

Module 10

Serverless Computing: AWS Lambda – Part 2

Overview

In Module 9, we created a simple AWS Lambda function that didn't do much other than print its argument payload and some other OS properties to CloudWatch. The purpose of Module 9 was to introduce serverless computing with AWS Lambda. In this module, we will create a Lambda function and use it in a more interesting and useful way. We will also read a real-world white paper where it is used in a somewhat similar fashion to what we will do in this module.

In Module 9, the code of the Lambda function was typed in the AWS console code editor. Another way of authoring a Lambda function is to use an IDE such as Visual Studio, VSCode, or Eclipse, and upload the function to AWS Lambda. For more complicated functions, this is preferred since you get the benefit of debugging and all other IDE help such as code completion, etc.

Objective of module

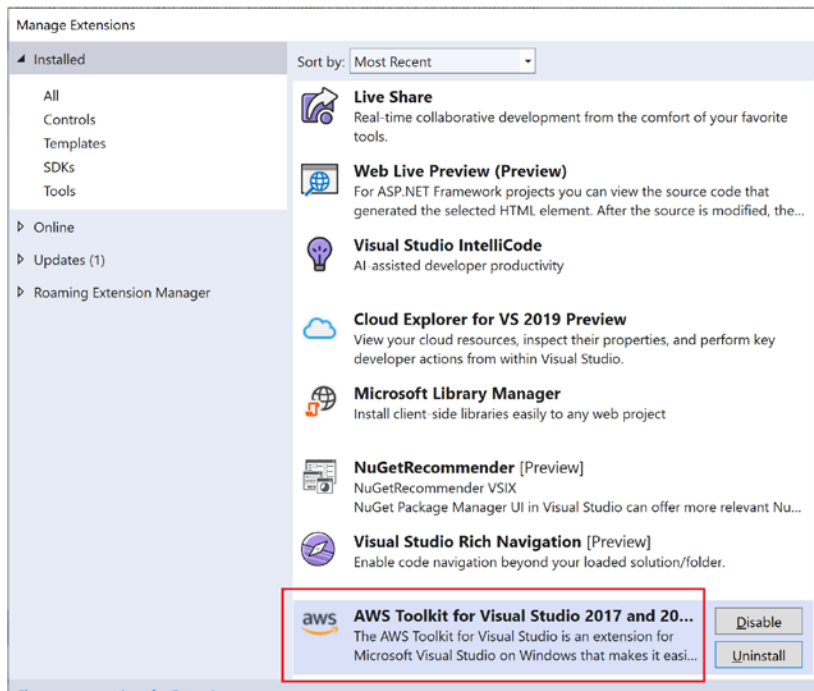
- Create an AWS Lambda function in C# and upload it to AWS.
- Have an S3 action trigger the Lambda function to execute.

Prerequisites

Verify that you have the AWS Toolkit for Visual Studio installed:

1. Open Visual Studio 2019 (or whatever version you have).
2. Dismiss the Open/Create project window.
3. Click menu Extensions → Manage Extensions.
4. In the Manage Extensions window, click **Installed** from the left navigation bar.

If the AWS Toolkit for Visual Studio is listed (as shown below), then you have it installed. In this case you can skip to section **Create AWS Lambda Project in Visual Studio**.



If you don't have it listed, install it like this:

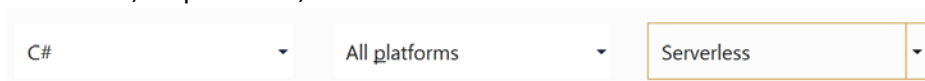
- a) Click Online on the left navigation bar.
- b) Search for AWS Toolkit. This should bring it into view.



- c) Click the **Download** button (you may need to close Visual Studio for installation to start).

Create AWS Lambda project in Visual Studio

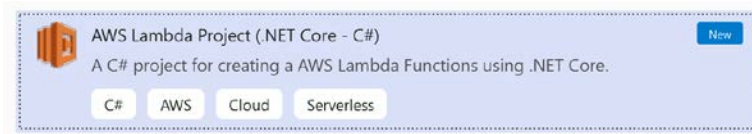
1. Create a Visual Studio solution. You can name it LambdaS3Interaction.
2. In the Solution Explorer pane, right-click on the solution and select menu **Add → New Project...**
Choose C#, All platforms, Serverless



You should see a few AWS Lambda project templates that were installed by the AWS toolkit. Also notice that because you specified All platforms, all the serverless project types listed target

.NET Core (not .NET Framework). This is because .NET Core is cross platform.

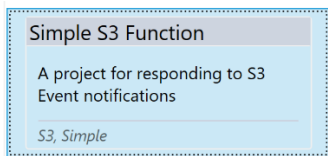
3. Choose the **AWS Lambda Project (.NET Core – C#)** template.



Click **Next**.

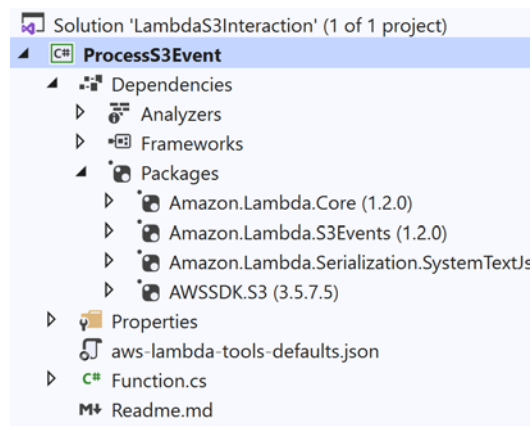
4. For project name, enter: ProcessS3Event.
Click the **Create** button.

5. In the Select Blueprint dialog, choose **Simple S3 Function**.



Click **Finish**.

6. Before we start coding, let's inspect the project structure to see what the template created and try to understand what we have and why things are the way they are.



- A few S3 and Lambda packages were added to the project.
- aws-lambda-tools-defaults.json: this file has default values that will appear in a UI wizard when you upload the function from Visual Studio to AWS. Its purpose is to make it such that you don't have to keep typing all this information every time you upload. The file itself does not play any role on the AWS side – but the values it contains do. In fact, the file itself

doesn't get included into the final DLL you build (ProcessS3Event.dll). It is simply to fill in some values in the wizard UI on the Visual Studio side.

Open this file. One important setting to look at is "function-handler":

```
"function-handler": "ProcessS3Event::ProcessS3Event.Function::FunctionHandler"
```

What this string does is tell AWS what function to call when invoking your Lambda. It should be of this format:

Assembly::Namespace.ClassName::MethodName

In our case:

Assembly: ProcessS3Event.dll
Namespace: ProcessS3Event
ClassName: Function
MethodName: FunctionHandler

You can verify the above by looking at file Function.cs. You can use a different name than FunctionHandler and adjust the value of "function-handler" accordingly.

- Read the Readme.md file.

Note that at this point you have not created a Lambda function in AWS (only in Visual Studio on your local computer). Once you deploy it, you will then see it in the AWS UI console.

7. Double-click the Function.cs file to open it. Notice the method **FunctionHandler**. This is the function that the Lambda service calls on your behalf every time your Lambda is invoked.
8. Change the code inside the try {} and add 2 lines to print the bucket and file names:

```
try
{
    var response = await this.S3Client.GetObjectMetadataAsync(s3Event.Bucket.Name, s3Event.Object.Key);

    Console.WriteLine("Bucket: {0}", s3Event.Bucket.Name);
    Console.WriteLine("File: {0}", s3Event.Object.Key);

    return response.Headers.ContentType;
}
```

9. Save the project then build it (menu Build → Rebuild Solution).

Note: Before doing the next step: If your credentials are old, close VS, update them now (replace the content of file credentials with new keys), then open VS again.

10. As mentioned above, so far you created a Lambda function skeleton in VS on your local computer. The function does not exist yet in your AWS account (if you go to the AWS console you will not see a function named ProcessS3Event yet) In this step you will upload the Lambda to AWS:

- a. In Solution Explorer, right-click on project ProcessS3Event and choose menu **Publish to AWS Lambda....** This opens a dialog titled “Upload to AWS Lambda”.
- b. Fill-in the first screen like below then press the **Next** button.

Upload to AWS Lambda

aws Upload Lambda Function
Enter the details about the function you want to upload.

AWS Credentials: Profile: default Region: US East (N. Virginia)

Package Type: Zip Lambda Runtime: .NET Core v3.1

Function Name: Process-S3-Event

Description: Responds to S3 events

Configuration: Release Framework: netcoreapp3.1

Assembly: ProcessS3Event

Type: ProcessS3Event.Function

Method: FunctionHandler

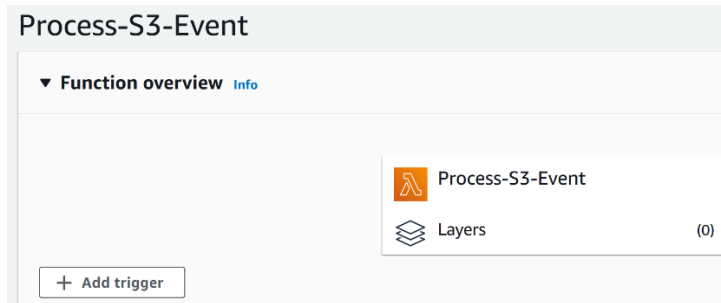
Handler: ProcessS3Event:ProcessS3Event.Function:FunctionHandler

☒ Save settings to aws-lambda-tools-defaults.json for future deployments.

Close Back Next Upload

Note that the Function Name field above (I chose to use “Process-S3-Event”) does not necessarily have to match your DLL project name in Visual Studio (ProcessS3Event). “Process-S3-Event” is what you will see in the AWS console (a logical name to identify your function – just a name). ProcessS3Event.dll is the library that the AWS Lambda engine loads before calling the function handler.

- c. In the Advanced Function Details screen, for Role Name, choose **LabRole**. Leave all other settings as they are.
 - d. Click the **Upload** button. If upload is successful the Publish screen should close.
11. Go to Canvas, start the lab, and navigate to the AWS Console. Go to the Lambda service. Verify that you have a new Lambda function named Process-S3-Event (or whatever name you gave it in the dialog above).
12. Click the Process-S3-Event function. Notice that you don’t have any triggers yet.



13. Go to one of the buckets you have in S3 (if you don't have any create one) and add any dummy file to it.

14. Come back to the Lambda service and click the Process-S3-Event. Click the **Monitoring** tab, then the **View logs in CloudWatch** button. You will notice that you don't have any logs and your Lambda wasn't invoked.

This is the correct behavior because we haven't added any trigger to the Lambda function. In Module 9 we invoked the simple-example-function Lambda function by manually clicking a Test button to simulate passing data to it. In this case we are interested in a different scenario: an external event (in this case an S3 one) triggering a notification that invokes our function. Therefore, we first need to define this trigger.

15. Go back to the Process-S3-Event function. Click the **Add trigger** button.

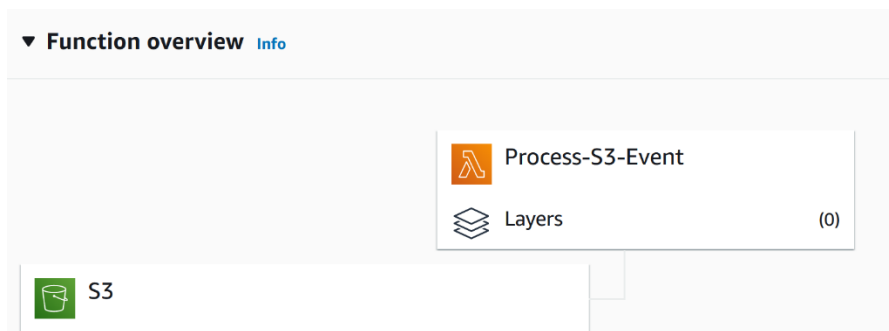
16. From the Select a trigger dropdown, choose S3.
From the Bucket dropdown, select the bucket you have.
From the Event Type dropdown, choose PUT.

Leave other settings unchanged.

Click the checkbox in the Recursive invocation warning area.

Click the **Add** button.

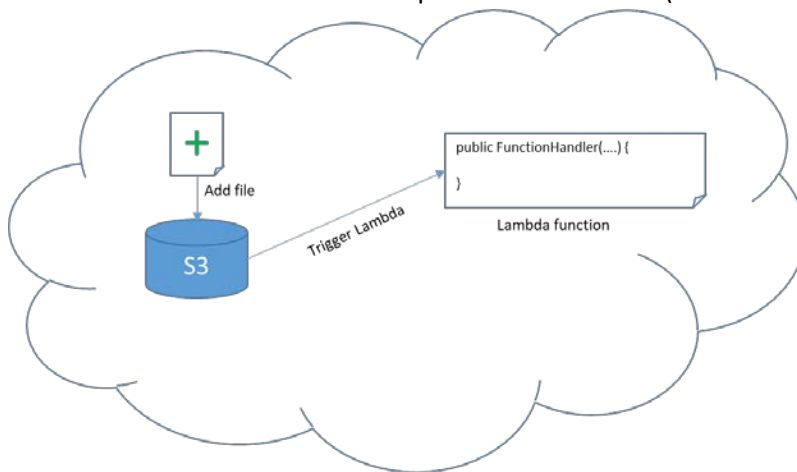
17. You should see that you have created an S3 trigger.



18. Go back to the S3 service and delete the dummy file you added in step 13.
19. Add the dummy file back to the S3 bucket.
20. Go to the Lambda service, click the Process-S3-Event function, and view the logs in CloudWatch (by clicking the **Monitor** tab then button **View logs in CloudWatch**). You should now see that some logging happened indicating that the function was called. You should also see the output of the two Console.WriteLine statements you added in C#.

```
▶ 2021-05-03T17:32:44.947-07:00      Bucket: alfred-cs455-bucket2
▶ 2021-05-03T17:32:44.947-07:00      File: todo.txt
```

What happened is that you configured the addition of a file to a bucket (PUT) via the trigger you defined to automatically invoke the Lambda function. You now have code that detects changes in another service and can do computation based on it (without having to do any polling).



21. This triggering mechanism (invoking Lambda when some event happen) is widely used in different cloud applications scenarios and use cases. Not only from S3 but from many other services.

Go back to Process-S3-Event function, and click the Add trigger button again (you will not add another trigger – we are doing this to inspect what other events can cause an invocation to Lambda).

Open the Select a trigger dropdown and inspect the different events that you can configure to trigger a Lambda function. You can see things like DynamoDB, IoT, etc.

One interesting one is the **API Gateway**. This allows you to turn your Lambda function into a **web service** without having to do any coding other than code the logic of your function.

We will use API Gateway later in the course to turn a Lambda into a web service endpoint.

You can dismiss the Add trigger page. No need to add a new trigger.

Exercises to Complete and Practice With


1. Enclosed with this module are 3 files. Use these files to test your function for (a), (b), and (c) below.

Good.xml: A file with well-formed XML.
Bad.xml: A file with XML that is not well-formed.
Empty.xml: An empty file.

Modify the function you created in this module to do the following:

When a file is added to an S3 bucket, read its content and print the following:

- a) If the file has well-formed XML, print its content to CloudWatch. Your output should look like this:



A screenshot of a CloudWatch log entry. The log shows a timestamp of 03:29:07 and the message: <customer><id>11223344</id></customer>

- b) If the file has XML that is not well-formed, print to CloudWatch the XML error in the file. Your output should look like this:



A screenshot of a CloudWatch log entry. The log shows a timestamp of 03:30:52 and the message: Invalid content: Unexpected end of file has occurred. The following elements are not closed: customer, customer. Line 3, position 11.

NOTE: Do not hardcode the above sentence in your program. The XML content can have a different problem and your function should identify that. HINT: use a try {} catch(XmlException ex) to load the xml content into an XmlDocument object. If the loading fails, the XmlException will have the details of the offending content – that is, why the XML is not well-formed.

- c) If the file is empty, print to CloudWatch that it is. Your output should look like this:



A screenshot of a CloudWatch log entry. The log shows a timestamp of 03:31:43 and the message: File is empty

Things you might want to use in your code:

```
using Amazon.S3.Model;  
using System.IO;  
using System.Xml;
```

2. Read this AWS [case study](#) (one of the quiz questions is about this case study)

What to submit:

Complete quiz **Module10-Quiz** after you finish the module.