# Build a Serverless app using API Gateway, Lambda and DynamoDB

## Introduction

### Overview

The .NET core serverless application provides a basic web service using [Amazon API Gateway](https://aws.amazon.com/api-gateway/) and [AWS Lambda](https://aws.amazon.com/lambda/) to retrieve reading list details stored in [Amazon DynamoDB](https://aws.amazon.com/dynamodb/).

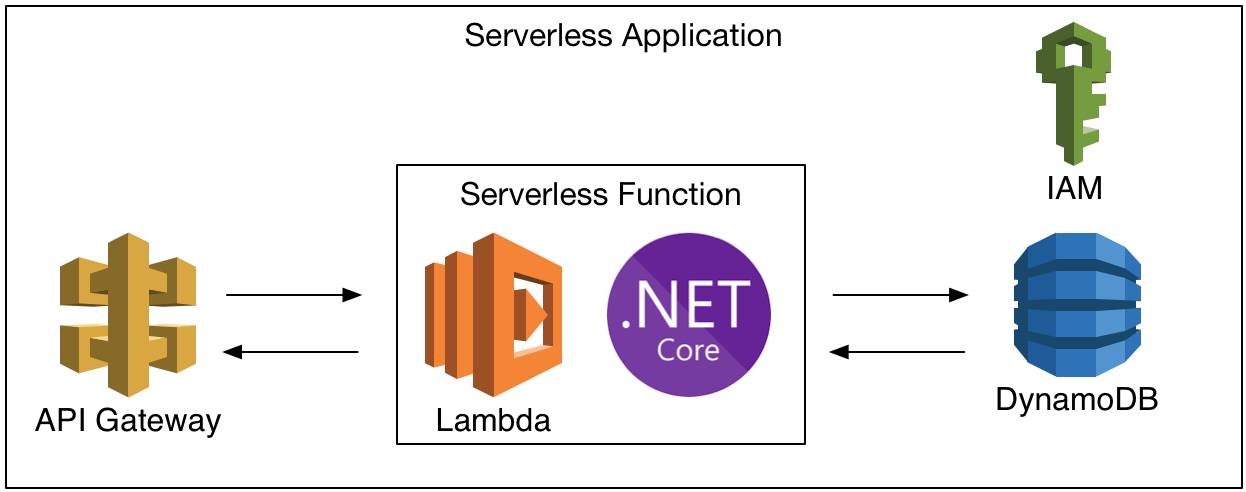
The application uses Amazon API Gateway to provide a simple RESTful API that can be called from client applications to display or act on the data. The API Gateway passes requests on to an AWS Lambda function which queries  an Amazon DynamoDB table for the reading list data. The AWS Lambda function converts data returned from DynamoDB into JSON, and then returns the converted data back to the API Gateway for initial request.

The application demonstrates how .NET Core can be used to quickly create serverless applications, allowing teams to develop, deploy and run code without provisioning or managing servers, and only paying for the compute time and storage consumed by the application.

The code for this tutorial is available to download from a git repository, although downloading it is optional since all the steps are listed to create the code from scratch.

### Architecture

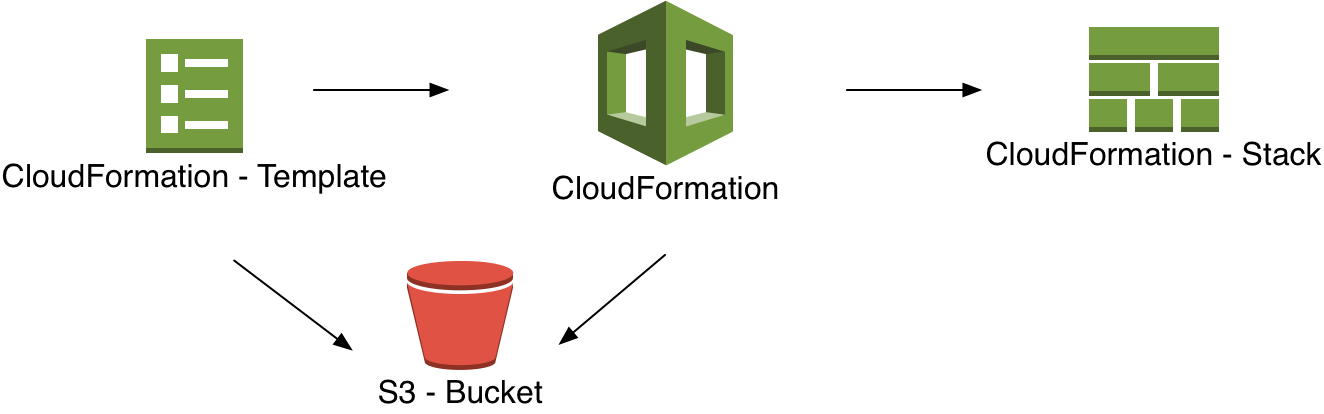
The application architecture uses Amazon API Gateway to communicate with an AWS Lambda function written in .NET Core, which calls Amazon DynamoDB, with [AWS Identity and Access Management](https://aws.amazon.com/iam/) (IAM) providing access control between the components, as shown below:



### Deployment

Deployment of the application components is handled by [AWS CloudFormation](https://aws.amazon.com/cloudformation) which uses the [AWS Serverless Application Model](https://docs.aws.amazon.com/lambda/latest/dg/deploying-lambda-apps.html) (AWS SAM) to simplify the template.

The CloudFormation template contains configuration details for each component, and also references an S3 bucket used to stage deployment artifacts, such as the .NET Core assembly. On execution, CloudFormation uses the template and the S3 bucket to create a CloudFormation stack which then deploys the application components.



### Modules

This workshop is broken up into four modules. You must complete each module before proceeding to the next.

1. Setup Environment
2. Create a Basic Lambda Function
3. Integrate Lambda with DynamoDB
4. Delete Serverless Resources

### Side-bar

* AWS Experience—Beginner
* Time to Complete—60 mins
* Cost to Complete—Each service used in this architecture is eligible for the AWS Free Tier. If you are outside the usage limits of the Free Tier, completing this learning path will cost you less than $0.25\*.
* Tutorial Prereqs—To complete this learning path, you will need:  
  ✓ An AWS Account\*\*

✓ An IAM user with access key credentials\*\*\*  
✓ (Optionally) Visual Studio 2017 for Windows

\*This estimate assumes you follow the recommended configurations throughout the tutorial and terminate all resources within 24 hours.

\*\*Accounts that have been created within the last 24 hours might not yet have access to the resources required for this learning. If you don’t have an account visit <https://aws.amazon.com> and click **Sign Up**.

\*\*\* You must have a set of valid AWS credentials, consisting of an access key and a secret key, which are used to sign programmatic requests to AWS. You can obtain a set of account credentials when you create your account, although we recommend you do not use these credentials and instead [create an IAM user](http://docs.aws.amazon.com/IAM/latest/UserGuide/Using_SettingUpUser.html) and use those credentials.

## Module 1: Setup Environment

In this module, you'll configure your development environment for working with AWS Lambda functions. These instructions provide 2 different options to choose from for your development environment, Visual Studio 2017 for Windows, or the .NET Core CLI on Windows, Mac, or Linux, using an editor of your choice.

### Implementation Instructions

Follow the step-by-step instructions below to setup your development environment. (To expand the section, click on each step number)

#### Step 1: Setup Visual Studio 2017 for Windows

If your development environment is Visual Studio 2017 on Windows, you will need to ensure the following components are installed:

1. The **.NET Core SDK 2.x** for Windows:   
   <https://www.microsoft.com/net/download/>
2. Visual Studio 2017 version 15.3 or later
3. The AWS Toolkit for Visual Studio:  
   <https://aws.amazon.com/visualstudio/>

#### Step 1: Setup .NET Core CLI on Windows, Mac, or Linux

If you are using .NET Core CLI on Windows, Mac, or Linux, you will need to install a few components, as follows:

1. The **.NET Core SDK 2.x** for Windows, Mac, or Linux:   
   <https://www.microsoft.com/net/download/>
2. Install the AWS Lambda templates with the AWS Lambda NuGet package by running dotnet new -i Amazon.Lambda.Templates::\* in a terminal window.
3. Verify the new AWS Lambda templates have been installed by running dotnet new lambda.EmptyFunction -l in a terminal window. If the command returns details of a single Lambda Empty Function template then the templates have been installed correctly.

You will also need the AWS Command Line Interface (AWS CLI) installed to Dynamo DB and S3 Bucket:

1. The AWS CLI for Windows, Mac, or Linux: <https://aws.amazon.com/cli/>
2. Once installed, you can configure the CLI by running the aws configure command in a terminal or command-line window.
3. When prompted, enter your AWS Access Key ID and press **Return**.
4. You will then be prompted for your AWS Secret Access Key, which you should enter and then press **Return**.
5. For the default region name you should enter your chosen region code (e.g. eu-west-1)
6. Finally, for the default output format you can just press **Return**.

Finally, you will also need a text editor or an IDE for modifying, such as vi, emacs, nano, [Visual Studio for Mac](https://www.microsoft.com/net/download/), or [Visual Studio Code](https://code.visualstudio.com/) for Windows, Mac, or Linux.

### Side-bar

* Time to Complete—10 mins

## Module 2: Create a Basic Lambda Function

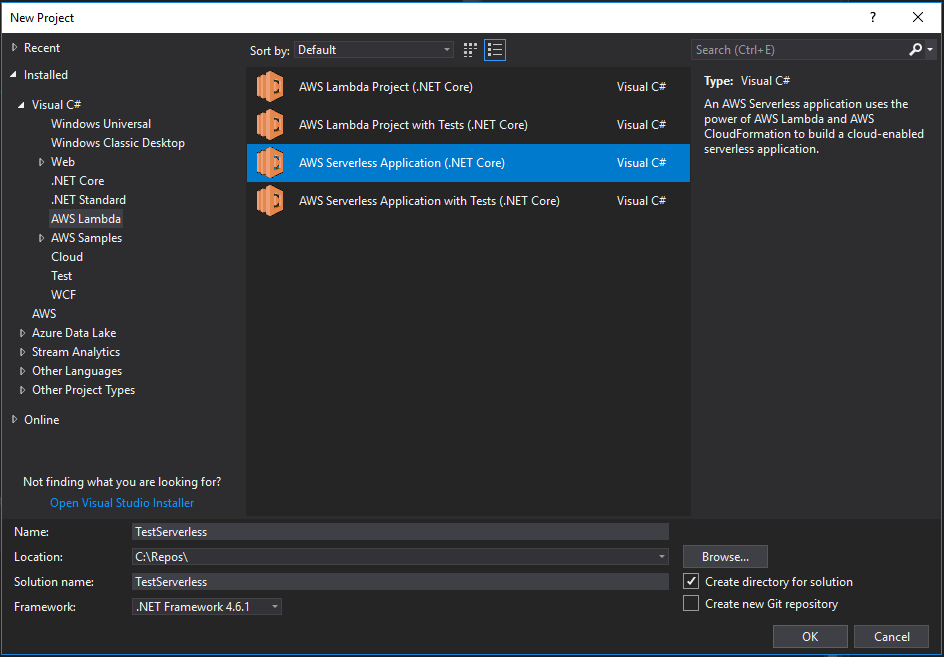
In this module you'll create a project for your Lambda function, create a basic Lambda function, and an API Gateway for calling the Lambda function. These instructions provide options for 3 different development environments: Visual Studio 2017 for Windows, Visual Studio for Mac, or .NET Core CLI on Windows, Mac, or Linux.

### Implementation Instructions

Follow the step-by-step instructions below to setup your development environment. (To expand the section, click on each step number)

#### Step 1: Create Solution in Visual Studio 2017 for Windows

If you are using Visual Studio 2017 on Windows as your development environment, you can create a solution as follows:

1. Open the **Visual Studio 2017** desktop application.
2. Create a new project by selecting **File > New > Project**.
3. In the **New Project** dialog, go to **Installed>Visual C#>AWS Lambda** and select **AWS Serverless Application,** entering the name **TestServerless** for the project, as shown in the screenshot below:  
   
4. In the **New AWS Serverless Application** dialog, select **Empty Serverless Application** and click **Finish**. Visual Studio will take a few moments to create the solution and will then open the **Solution Explorer** window containing a variety of files.

#### Step 1: Create Solution using .NET Core CLI

If you are using .NET Core CLI on Windows, Mac, or Linux, you will need to do the following:

1. Open a command line or terminal window.
2. Navigate to the directory you want to use as the parent for your solution directory.
3. Create a solution called **TestServerless** by executing the following command dotnet new lambda.EmptyServerless -n TestServerless

#### Step 2: Open Solution Files in Visual Studio 2017 for Windows

The solution will already contain the skeleton code required to build and deploy a basic Lambda function and an API Gateway, but before deployment it’s worth having a look at the key files.

In Visual Studio’s **Solution Explorer** expand the **TestServerless** project, and then open the **Function.cs** file, which contains the code for our basic Lambda function, and also the **serverless.template** file, which contains the CloudFormation template for deployment.

#### Step 2: Open Solution Files using .NET Core CLI

The solution will already contain the skeleton code required to build and deploy a basic Lambda function and an API Gateway, but before deployment it’s worth having a look at the key files.

To edit the solution code you will need to use a text editor or an IDE for modifying, such as vi, emacs, nano, [Visual Studio for Mac](https://www.microsoft.com/net/download/), or [Visual Studio Code](https://code.visualstudio.com/) for Windows, Mac, or Linux.

In a terminal or command-line window, start by navigating to the /**TestServerless/src/TestServerless** directory, and then open the **Function.cs** file, which contains the code for our basic Lambda function, and the **serverless.template** file, which contains the CloudFormation template for deployment

#### Step 3: Review code

The **Function.cs** file contains the C# code for the auto-generated Lambda function.

The top of the file contains a number of using statements, after which is an assembly attribute that enables automatic deserialization of requests from the API Gateway into APIGatewayProxyRequest objects, and also automatically serializes responses from APIGatewayProxyResponse objects.

[assembly: LambdaSerializer(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer)]

The rest of the file contains the Function class, which contains a single Get method to handle requests from the API Gateway and return a static response, "Hello AWS Serverless".

public APIGatewayProxyResponse Get(APIGatewayProxyRequest request,   
ILambdaContext context)  
{  
 context.Logger.LogLine("Get Request\n");  
  
 var response = new APIGatewayProxyResponse  
 {  
 StatusCode = (int)HttpStatusCode.OK,  
 Body = "Hello AWS Serverless",  
 Headers = new Dictionary<string, string>

{ { "Content-Type", "text/plain" } }  
    };  
  
 return response;          
}

This is all the barebones code we need to create a basic Lambda function.

After looking in the **Function.cs** file, the **serverless.template** file contains the CloudFormation template that can be used for deploying the solution’s components.

While by default the template contains all the details for creating the API Gateway and Lambda components, there are a couple of changes you’ll need to make to make customize the Lambda function’s settings.

First of all, you’ll add a FunctionName parameter with a value of **TestServerless**, which stops CloudFormation from adding a physical id onto the function’s name.

Secondly, since our Lambda function is very simple it won’t need much memory to run so we can reduce the requested memory down from 256MB to 128MB by editing the MemorySize parameter.

Both these changes are shown below in **bold**:

{  
  "AWSTemplateFormatVersion" : "2010-09-09",  
  "Transform" : "AWS::Serverless-2016-10-31",  
  "Description" : "An AWS Serverless Application.",  
  "Parameters" : {  
  },  
  "Resources" : {  
    "GetReadingList" : {  
      "Type" : "AWS::Serverless::Function",  
      "Properties": {  
        "Handler": "TestServerless::TestServerless.  
Functions::Get",  
 **"FunctionName": "TestServerless",**        "Runtime": "dotnetcore2.0",  
        "CodeUri": "",  
        "Description": "",  
        "MemorySize": **128**,  
        "Timeout": 30,  
        "Role": null,  
        "Policies": [ "AWSLambdaBasicExecution" ],  
        "Events": {  
          "PutResource": {  
            "Type": "Api",  
            "Properties": {  
              "Path": "/",  
              "Method": "GET"  
            }  
          }  
        }  
      }  
    },   
 "Outputs": {  
 }  
}

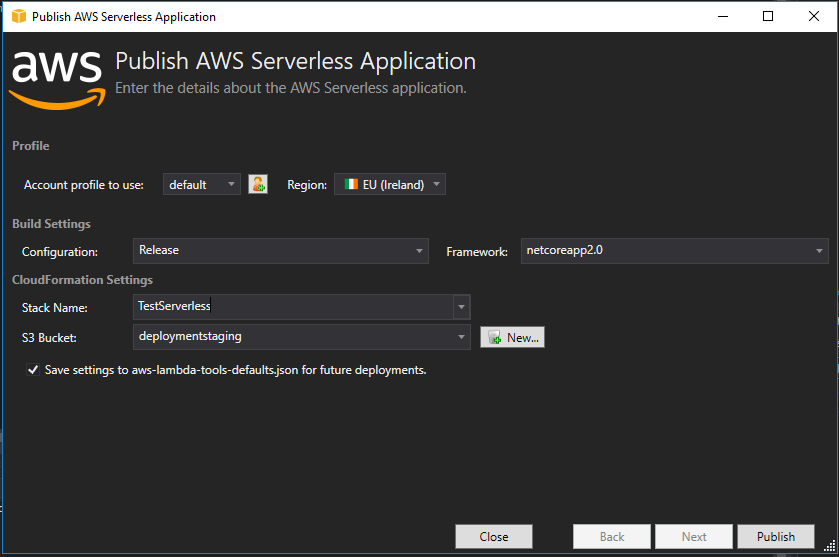
Save your changes to the file and you are now ready to build the solution and deploy to AWS.

#### Step 3: Build and Deploy using Visual Studio 2017 for Windows

Before deploying the solution it’s worth checking that everything builds correctly, and you can do this by opening Visual Studio’s **Build** menu and then selecting **Build Solution**.

Once the build has completed successfully the status bar will show a message *Build succeeded*, and you are now ready to deploy the solution to AWS.

To deploy the solution:

1. Right-click on the project in **Solution Explorer** and select **Publish to AWS Lambda…**
2. The **Publish AWS Serverless Application** dialog will appear, and you’ll need to select an AWS Region to deploy to, a unique stack name for the CloudFormation stack, and an S3 bucket to store the deployment artifacts, as shown below:  
   
3. If you already have a suitable S3 bucket for storing deployment files you can select it in the drop-down, otherwise create a new S3 bucket by clicking **New** next the S3 Bucket drop-down, entering a suitable name, and then clicking **Ok**.
4. Once you’ve completed the form, click on the **Publish** button to deploy to AWS. The dialog will display a **Publishing** progress bar and will show all the deployment actions as it carries them out.
5. Once publishing completes Visual Studio will display a dialog informing you the **aws-lambda-tools-defaults.json** file has been updated with the details you entered on the **Publish AWS Serverless Application** dialog. Click **Yes** to save the changes.
6. Visual Studio will now display details of the CloudFormation stack in a **Stack** window, and you can use the window to review including the CloudFormation actions that were executed, and the resources created during deployment.

You are now ready to test the deployed solution.

#### Step 3: Build and Deploy using .NET Core CLI

Before deploying the solution it’s worth checking that everything builds correctly. To build the solution using >NET Core CLI, do the following:

1. Open a terminal or command-line window
2. Navigate to the **/TestServerless/src/TestServerless** directory
3. Run the command dotnet build and wait until it prints the message **Build succeeded**.

Once the build has completed you are almost ready to deploy the solution to AWS. However, before deploying the solution you will need to choose a region to deploy the resources to, and you will also need an S3 bucket in that region for storing the deployment assets.

If you already have a suitable S3 bucket then you can move on with the next step, otherwise you can create an S3 bucket, by running the following command in a terminal or command-line window, aws s3 mb s3://bucket-name

Bucket names must be DNS compliant and globally unique. They can contain lowercase letters, numbers, hyphens and periods, but they can only start and end with a letter or number.

When you’ve chosen a region and an S3 bucket, you can deploy your serverless application as follows:

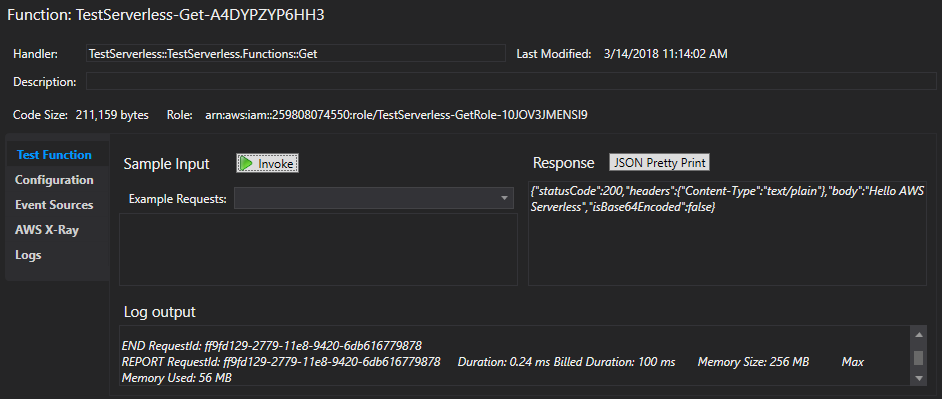
1. In the terminal or command-line window, execute the following command: dotnet lambda deploy-serverless
2. When the command asks for a CloudFormation stack name enter **TestServerless** and press **Return**.
3. When the command asks for an S3 bucket name, enter the name from earlier and press **Return**.
4. Finally, when the command asks for a region, enter the region to create the resources in (e.g. **eu-west-1**) and press **Return**.
5. The command will then display progress as it deploys the components to AWS, and on successful completion will print the message Stack finished updating with status: CREATE\_COMPLETE

You are now ready to test the deployed solution.

#### Step 4: Test API Gateway and Lambda in Visual Studio 2017 for Windows

To test the deployed solution, expand the **AWS Explorer** window in Visual Studio, and then under **AWS Lambda** double-click on the **TestServerless** Lambda function to open its **Function** window.

The window contains a number of tabs, and you can use the **Test Function** tab to test the function by clicking the **Invoke** button.



After the call to the API Gateway and the Lambda function completes, you will see the full JSON response in the **Response** area of the screen, and the **Log output** area shows details of the request, including its execution time.

Congratulations! You’ve now created a working Lambda function and deployed it along with an API Gateway endpoint to AWS.

If you want to play with the function you can easily change the message returned in the **Function.cs** file, re-deploy the solution, and then test it again.

#### Step 4: Test API Gateway and Lambda using .NET Core CLI

To test a call to the deployed API Gateway and Lambda function, run the following command from the terminal or command-line window: dotnet lambda invoke-function TestServerless -–region eu-west-1 replacing the region with the region code you selected previously.

When the command completes successfully you should see a Payload message displayed in the terminal or command-line window that contains the body provided by your Lambda function, "*Hello AWS Serverless*".

Congratulations! You’ve now created a working Lambda function and deployed it along with an API Gateway endpoint to AWS.

If you want to play with the function you can easily change the message returned in the **Function.cs** file, re-deploy the solution again, and then test it as above.

#### Side-bar

* Time to Complete—20 mins
* Services Used—Amazon Gateway API, AWS Lambda, AWS CloudFormation, Amazon S3

## Module 3: Integrate Lambda with DynamoDB

In this module you'll extend the solution from Module 2 by creating a DynamoDB table to store reading list details, and then updating the Lambda function to retrieve the reading list data from DynamoDB.

These instructions provide options for 2 different development environments: Visual Studio 2017 for Windows, or .NET Core CLI on Windows, Mac, or Linux.

### Implementation Instructions

Follow the step-by-step instructions below to setup your development environment. (To expand the section, click on each step number)

#### Step 1: Add NuGet Packages in Visual Studio 2017 for Windows

First of all, you need to add some additional NuGet packages to the Visual Studio solution, one to provide access to DynamoDB, and the other to carry out some rudimentary JSON formatting.

To add the required NuGet packages:

1. Open the solution from Module 2 in Visual Studio.
2. Open the Package Manager Console by going to Tools > NuGet Package Manager > Package Manager Console
3. In the **Package Manager Console** window, enter the following command Install-Package AWSSDK.DynamoDBv2
4. Once it has successfully installed, add the **Newtonsoft JSON** package by running the following command in the **Package Manager Console** window Install-Package Newtonsoft.Json

#### Step 1: Add NuGet Packages using .NET Core CLI on Windows, Mac, or Linux

Before updating the solution code, you first of all need to add some additional NuGet packages to the solution, one providing access to DynamoDB, and the other to help format the reading list results as JSON.

To add the required NuGet packages:

1. Open a command-line or terminal window.
2. Navigate to the **/TestServerless/src/TestServerless** directory
3. Run the following command dotnet add package AWSSDK.DynamoDBv2
4. Once it has successfully completed, run the command dotnet add package Newtonsoft.Json

#### Step 2: Integrate code with DynamoDB

Once the NuGet packages have been added to the solution, you’ll need to update the **Function.cs** file.

Start by adding the following using statements at the top of the file:

using Amazon.DynamoDBv2;  
using Amazon.DynamoDBv2.Model;  
using Newtonsoft.Json;

You’ll then need to create a new asynchronous function, ScanReadingListAsync, to call DynamoDB, to scan the table for data, and then converting the results into JSON:

private async Task<string> ScanReadingListAsync()  
{  
 string readingListJson;  
    using (var client = new AmazonDynamoDBClient(Amazon.RegionEndpoint.EUWest1))  
    {  
     var response = await client.ScanAsync(new ScanRequest("readingList"));  
        readingListJson = JsonConvert.SerializeObject(response.Items);  
    }  
  
    return (readingListJson);  
}

Since the code scans the DynamoDB table without setting a filter, it will return all the data contained in the table, which is fine for this example, but would need proper consideration for a production system to reduce the data retrieved by the Lambda function.

Next, you need to change the Get method by making it asynchronous so it can call the ScanReadingListAsync method. You do this by first of all adding an async modifier to the method declaration, then changing its return type to Task< APIGatewayProxyResponse>, and finally changing its name to GetAsync to fit with .NET naming conventions.

public async Task<APIGatewayProxyResponse> GetAsync(  
 APIGatewayProxyRequest request, ILambdaContext context)  
{

You then need to change the line that sets the Body of the response to call the new method, ScanReadingListAsync, remembering to add an await to its call, since it’s an asynchronous method.

The rest of the method stays the same as before.

context.Logger.LogLine("Get Request\n");  
  
    var response = new APIGatewayProxyResponse  
    {  
     StatusCode = (int)HttpStatusCode.OK,  
        **Body = await ScanReadingListAsync(),**  
        Headers = new Dictionary<string, string> { { "Content-Type", "text/plain" } }  
    };  
  
    return response;

}

Those are the only code changes needed in the Lambda function, although you also need to ensure the DynamoDB table is created when the solution is deployed, which can be done by editing the **serverless.template** file as follows:

1. Change the Handler in the AWS::Serverless::Function section to GetAsync, since we updated the name of the method in the **Function.cs** file.
2. Change the permissions granted to the Lambda function by updating the Policies parameter to AWSLambdaFullAccess, since it’ll now need permission to call the DynamoDB table.   
   N.B. While using AWSLambdaFullAccess for this example is fine, in a production system you should restrict the Lambda function’s permissions using the principle of least privilege.
3. Finally, you need to add a new resource section containing settings for the DynamoDB table.

All these changes are shown below in **bold**:

{  
  "AWSTemplateFormatVersion" : "2010-09-09",  
  "Transform" : "AWS::Serverless-2016-10-31",  
  "Description" : "Starting template for an AWS Serverless Application.",  
  "Parameters" : {  
  },  
  "Resources" : {  
    "GetReadingList" : {  
      "Type" : "AWS::Serverless::Function",  
      "Properties": {  
        "Handler": "ServerlessReadingList::ServerlessReadingList.Function::**GetAsync**",  
 "FunctionName": "TestServerless",  
        "Runtime": "dotnetcore2.0",  
        "CodeUri": "",  
        "Description": "",  
        "MemorySize": 128,  
        "Timeout": 30,  
        "Role": null,  
        "Policies": [ "**AWSLambdaFullAccess**" ],  
        "Events": {  
          "PutResource": {  
            "Type": "Api",  
            "Properties": {  
              "Path": "/readinglist",  
              "Method": "GET"  
            }  
          }  
        }  
      }  
    }**,**  
    **"readingListDynamoDBTable" : {  
      "Type" : "AWS::DynamoDB::Table",  
      "Properties" : {  
        "AttributeDefinitions" : [  
          {  
            "AttributeName" : "ItemId",  
            "AttributeType" : "S"     
          },  
          {  
            "AttributeName" : "Title",  
            "AttributeType" : "S"  
          }  
        ],  
        "KeySchema" : [  
          {  
            "AttributeName" : "ItemId",  
            "KeyType" : "HASH"  
          },  
          {  
            "AttributeName" : "Title",  
            "KeyType" : "RANGE"  
          }  
        ],  
        "ProvisionedThroughput" : {  
          "ReadCapacityUnits" : "1",  
          "WriteCapacityUnits" : "1"  
        },  
        "TableName" : "readingList"  
      }  
    }**  },  
  "Outputs" : {  
  }  
}

#### Step 2: Deploy in Visual Studio 2017 for Windows

Deploying the solution is almost exactly the same as before, but before deploying the solution it’s worth checking that everything builds correctly, and you can do this by opening Visual Studio’s **Build** menu and then selecting **Build Solution**.

Once the build has completed successfully you deploy the solution as follows:

1. Right-click on the project in **Solution Explorer** and select **Publish to AWS Lambda…**
2. The **Publish AWS Serverless Application** dialog will appear, although it should now be pre-populated with the answers you entered last time. Click **Publish** to deploy the components to AWS:
3. Once publishing completes Visual Studio will display details of the CloudFormation stack in a **Stack** window, and you can use the window to review including the CloudFormation actions that were executed, and the resources created during deployment.

You are now ready to test the deployed solution.

#### Step 2: Deploy using .NET Core CLI

Before deploying the solution it’s worth checking that everything builds correctly, and to build the solution using >NET Core CLI, do the following:

1. Open a terminal or command-line window
2. Navigate to the **/TestServerless/src/TestServerless** directory
3. Run the command dotnet build and wait until it prints the message **Build succeeded**.

You can now re-deploy your serverless application as follows:

1. In the terminal or command-line window, execute the following command: dotnet lambda deploy-serverless
2. When the command asks for a CloudFormation stack name enter **TestServerless** and press **Return**.
3. When the command asks for an S3 bucket name, enter the name from earlier and press **Return**.
4. Finally, when the command asks for a region, enter the region to create the resources in (e.g. **eu-west-1**) and press **Return**.
5. The command will then display progress as it deploys the components to AWS, and on successful completion will print the message Stack finished updating with status: CREATE\_COMPLETE

You are now ready to test the deployed solution.

#### Step 3: Test in Visual Studio 2017 for Windows

As before, to test the solution, expand the **AWS Explorer** window in Visual Studio, and then under **AWS Lambda** double-click on the **TestServerless** Lambda function to open its **Function** window.

You can then use the **Test Function** tab to test the function by clicking the **Invoke** button.

After the call to the API Gateway and the Lambda function completes, you will see the full JSON response in the **Response** area of the screen, although since there’s no data in the DynamoDB table the response will be minimal.

To make it a bit more interesting you can add some data as follows:

1. In the AWS Explorer expand the Amazon DynamoDB section and then double-click on the **readingList** table to open a Table window.
2. The Table window is split into two parts, the top past allowing you to set Scan settings, and the bottom part showing you items in the table.
3. Since the table is empty we’re going to add a couple of items. Click on the row under ItemId and enter a value A1 for ItemId, and Cross-platform .NET for the title.
4. To save the item to **DynamoDB** click **Commit Changes** at the top of the window.
5. Once the item has been added it will change colour, and you can add some other rows.

If you now go back to the **Function** window and invoke the **Test Function** again, your response will be full on JSON containing details of the items you just added to DynamoDB.

Congratulations! You’ve now extended your Lambda function to call Dynamo DB, deployed the updated Lambda function and DynamoDB table to AWS, and demonstrated that they’re working seamlessly together.

#### Step 3: Test using .NET Core CLI

To test, run the following command from the terminal or command-line window: dotnet lambda invoke-function TestServerless -–region eu-west-1 replacing the region with the region code you selected previously.

When the command completes successfully you should see a payload message displayed in the terminal or command-line window, although since there’s no data in the DynamoDB table the response will be minimal.

To make the test a bit more interesting you can add an item to the DynamoDB table by running the following command in a terminal or command-line window, remembering to change the region for your chosen region:

aws dynamodb put-item –-region **eu-west-1** \

--table-name readingList \

--item '{

"ItemId": {"S": "A1"},

"Title": {"S": "Cross-platform.NET"}}'

If you now go re-run the test command in the terminal or command-line window, the payload displayed will now be full of JSON containing details of the items you added to DynamoDB.

Congratulations! You’ve now extended your Lambda function to call Dynamo DB, deployed the updated Lambda function and DynamoDB table to AWS, and demonstrated that they’re working seamlessly together.

#### Side-bar

* Time to Complete—25 mins
* Services Used—Amazon Gateway API, AWS Lambda, Amazon DynamoDB, AWS CloudFormation, Amazon S3

## Module 4: Delete Serverless Resources

Once you’ve had enough playing with the solution you created in Modules 1–3, deleting the AWS resources is simply a matter of deleting the CloudFormation stack, which automatically deletes the associated resources.

These instructions provide options for 2 different development environments: Visual Studio 2017 for Windows, or .NET Core CLI on Windows, Mac, or Linux.

### Implementation Instructions

Follow the step-by-step instructions below to delete the AWS resources created in Modules 1–3. (To expand the section, click on each step number)

#### Step 1: Deleting the CloudFormation stack from Visual Studio 2017 for Windows

Once you’ve had enough playing with the solution, deleting the AWS resources is simply a matter of deleting the CloudFormation stack, which automatically deletes the associated resources.

To delete a CloudFormation stack from Visual Studio:

1. In AWS Explorer, expand the AWS Cloud Formation entry.
2. Right-click on the **TestServerless** stack and then select **Delete**.
3. A dialog will pop-up asking you if you’re sure you want to delete the stack. Click **Yes** to confirm deletion of the stack.

While it might seem a shame to delete a working solution, one of the great advantages of Cloud Formation is that recreating the resources is simply a matter of executing the publish command from Visual Studio and waiting briefly as the resources are automatically created by Cloud Formation.

#### Step 1: Deleting the CloudFormation stack using .NET Core CLI

Once you’ve had enough playing with the solution, deleting the AWS resources is simply a matter of deleting the CloudFormation stack, which automatically deletes the associated resources.

To delete a CloudFormation stack from Visual Studio:

1. In the terminal or command-line window, execute the following command: dotnet lambda delete-serverless
2. When the command asks for a CloudFormation stack name enter **TestServerless** and press **Return**.
3. Finally, when the command asks for a region, enter the region to create the resources in (e.g. **eu-west-1**) and press **Return**.
4. On successful deletion the command will print the message CloudFormation stack TestServerless deleted

One of the great advantages of CloudFormation is creating and deleting resources can be execute through commands, so if you want to re-create the serverless application, you just need to re-run the dotnet lambda deploy-serverless command and waiting briefly as the resources are automatically provisioned by CloudFormation.

#### Side-bar

* Time to Complete—5 mins
* Services Used—CloudFormation