AWS® Certified Developer

Official Study Guide

Associate (DVA-C01) Exam

With Amazon Aurora Serverless, you also get the same high availability as traditional Amazon Aurora, which means that you get six-way replication across three Availability Zones inside of a region in order to prevent against data loss.

Amazon Aurora Serverless is great for infrequently used applications, new applications, variable workloads, unpredictable workloads, development and test databases, and multitenant applications. This is because you can scale automatically when you need to and scale down when application demand is not high. This can help cut costs and save you the heartache of managing your own database infrastructure.

Amazon Aurora Serverless is easy to set up, either through the console or directly with the CLI. To create an Amazon Aurora Serverless cluster with the CLI, you can run the following command:

```
aws rds create-db-cluster --db-cluster-identifier sample-cluster --engine aurora
--engine-version 5.6.10a \
--engine-mode serverless --scaling-configuration
MinCapacity=4,MaxCapacity=32,SecondsUntilAutoPause=1000,AutoPause=true \
--master-username user-name --master-user-password password \
--db-subnet-group-name mysubnetgroup --vpc-security-group-ids sg-c7e5b0d2
-region us-east-1
```

Amazon Aurora Serverless gives you many of the similar benefits as other serverless technologies, such as AWS Lambda, but from a database perspective. Managing databases is hard work, and with Amazon Aurora Serverless, you can utilize a database that automatically scales and you don't have to manage any of the underlying infrastructure.

AWS Serverless Application Model

The *AWS Serverless Application Model* (AWS SAM) allows you to create and manage resources in your serverless application with *AWS CloudFormation* to define your serverless application infrastructure as a SAM template. A *SAM template* is a JSON or YAML configuration file that describes the AWS Lambda functions, API endpoints, tables, and other resources in your application. With simple commands, you upload this template to AWS CloudFormation, which creates the individual resources and groups them into an *AWS CloudFormation stack* for ease of management. When you update your AWS SAM template, you re-deploy the changes to this stack. AWS CloudFormation updates the individual resources for you.

AWS SAM is an extension of AWS CloudFormation. You can define resources by using the AWS CloudFormation in your AWS SAM template. This is a powerful feature, as you can use AWS SAM to create a template of your serverless infrastructure, which you can then build into a DevOps pipeline. For example, examine the following:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: 'Example of Multiple-Origin CORS using API Gateway and Lambda'
```

```
Resources:
  ExampleRoot:
    Type: 'AWS::Serverless::Function'
    Properties:
      CodeUri: '.'
      Handler: 'routes/root.handler'
      Runtime: 'nodejs8.10'
      Events:
        Get:
          Type: 'Api'
          Properties:
            Path: '/'
            Method: 'get'
  ExampleTest:
    Type: 'AWS::Serverless::Function'
    Properties:
      CodeUri: '.'
      Handler: 'routes/test.handler'
      Runtime: 'nodejs8.10'
      Events:
        Delete:
          Type: 'Api'
          Properties:
            Path: '/test'
            Method: 'delete'
        Options:
          Type: 'Api'
          Properties:
            Path: '/test'
            Method: 'options'

Outputs:
    ExampleApi:
      Description: "API Gateway endpoint URL for Prod stage for API Gateway
Multi-Origin CORS Function"
      Value: !Sub "https://${ServerlessRestApi}.execute-api.${AWS::Region}
.amazonaws.com/Prod/"
    ExampleRoot:
      Description: "API Gateway Multi-Origin CORS Lambda Function (Root) ARN"
      Value: !GetAtt ExampleRoot.Arn
    ExampleRootIamRole:
      Description: "Implicit IAM Role created for API Gateway Multi-Origin CORS
Function (Root)"
      Value: !GetAtt ExampleRootRole.Arn
    ExampleTest:
      Description: "API Gateway Multi-Origin CORS Lambda Function (Test) ARN"
      Value: !GetAtt ExampleTest.Arn
    ExampleTestIamRole:
      Description: "Implicit IAM Role created for API Gateway Multi-Origin CORS
Function (Test)"
      Value: !GetAtt ExampleTestRole.Arn
```

In the previous code example, you create two AWS Lambda functions and then associate *three* different Amazon API Gateway endpoints to trigger those functions. To deploy this AWS SAM template, download the template and all of the necessary dependencies from here:

```
https://github.com/awslabs/serverless-application-model/tree/develop/
examples/apps/api-gateway-multiple-origin-cors
```

AWS SAM is similar to AWS CloudFormation, with a few key differences, as shown in the second line:

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

This important line of code transforms the AWS SAM template into an AWS CloudFormation template. Without it, the AWS SAM template will not work.

Similar to the AWS CloudFormation, you also have a `Resources` property where you define infrastructure to provision. The difference is that you provision serverless services with a new `Type` called `AWS::Serverless::Function`. This provisions an AWS Lambda function to define all properties from an AWS Lambda point of view. AWS Lambda includes `Properties`, such as `MemorySize`, `Timeout`, `Role`, `Runtime`, `Handler`, and others.

While you can create an AWS Lambda function with AWS CloudFormation using `AWS::Lambda::Function`, the benefit of AWS SAM lies in a property called `Event`, where you can tie in a trigger to an AWS Lambda function, all from within the `AWS::Serverless::Function` resource. This `Event` property makes it simple to provision an AWS Lambda function and configure it with an Amazon API Gateway trigger. If you use AWS CloudFormation, you would have to declare an Amazon API Gateway separately with `AWS::ApiGateway::Resource`.

To summarize, AWS SAM allows you to provision serverless resources more rapidly with less code by extending AWS CloudFormation.

AWS SAM CLI

Now that we've addressed AWS SAM, let's take a closer look at the AWS SAM CLI. With AWS SAM, you can define templates, in JSON or YAML, which are designed for provisioning serverless applications through AWS CloudFormation.

AWS SAM CLI is a command line interface tool that creates an environment in which you can develop, test, and analyze your serverless-based application, all locally. This allows you to test your AWS Lambda functions before uploading them to the AWS service. AWS SAM CLI also allows you to develop and test your code quickly, and this gives you the ability to test it locally, which allows you to develop it faster. Previously, you would have had to upload your code each time you wanted to test an AWS Lambda function. Now, with the AWS SAM CLI, you can develop faster and get your application out the door more quickly.

To use AWS SAM CLI, you must meet a few prerequisites. You must install Docker, have Python 2.7 or 3.6 installed, have pip installed, install the AWS CLI, and finally install the AWS SAM CLI. You can read more about how to install AWS SAM CLI at `https://github.com/awslabs/aws-sam-cli`.

With AWS SAM CLI, you must define three key things.

- You must have a valid AWS SAM template, which defines a serverless application.
- You must have the AWS Lambda function defined. This can be in any valid language that Lambda currently supports, such as Node.js, Java 8, Python, and so on.
- You must have an event source. An *event source* is simply an `event.json` file that contains all the data that the Lambda function expects to receive. Valid event sources are as follows:
  - Amazon Alexa
  - Amazon API Gateway
  - AWS Batch
  - AWS CloudFormation
  - Amazon CloudFront
  - AWS CodeCommit
  - AWS CodePipeline
  - Amazon Cognito
  - AWS Config
  - Amazon DynamoDB
  - Amazon Kinesis
  - Amazon Lex
  - Amazon Rekognition
  - Amazon Simple Storage Service (Amazon S3)
  - Amazon Simple Email Service (Amazon SES)
  - Amazon Simple Notification Service (Amazon SNS)
  - Amazon Simple Queue Service (Amazon SQS)
  - AWS Step Functions

To generate this JSON event source, you can simply run this command in the AWS SAM CLI:

```
sam local generate-event <service> <event>
```

AWS SAM CLI is a great tool that allows developers to iterate quickly on their serverless applications. You will learn how to create and test an AWS Lambda function locally in the "Exercises" section of this chapter.

AWS Serverless Application Repository

The *AWS Serverless Application Repository* enables you to deploy code samples, components, and complete applications quickly for common use cases, such as web and mobile backends, event and data processing, logging, monitoring, Internet of Things (IoT), and more. Each application is packaged with an AWS SAM template that defines the AWS resources. Publicly shared applications also include a link to the application's source code. There is no additional charge to use the serverless application repository. You pay only for the AWS resources you use in the applications you deploy.

You can also use the serverless application repository to publish your own applications and share them within your team, across your organization, or with the community at large. This allows you to see what other people and organizations are developing.

Serverless Application Use Cases

Case studies on running serverless applications are located at the following URLs:

The Coca-Cola Company:

`https://aws.amazon.com/blogs/aws/things-go-better-with-step-functions/`

FINRA:

`https://aws.amazon.com/solutions/case-studies/finra-data-validation/`

iRobot:

`https://aws.amazon.com/solutions/case-studies/irobot/`

Localytics:

`https://aws.amazon.com/solutions/case-studies/localytics/`

Summary

This chapter covered the AWS serverless core services, how to store your static files inside of Amazon S3, how to use Amazon CloudFront in conjunction with Amazon S3, how to integrate your application with user authentication flows using Amazon Cognito, and how to deploy and scale your API quickly and automatically with Amazon API Gateway.

Serverless applications have three main benefits: no server management, flexible scaling, and automated high availability. Without server management, you no longer have to provision or maintain servers. With AWS Lambda, you upload your code, run it, and focus on your application updates. With flexible scaling, you no longer have to disable Amazon EC2 instances to scale them vertically, groups do not need to be auto-scaled, and you do not need to create Amazon CloudWatch alarms to add them to load balancers. With AWS Lambda, you adjust the units of consumption (memory and execution time), and AWS adjusts the rest of the instance appropriately. Finally, serverless applications have built-in availability and fault tolerance. When periods of low traffic occur, you do not spend money on Amazon EC2 instances that do not run at their full capacity.

You can use an Amazon S3 web server to create your presentation tier. Within an Amazon S3 bucket, you can store HTML, CSS, and JavaScript files. JavaScript can create HTTP requests. These HTTP requests are sent to a REST endpoint service called Amazon API Gateway, which allows the application to save and retrieve data dynamically by triggering a Lambda function.

After you create your Amazon S3 bucket, you configure it to use static website hosting in the AWS Management Console and enter an endpoint that reflects your AWS Region.

Amazon S3 allows you to configure web traffic logs to capture information, such as the number of visitors who access your website in the Amazon S3 bucket.

One way to decrease latency and improve your performance is to use Amazon CloudFront with Amazon S3 to move your content closer to your end users. Amazon CloudFront is a serverless service.

The Amazon API Gateway is a fully managed service designed to define, deploy, and maintain APIs. Clients integrate with the APIs using standard HTTPS requests. Amazon API Gateway can integrate with a service-oriented multitier architecture. The Amazon API Gateway provides dynamic data in the logic or app tier.

There are three types of endpoints for Amazon API Gateway: regional endpoints, edge-optimized endpoints, and private endpoints.

In the Amazon API Gateway service, you expose addressable resources as a tree of API Resources entities, with the root resource (/) at the top of the hierarchy. The root resource is relative to the API's base URL, which consists of the API endpoint and a stage name.

You use Amazon API Gateways to help drive down the total response-time latency of your API. Amazon API Gateway uses the HTTP protocol to process these HTTP methods and send/receive data to and from the backend. Serverless data is sent to AWS Lambda to process.

You can use Amazon Route 53 to create a more user-friendly domain name instead of using the default host name (Amazon S3 endpoint). To support two subdomains, you create two Amazon S3 buckets that match your domain name and subdomain.

A stage is a named reference to a deployment, which is a snapshot of the API. Use a stage to manage and optimize a particular deployment. You create stages for each of your environments such as DEV, TEST, and PROD, so you can develop and update your API and applications without affecting production. Use Amazon API Gateway to set up authorizers with Amazon Cognito user pools on an AWS Lambda function. This enables you to secure your APIs.

An Amazon Cognito user pool includes a prebuilt user interface (UI) that you can use inside your application to build a user authentication flow quickly. A user pool is a user directory in Amazon Cognito. With a user pool, your users can sign in to your web or mobile app through Amazon Cognito. Users can also sign in through social identity providers such as Facebook or Amazon and through Security Assertion Markup Language (SAML) identity providers.

Amazon Cognito identity pools allow you to create unique identities and assign permissions for your users to help you integrate with authentication providers. With the combination of user pools and identity pools, you can create a serverless user authentication system.

You can choose how users sign in with a username, an email address, and/or a phone number and to select attributes. Attributes are properties that you want to store about your end users. You can also configure password policies. Multi-factor authentication (MFA) prevents anyone from signing in to a system without authenticating through two different sources, such as a password and a mobile device–generated token. You create an Amazon Cognito role to send Short Message Service (SMS) messages to users.

The AWS Serverless Application Model (AWS SAM) allows you to create and manage resources in your serverless application with AWS CloudFormation as a SAM template. A SAM template is a JSON or YAML file that describes the AWS Lambda function, API endpoints, and other resources. You upload the template to AWS CloudFormation to create a stack. When you update your AWS SAM template, you redeploy the changes to this stack, and AWS CloudFormation updates the resources. You can use AWS SAM to create a template of your serverless infrastructure, which you can then build into a DevOps pipeline.

The `Transform: 'AWS::Serverless-2016-10-31'` code converts the AWS SAM template into an AWS CloudFormation template.

The AWS Serverless Application Repository enables you to deploy code samples, components, and complete applications for common use cases. Each application is packaged with an AWS SAM template that defines the AWS resources.

Additionally, you learned the differences between the standard three-tier web applications and the AWS serverless stack. You learned how to build your infrastructure quickly with AWS SAM and AWS SAM CLI for testing and development purposes.

Exam Essentials

Know serverless applications' three main benefits. The benefits are as follows:

- No server management
- Flexible scaling
- Automated high availability

Know what no server management means. Without server management, you no longer have to provision or maintain servers. With AWS Lambda, you upload your code, run it, and focus on your application updates.

Know what flexible scaling means. With flexible scaling, you no longer have to disable Amazon Elastic Compute Cloud (Amazon EC2) instances to scale them vertically, groups do not need to be auto-scaled, and you do not need to create Amazon CloudWatch alarms to add them to load balancers. With AWS Lambda, you adjust the units of consumption (memory and execution time), and AWS adjusts the rest of the instances appropriately.

Know what serverless applications mean. Serverless applications have built-in availability and fault tolerance. You do not need to architect for these capabilities, as the services that run the application provide them by default. Additionally, when periods of low traffic occur on the web application, you do not spend money on Amazon EC2 instances that do not run at their full capacity.

Know what services are serverless. On the exam, it is important to understand which Amazon services are serverless and which ones are not. The following services are serverless:

- Amazon API Gateway
- AWS Lambda
- Amazon SQS
- Amazon SNS
- Amazon Kinesis
- Amazon Cognito
- Amazon Aurora Serverless
- Amazon S3

Know how to host a serverless web application. Hosting a serverless application means that you need Amazon S3 to host your static website, which comprises your HTML, JavaScript, and CSS files. For your database infrastructure, you can use Amazon DynamoDB or Amazon Aurora Serverless. For your business logic tier, you can use AWS Lambda. For DNS services, you can utilize Amazon Route 53. If you need the ability to host an API, you can use Amazon API Gateway. Finally, if you need to decrease latency to portions of your application, you can utilize services like Amazon CloudFront, which allows you to host your content at the edge.

Resources to Review

Serverless Computing and Applications:

`https://aws.amazon.com/serverless/`

Amazon S3 Website Endpoints:

`https://docs.aws.amazon.com/general/latest/gr/rande.html#s3_website_region_endpoints`

Amazon Cognito FAQs:

`https://aws.amazon.com/cognito/faqs/`

Amazon API Gateway FAQ:

`https://aws.amazon.com/api-gateway/faqs/`

AWS Well-Architected Framework—Serverless Applications Lens:

`https://d1.awsstatic.com/whitepapers/architecture/AWS-Serverless-Applications-Lens.pdf`

Serverless Architectures with AWS Lambda:

`https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf`

AWS Serverless Multi-Tier Architectures (Amazon API Gateway and AWS Lambda):

`https://d1.awsstatic.com/whitepapers/AWS_Serverless_Multi-Tier_Architectures.pdf`

Serverless Streaming Architectures and Best Practices:

`https://d1.awsstatic.com/whitepapers/Serverless_Streaming_Architecture_Best_Practices.pdf`

Optimizing Enterprise Economics with Serverless Architectures:

`https://d1.awsstatic.com/whitepapers/optimizing-enterprise-economics-serverless-architectures.pdf`

Common Serverless Architectures discussed at re:Invent 2017 (Video):

`https://www.youtube.com/watch?v=xJcm9V2jagc`

AWS Serverless Application Model (AWS SAM) FAQs:

`https://aws.amazon.com/serverless/sam/faqs/`

Exercises

For this "Exercises" section, expand the OpenPets API Template that comes with Amazon API Gateway and build a frontend with HTML and JavaScript. You use AWS Lambda for some compute processing to save data to an Amazon DynamoDB database.

EXERCISE 13.1

Create an Amazon S3 Bucket for the Swagger Template

In this exercise, you use an AWS SAM template and a Swagger template to deploy your infrastructure. You will need to create an Amazon S3 bucket for the Swagger file.

1. Create an Amazon S3 bucket.

```
aws s3 mb s3://my-bucket-name --region us-east-1
```

   If the command was successful, you should see output similar to the following, which means the bucket has been created:

```
make_bucket: my-bucket-name
```

2. Upload the **Swagger template**.

```
aws s3 cp petstore-api-swagger.yaml s3://my-bucket-name/petstore-api-swagger
.yaml
```

   If the file was successfully uploaded, you should be able to navigate to the Amazon S3 bucket and see it. This file is for the Swagger template, and it is used to create the REST API inside the Amazon API Gateway. You have not yet deployed the API.

3. Use AWS SAM to deploy your serverless infrastructure. To package your SAM template, run the following command:

```
aws cloudformation package \
    --template-file ./petStoreSAM.yaml \
    --s3-bucket my-bucket-name \
    --output-template-file petStoreSAM-output.yaml \
    --region us-east-1
```

   If the command was successful, you should see that the file has been uploaded, and a new file called `petStoreSAM-output.yaml` has been created locally. You have packaged the AWS SAM template and converted it to a full AWS CloudFormation template. You will use this template in the next step to deploy the package to the Amazon API Gateway.

4. Deploy the package.

```
aws cloudformation deploy \
    --template-file ./petStoreSAM-output.yaml \
    --stack-name petStoreStack \
    --capabilities CAPABILITY_IAM \
    --parameter-overrides S3BucketName=s3://my-bucket-name/
petstore-api-swagger.yaml \
    --region us-east-1
```

If the command was successful, you should see that the `cloudformation` stack has been deployed. While it is in the process of deploying the resources, you will see something similar to the following:

```
Waiting for stack create/update to complete
```

This make take a few minutes. When it is finished deploying, the console displays the following message:

```
Successfully created/updated stack - petStoreStack
```

You have now successfully deployed the `cloudformation` stack and can view the resources it created inside the AWS Management Console under the **AWS CloudFormation** service.

5. After the stack is created, run the command and write the results down for subsequent steps:

```
aws cloudformation describe-stacks --stack-name petStoreStack --region
us-east-1 --query 'Stacks[0].Outputs[0].{PetStoreAPI:OutputValue}'
```

After running this command, the URL for the API is returned. Navigate to this URL to view the default page returned by the `PetStore` API. You will be changing this in the next exercise.

You have successfully completed the first exercise, created your AWS SAM template, and deployed it using AWS CloudFormation. Now your Amazon API Gateway is active, and you have the URL for accessing it.

EXERCISE 13.2

Edit the HTML Files

In steps 1 through 5, you are going to update the URL inside your `.html` files to point to the Amazon API Gateway stage that you have created. You do this so that your web application (`.html` files) knows the endpoint where to send your pet data.

1. Open `index.html` in the project folder and locate line 68 to find the variable named `api_gw_endpoint`. Input the value you retrieved from the previous command in Exercise 13.1.

```
var api_gw_endpoint = "https://cdvhqasdfnk444fe.execute-api.us-east-1
.amazonaws.com/PetStoreProd/"
```

2. Open `pets.html`.

(*continued*)

EXERCISE 13.2 *(continued)*

3. Input the value you received from the last command on line 96, and add `/pets` to the end of the string:

   ```
   var api_gw_endpoint = "https://cdvhqasdfnk444fe.execute-api.us-east-1.
   amazonaws.com/PetStoreProd/pets"
   ```

4. Open `add-pet.html`.
5. Input the value you received from the last command on line 87, and add `/pets` to the end.

   ```
   var api_gw_endpoint = "https://cdvhqasdfnk444fe.execute-api.us-east-1
   .amazonaws.com/PetStoreProd/pets"
   ```

6. Create a new Amazon S3 bucket for your website.

   ```
   aws s3 mb s3://my-bucket-name --region us-east-1
   ```

7. Copy the project files to the website.

   ```
   aws s3 cp . s3://my-bucket-name --recursive
   aws s3 rm s3://my-bucket-name/sam –recursive
   ```

   Here you are uploading all the files from your project folder to Amazon S3 and then removing the SAM template from the bucket. You do not want others to have access to your template files and AWS Lambda functions. You want others to have access only to the end application.

8. Change the Amazon S3 bucket name inside of the `policy.json` to your bucket name. This will be on line 12.
9. Enable public read access for the bucket:

   ```
   aws s3api put-bucket-policy --bucket my-bucket-name --policy file://policy.json
   ```

   If successful, this command will not return any information. You are enabling the Amazon S3 bucket to be publicly accessible, meaning that everyone can access your website.

10. Enable the static website.

    ```
    aws s3 website s3://my-bucket-name/ --index-document index.html --error-
    document index.html
    ```

    The Amazon S3 bucket now acts as a web server and is running your pet store application.

11. Navigate to the website.

    ```
    url: http://my-bucket-name.s3-website-us-east-1.amazonaws.com/index.html
    ```

12. Navigate Amazon API Gateway, AWS Lambda, Amazon DynamoDB, and the AWS SAM template to view the configuration.

    Now that the application has been deployed, you can view all the individual components inside the AWS Management Console.

    Inside Amazon API Gateway, you should see the `PetStoreAPIGW`. If you review the resources, you will see the various HTTP methods that you are allowing for your API.

    In AWS Lambda, two functions were created: `savePet` for saving pets to Amazon DynamoDB and `getPets` for retrieving pets stored in Amazon DynamoDB.

    In Amazon DynamoDB, you should have a table called `PetStore`. You can view the items in this table, though by default there should be none. After you create your first pet, however, you will be able to see some items in the table.

    You can view the AWS SAM template and the AWS CloudFormation stack to see exactly how each of these resources were created.

WARNING

With YAML, tab indentations are extremely important. Make sure that you have a valid YAML template. There are a variety of tools that you can use to validate YAML syntax. You can use the following websites to validate the YAML:

`https://codebeautify.org/yaml-validator`

`http://www.yamllint.com/`

If you want to perform client-side validation and not use a website, a number of IDEs support YAML validation. Refer to your IDE documentation to check for YAML support.

EXERCISE 13.3

Define an AWS SAM Template

In this exercise, you will develop an AWS Lambda function locally and then test that Lambda function using the AWS SAM CLI. To perform this exercise successfully, you must have AWS SAM CLI installed. For information on how to install the AWS SAM CLI, review the following documentation: `https://github.com/awslabs/aws-sam-cli`. The following steps assume that you have a working AWS SAM CLI installation.

1. Once you have installed AWS SAM CLI, open your favorite integrated development environment (IDE) and define an AWS SAM template.

(*continued*)

EXERCISE 13.3 *(continued)*

2. Enter the following in your template file:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Description: Welcome to the Pet Store Demo

Resources:
  PetStore:
    Type: AWS::Serverless::Function
    Properties:
      Runtime: nodejs8.10
      Handler: index.handler
```

3. Save the file as `template.yaml`.

You have created the SAM template and saved the file locally. In subsequent exercises, you will use this information to execute an AWS Lambda function.

EXERCISE 13.4

Define an AWS Lambda Function Locally

Now that you have a valid SAM template, you can define your AWS Lambda function locally. In this example, use Nodejs 8.10, but you can use any AWS Lambda supported language.

1. Open your favorite IDE, and type the following Nodejs code:

```
'use strict';

//A simple Lambda function
exports.handler = (event, context, callback) => {

    console.log('This is our local lambda function');
    console.log('Creating a PetStore service');
    callback(null, "Hello " + event.Records[0].dynamodb.NewImage.Message.S + "!
What kind of pet are you interested in?");
}
```

2. Save the file as `index.js`.

You have two files: an `index.js` and the SAM template. In the next exercise, you will generate an event source that will be used as the trigger for the AWS Lambda function.

EXERCISE 13.5

Generate an Event Source

Now that you have a valid SAM template and a valid AWS Lambda Nodejs 8.10 function, you can generate an event source.

1. Inside your terminal, type the following to generate an event source:

   ```
   sam local generate-event dynamodb update > event.json
   ```

   This will generate an Amazon DynamoDB update event. For a list of all of the event sources, type the following:

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

2. Modify the event source JSON file (`event.json`). On line 17, change New Item! to your first and last names.

   ```
   "S": "John Smith"
   ```

You have now configured the three pieces that you need: the AWS SAM template, the AWS Lambda function, and the event source. In the next exercise, you will be able to run the AWS Lambda function locally.

EXERCISE 13.6

Run the AWS Lambda Function

Trigger and execute the AWS Lambda function.

1. In your terminal, type the following to execute the AWS Lambda function:

   ```
   sam local invoke "PetStore" -e event.json
   ```

   You will see the following message:

   Hello Casey Gerena! What kind of pet are you interested in?

   The AWS Lambda Docker image is downloaded to your local environment, and the `event.json` serves as all of the data that will be received as an event source to the AWS Lambda function. Inside the AWS SAM template, you will have given this function the name `PetStore`; however, you can define as many functions as you need to in order to build your application.

EXERCISE 13.7

Modify the AWS SAM template to Include an API Locally

To make your pet store into an API, modify the `template.yaml`.

1. Open the `template.yaml` file, and modify it to look like the following:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Description: Welcome to the Pet Store Demo

Resources:
  PetStore:
    Type: AWS::Serverless::Function
    Properties:
      Runtime: nodejs8.10
      Handler: index.handler
      Events:
        PetStore:
          Type: Api
          Properties:
            Path: /
            Method: any
```

2. Save the `template.yaml` file.

You have modified the AWS SAM template to connect an Amazon API Gateway event for any method (`GET`, `POST`, and so on) to the AWS Lambda function. In the next exercise, you will modify the AWS Lambda function to work with the API.

EXERCISE 13.8

Modify Your AWS Lambda Function for the API

After you have defined an API, modify your AWS Lambda function.

1. Open the `index.js` file, and make the following changes:

```
'use strict';

//A simple Lambda function
exports.handler = (event, context, callback) => {

    console.log('DEBUG: This is our local lambda function');
    console.log('DEBUG: Creating a PetStore service');

    callback(null, {
        statusCode: 200,
        headers: { "x-petstore-custom-header": "custom header from petstore
service" },
        body: '{"message": "Hello! Welcome to the PetStore. What kind of Pet
are you interested in?"}'
    })

}
```

2. Save the `index.js` file.

   You have modified the AWS Lambda function to respond to an API REST request. However, you have not actually executed anything—you will do that in the next exercise.

EXERCISE 13.9

Run Amazon API Gateway Locally

Now that you have everything defined, run Amazon API Gateway locally.

1. Open a terminal and type the following:

   ```
   sam local start-api
   ```

   You will see output that looks like the following. Take note of the URL.

   ```
   2018-10-11 23:05:25 Mounting PetStore at http://127.0.0.1:3000/hello [GET]
   2018-10-11 23:05:25 You can now browse to the above endpoints to invoke your
   functions. You do not need to restart/reload SAM CLI while working on your
   functions changes will be reflected instantly/automatically. You only need to
   restart SAM CLI if you update your AWS SAM template
   2018-10-11 23:05:25  * Running on http://127.0.0.1:3000/ (Press CTRL+C to quit)
   ```

2. Open a web browser, and navigate to the previous URL.

   You will see the following message:

   ```
   Message: "Hello! Welcome to the Pet Store. What kind of Pet are you interested in?"
   ```

   When you navigate to the URL, the local API Gateway forwards the request to AWS Lambda, which is also running locally, provided by `index.js`. You can now build serverless applications locally. When you are ready to deploy to a development or production environment, deploy the serverless applications to the AWS Cloud with AWS SAM. This allows developers to iterate through their code quickly and make improvements locally.

JON BONSO AND KENNETH SAMONTE

AWS CERTIFIED

DEVOPS ENGINEER PROFESSIONAL

Tutorials Dojo
Study Guide and Cheat Sheets

AWS Serverless Application Model (SAM)

- An open-source framework for building serverless applications.
- It provides shorthand syntax to express functions, APIs, databases, and event source mappings.
- You create a **JSON** or **YAML** configuration template to model your applications.
- During deployment, SAM transforms and expands the SAM syntax into **AWS CloudFormation syntax**. Any resource that you can declare in an AWS CloudFormation template you can also declare in an AWS SAM template.
- The **SAM CLI** provides a Lambda-like execution environment that lets you locally build, test, and debug applications defined by SAM templates. You can also use the SAM CLI to deploy your applications to AWS.
- You can use AWS SAM to build serverless applications that use **any runtime supported by AWS Lambda**. You can also use SAM CLI to locally debug Lambda functions written in Node.js, Java, Python, and Go.
- Commonly used SAM CLI commands
    - The **sam init** command generates pre-configured AWS SAM templates.
    - The **sam local c**ommand supports local invocation and testing of your Lambda functions and SAM-based serverless applications by executing your function code locally in a Lambda-like execution environment.
    - The **sam package** and **sam deploy c**ommands let you bundle your application code and dependencies into a “deployment package" and then deploy your serverless application to the AWS Cloud.
    - The **sam logs** command enables you to fetch, tail, and filter logs for Lambda functions.
    - The output of the **sam publish** command includes a link to the AWS Serverless Application Repository directly to your application.
    - Use **sam validate** to validate your SAM template.