Module 15
Simple Queue Service (also known as SQS)

Overview

Let's look at the services we learned about so far and where we are going next. Under the *Compute*, *Storage*, and *Database* family of services, we covered the most widely used ones: EC2, Lambda, S3, Dynamo DB and RDS (Figure 1). Note that RDS is a catch-all term for many relational databases. Although we experimented with PostgreSQL, we could have launched a different database engine (e.g., MySQL, Oracle, SQL Server, etc.). All these relational databases are grouped under RDS.

As I said at the beginning of the class, we don't have time to cover every single AWS service – there are too many to fit in a quarter course. But we are covering the most widely used ones that you are likely to encounter at work or for your own cloud projects. For example, under the *Compute* family, you are likely to use Lambda and EC2 much more than the other services.

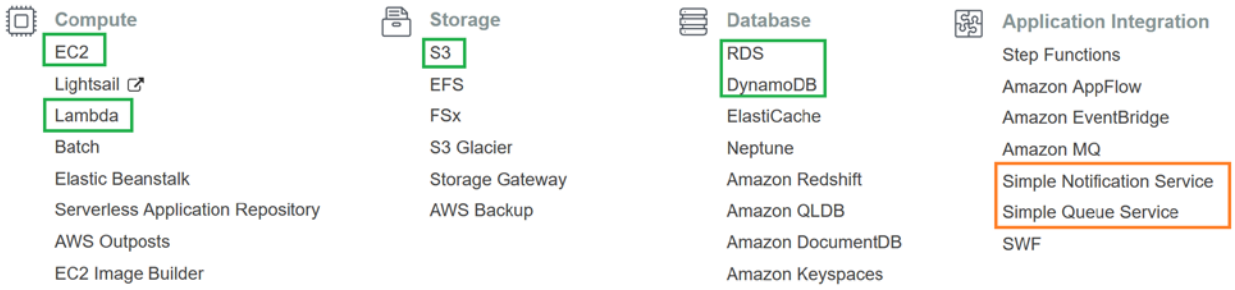


Figure 1: Services we covered so far are highlighted in green. In this module and the next we will cover 2 services under the Application Integration family of services.

In this module and the next we will look at a new family of services, *Application Integration*. In this module we cover **Simple Queue Service** (**SQS**). A cloud queue is an essential cloud resource that you are likely to find and use in many cloud applications. Many AWS services also use AWS queues in how they operate.

In a standard local application (i.e., not a cloud one), you typically use a queue as a temporary in-memory storage to decouple two components in an application. For example, in a producer-consumer scenario, the producer puts a message in the queue. When the consumer is ready to process the message, it de-queues the message from the queue and processes it (Figure 2).

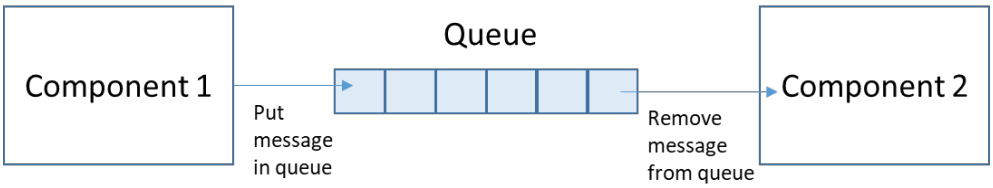


Figure 2: A queue between 2 components. Component 1 is the producer. It puts some request message in the queue. Component 2 is the consumer. When it is ready to process a message, it de-queues it from the queue and acts on it.

There are different types of queues that you can review in a Data Structures book. One simple variation is a First in First out (FIFO) queue. In a way a FIFO queue provides the following benefits:

- It decouples Component 1 and Component 2.
- It enforces some order on message processing: if Component 1 puts message A before message B, Component 2 processes message A before it processes message B. This might be desirable if message A and message B have the same priority. If they don't have the same priority, then FIFO is not a suitable choice, and a priority-based queue is a better choice if one is available.

An SQS cloud queue works somewhat similar to Figure 2. In this case Component 1 and Component 2 can be two cloud services or a mix of a cloud service and a non-cloud component. We will look at this latter case which is interesting and allow solving a particular type of problem. The other great advantage is that the decoupling allows different services to process messages at different rates. For example, consider the setup of Figure 3 where there is a downstream service (represented in the picture as two gears) that our application instances are using. Further assume that spikes in the utilization of the application instances overwhelm this service.

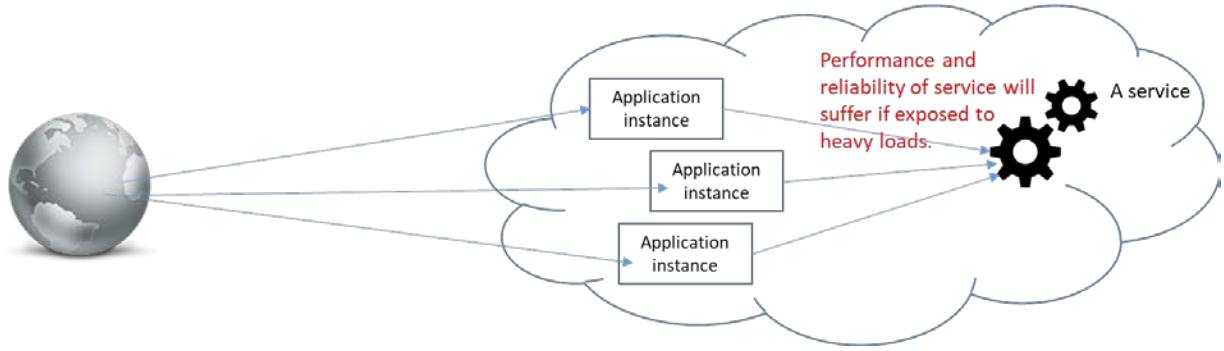


Figure 3: Application instances making use of a downstream service.

One way to mitigate this problem is to refactor the design and introduce a queue as a buffer between the application instances and the service as shown in Figure 4. In this way the service can process

requests at its own pace. And the application instances become more responsive to users since they no longer have to block until a message is processed. They simply deposit a request in the queue and become freely available to answer other incoming requests. This is a common pattern known as **Queue-Based Load Leveling** that you can read about it here.

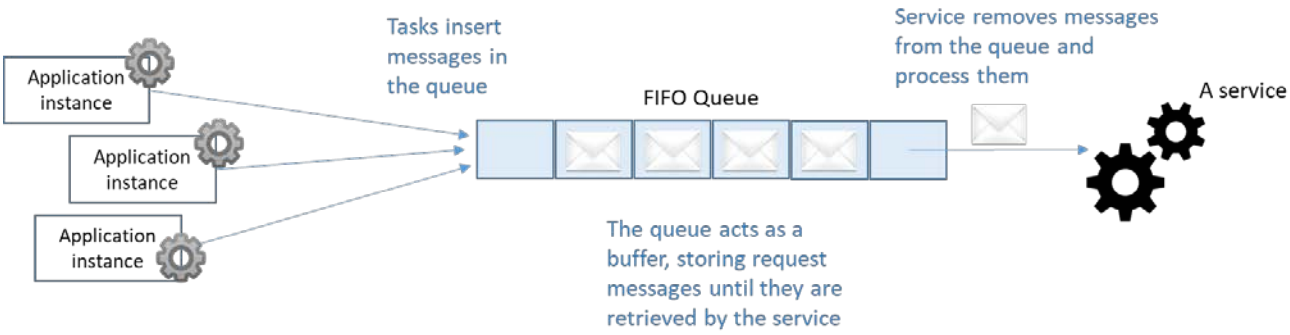


Figure 4: Using a queue as a buffer between two components of an application to load-balance requests and keep application instances responsive.

Read the Basic SQS Architecture here.

Objective

In this module, you will do 2 projects:

The objective of the first is to create a queue, put messages in it, and have a Lambda function listen to whenever a message arrives to the queues. It is similar to how Lambda can listen to files uploaded to S3 or messages coming from Alexa. In this case the event source is data in a cloud queue instead of a file uploaded to S3.

The objective of the second part is to somewhat reverse the order of functionalities of the first part. In the second part you will write a Lambda function that adds messages to a queue (that is, the Lambda is putting the messages instead of reading them), and a standard application that continuously polls the queue to read messages from it. This is a common pattern used to communicate data from a cloud environment to a local environment inside a corporate network.

Create an SQS Queue

1. Sign in to the AWS Management Console and go to the SQS service.
2. Click the **Create queue** button.
3. For the queue type, choose the **FIFO** queue.

4. For the queue name, enter something like: TestQueue.fifo

5. Read what's under the Configuration section. But leave the default values.

6. Leave everything else as-is.

7. Click the **Create queue** button.
   You should now see the queue listed as a queue that your account owns.

8. Look at the **Details** info. You will later use some of this information.

   Click the **Lambda Triggers** tab. Notice that you can have messages in a queue trigger a Lambda function (similar to how an S3 file operation can trigger a Lambda).

   Click the **Encryption** tab. Messages in a queue can be encrypted. However, we will not use this advanced feature.

9. Click button **Send and receive messages**.

10. For the Message body, enter any string you want. For example:

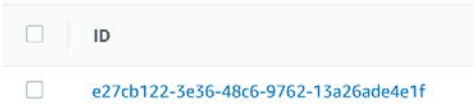
    ```
    <customer><id>223344</id></customer>
    ```

    You don't necessarily need to enter XML or JSON. However, since you want to be able to parse your data, XML and JSON are commonly used (and we learned how to parse them).

11. For Message group ID, enter: group1
12. For Message duplication ID, enter: customerIDMessage

13. Click the **Send message** button.

14. Now if you look under the **Receive messages** section, you should see that you have 1 message.
15. Click the **Poll for messages** button. The message should appear under **Messages**.
16. Click the message ID link under **Messages**:



    Notice the data you entered earlier. Click the **Done** button.

17. Click the **Queues** link:

18. Select the radio button next to TestQueue.fifo, then click the **Delete** button. Confirm deletion.

19. Click the **Create queue** button again. But this time choose **Standard Queue** (not FIFO Queue). Name it: InputQueue

20. Click the **Create queue** button.

    This InputQueue queue will be used in the remaining parts of this module.

Create Lambda that Listens to Messages In a Queue.

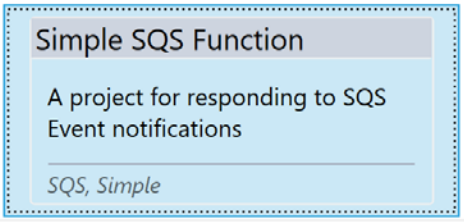
In this section you will create a Lambda function that creates messages and sends them to the InputQueue.

1. Create a Visual Studio solution called QueueSolution.

2. Under the solution create a project of type "AWS Lambda Project (.NET Core – C#)".

   Give the project the name: ProcessQueueMessagesFunction. Then click the **Create** button.

   In the Select Blueprint dialog, select the **Simple SQS Function** project template. Then click the **Finish** button.



   There is no need to change anything in the template project that was created. The code in Function.cs prints the queue message to the log as shown below:

```
context.Logger.LogLine($"Processed message {message.Body}");
```

   This is enough for us to check that the Lambda function was able to process a message that you will put in the queue. Click **File → Save All**.

If you credential keys are stale, close Visual Studio, renew them, and then open Visual Studio.

3. Build the project. Then right-click the project and select menu **Publish to AWS Lambda…**

4. Choose the default profile and US-East-1 region.
   For Function Name, enter: ProcessQueueMessagesFunction

   Click the **Next** button.

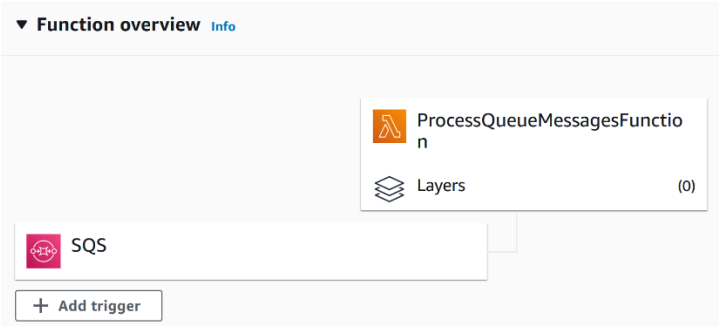
5. For Role Name, choose: LabRole

6. Click the **Upload** button.

7. After uploading of the function is done, login to the AWS Management Console and go to the Lambda Service.

8. Click the ProcessQueueMessagesFunction.

9. Click the **Add trigger** button.
   From the **Triggers** list, choose **SQS**.
   In the SQS Queue textbox, choose the InputQueue you created earlier.
   Make sure the **Enable trigger** checkbox is checked.
   Click the **Add** button.



   What you did in this step is configure the event source that will trigger your ProcessQueueMessagesFunction Lambda function (similar to how you did it with S3 in previous modules and Project 1). You should appreciate now how versatile Lambda is and how it can tie many things together. That's one of the reasons why Lambda is a widely used service.

10. Go to the Simple Queue Service.

11. Click the radio button to the left of InputQueue and click button **Send and receive messages**.

12. Enter a simple message like the below, then click the **Send message** button.

```
{ "id" : "223344" }
```

13. Go back to the Lambda service, click the ProcessQueueMessagesFunction function, and go to the Cloud Watch logs.

14. Verify that you see in the log the message you entered:

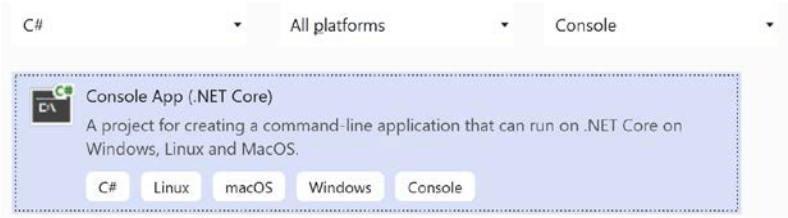
```
Processed message { "id": "223344" }
```

This shows that the Lambda function successfully listened to a message that was placed in the queue. In this case you placed the message manually from the AWS console. But this message could have come from another system in a larger application (the more realistic case).

Exercise to Do and Submit

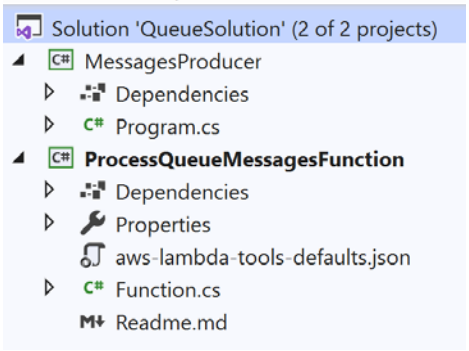
We said in the comment above that a more realistic scenario is that some application component put messages in a queue and Lambda acts on them. You will demonstrate this in this exercise.

1. In the same Visual Studio QueueSolution you created earlier, create a project of type **Console App (.NET Core)** (you do that by right-clicking on the solution and choosing menu **Add → New Project…**).



Give the project the name **MessagesProducer**.
Click the **Create** button to create the project.

Your solution should now have 2 projects: MessagesProducer (a console application) and ProcessQueueMessagesFunction (a Lambda function).



Right-click on MessagesProducer and choose menu **Set as Startup Project**.

NOTE: In Visual Studio, a solution can have multiple projects (as shown above). When it is the case, Visual Studio needs to know which project it should start when you click the run button (or press F5). The menu **Set as Startup Project** is just doing that: you are telling Visual Studio which project you want to run when you press the run button. In this case we are telling Visual Studio to run the MessagesProducer console application.

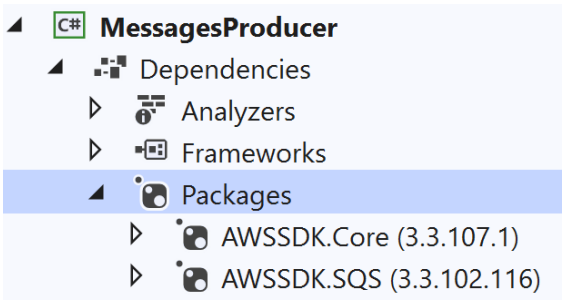
In a solution, you can always tell which project is the startup project by looking at the project names. The one that appears in Bold is the startup project.

2. In the MessagesProducer project, right-click on **Dependencies** and choose menu **Manage NuGet Packages…**.

3. While the Browse tab is selected, search for and install the following 2 packages:

   AWSSDK.Core
   AWSSDK.SQS

   When done you should have the above listed under Packages:



4. Complete the code in Program.cs to do the following:

a. Send an arbitrary message to your InputQueue.
b. When done print the following to the console (Figure 5):

Figure 5

c. Go to your ProcessQueueMessagesFunction Lambda function in the AWS console and verify that the arbitrary message you sent from project MessagesProducer was read by your Lambda and was printed to CloudWatch (Figure 6):

```
Processed message { "item":"laptop", "purchase-amount":"725.00" }
```

Figure 6

Implementation hints:

You might need to add the following using statements:

```
using Amazon;
using Amazon.Runtime;
using Amazon.Runtime.CredentialManagement;
using Amazon.SQS;
using Amazon.SQS.Model;
```

You might find the use of these classes/methods helpful:

```
AmazonSQSClient           // class
SendMessageRequest        // class
SendMessageResponse       // class
SendMessageAsync          // method
```

You can create a `SendMessageRequest` like this (where `InputQueueUrl` is a string variable that holds the URL of your queue that you can get from the Details tab of your queue in the AWS Console):

```
string message = "{ \"item\":\"laptop\", \"purchase-amount\":\"725.00\" }";
SendMessageRequest request = new SendMessageRequest
{
    QueueUrl = InputQueueUrl,
    MessageBody = message
};
```

What to Submit:
Nothing to submit for this module.