another team, but it isn't necessary. You can safely skip it when creating templates for small experiments.

Multi-line text in YAML

YAML files usually contain text values on the same line as the corresponding key. You can use the right angle bracket (`>`) or a pipe (`|`)

> to signal that what follows is multi-line text, indented one level. The
>
> difference is that `>` removes line breaks from the result, and `|` preserves line breaks.

Global application settings

After the description, the example SAM template contains global application settings. This is a SAM-specific extension to CloudFormation and enables you to reduce the overall template file size by listing common settings in a single place instead of repeating them for each Lambda function. The example template contains just a single function, so adding global values doesn't make a lot of difference, but this section is useful for more complex applications. The sample template usually contains a global `Timeout` setting, the number of seconds the function is allowed to run in Lambda:

```
Globals:
  Function:
    Timeout: 3
```

Resources

Next comes the `Resources` section, which lists the services or resources for CloudFormation to configure:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: hello-world/
      Handler: app.lambdaHandler
      Runtime: nodejs12.x
      Events:
        HelloWorld:
          Type: Api
          Properties:
            Path: /hello
            Method: get
```

This is where things become really interesting. With CloudFormation, each resource starts with its own block. The first line specifying a resource is the

logical ID of the resource. In the sample template, `HelloWorldFunction` is a

logical ID of a Lambda function. The logical ID is an internal name that CloudFormation uses to refer to a resource in a single template. It is effectively a variable name, and you can use it to refer to a resource when connecting it with other resources within the same template.

CloudFormation will create the actual ('physical') function name using a randomised string based on the logical ID. This makes it possible to create several instances of the same infrastructure in a single AWS account (for example, for development, testing and production), without causing any resource conflicts.

> ### Do not set resource names
>
> CloudFormation allows you to specify names for many resource types, including functions, using the `Name` property. Unless there is a very specific reason why you need this, avoid setting physical names. Letting CloudFormation set random names makes it easy to create several stacks based on the same template and avoid resource naming conflicts.

Each resource needs to specify a `Type` property which tells CloudFormation what to create. For the `HelloWorldFunction`, the type is `AWS::Serverless::Function`. This is another SAM extension to CloudFormation, making it easy to set up Lambda functions. You will almost always use this type when creating Lambda functions with SAM.

Following the type, a template usually lists configuration properties for a resource, depending on its type. The following three properties are required for a Lambda function:

- `CodeUri` points to the source code package for the function. This is usually a local directory with the source code, relative to the SAM template file.
- `Runtime` is the execution engine for the function.
- `Handler` is the main execution entry point.

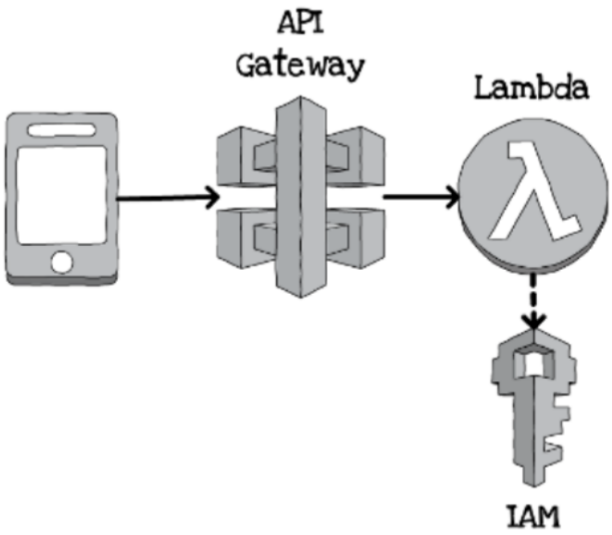
Specifying Lambda handlers

The Lambda handler is the main function responsible for processing incoming requests. The format of this setting depends on the runtime type. For Node.js, the format is `module.function`, where `module` is the file name (without the `.js` extension). The previous example points at `app.lambdaHandler`, meaning the function `lambdaHandler` in the `app.js` file.

Serverless functions can have some other properties, which will be introduced in the following chapters. For now, only the `Events` property is important. Lambda functions usually run in response to some external events. The `Events` property configures a set of triggers for the function. Structurally, this property resembles the main `Resources` section. Each event has a logical ID followed by `Type` and `Properties`, which are type-specific event configuration attributes.

The example template sets up an HTTPS service, so it automatically declares an `Api` event type. This represents API Gateway, an AWS service that can accept client HTTP requests. You will use a browser to send HTTPS requests to API Gateway, and the gateway will pass requests to your Lambda function. For `Api` event types, it's important to specify the HTTP method and request path. The example template sets up a Lambda call when clients request the `/hello` path, using the `GET` method.

This template involves another AWS service, which is not explicitly listed. It is called Identity and Access Management (IAM). IAM lets you configure access policies for AWS resources. If you just create a Lambda function without configuring IAM, no other AWS service will be allowed to invoke it. To actually make the function do something useful, you need to let IAM know who is allowed to send requests to it. SAM makes this easy because it will automatically set up the correct security privileges from event definitions. Because you have a single event configured to invoke a Lambda function from API Gateway, SAM will tell IAM to allow this connection automatically.



Basic infrastructure set up by the sample template: API Gateway will be able to receive HTTP requests from clients and call a Lambda function to handle the requests. An IAM policy tells Lambda who is allowed to invoke it.

Outputs

The last few lines of the template specify outputs. These normally list key information required for the users of a template or for clients connecting to this application. You will see logical identifiers used as variables in the output values. CloudFormation will replace those values with actual physical resource identifiers once it creates the stack.

```
Outputs:
  HelloWorldApi:
    Description: "API Gateway endpoint URL for Prod stage for Hello Worl
d function"
    Value: !Sub "https://${ServerlessRestApi}.execute-api.${AWS::Region}.a
mazonaws.com/Prod/hello/"
  HelloWorldFunction:
    Description: "Hello World Lambda Function ARN"
    Value: !GetAtt HelloWorldFunction.Arn
  HelloWorldFunctionIamRole:
    Description: "Implicit IAM Role created for Hello World function"
    Value: !GetAtt HelloWorldFunctionRole.Arn
```

The first web service I created with Lambda, in early 2016 before SAM was available, had less than 20 lines of code but required about 200 lines of configuration scripts to set everything up. With SAM, you can achieve the

same thing in just 10-15 lines of CloudFormation templates. Not bad!

> ### Serverless functions or Lambda functions
>
> CloudFormation has another resource for Lambda functions: `AWS::Lambda::Function`. SAM extends it with `AWS::Serverless::Function`. The benefit of using the SAM version is that it automatically creates IAM policies and event triggers. If you want to have more direct control over IAM and trigger invocation, you can use the lower-level resource instead. However, you will then need to create security policies yourself.

Hopefully by now you have a clear idea about the configurations in `template.yaml` and were able to understand the concept of the CloudFormation template and stack.

Now, take a look at the JavaScript code for the Lambda function in the next lesson.