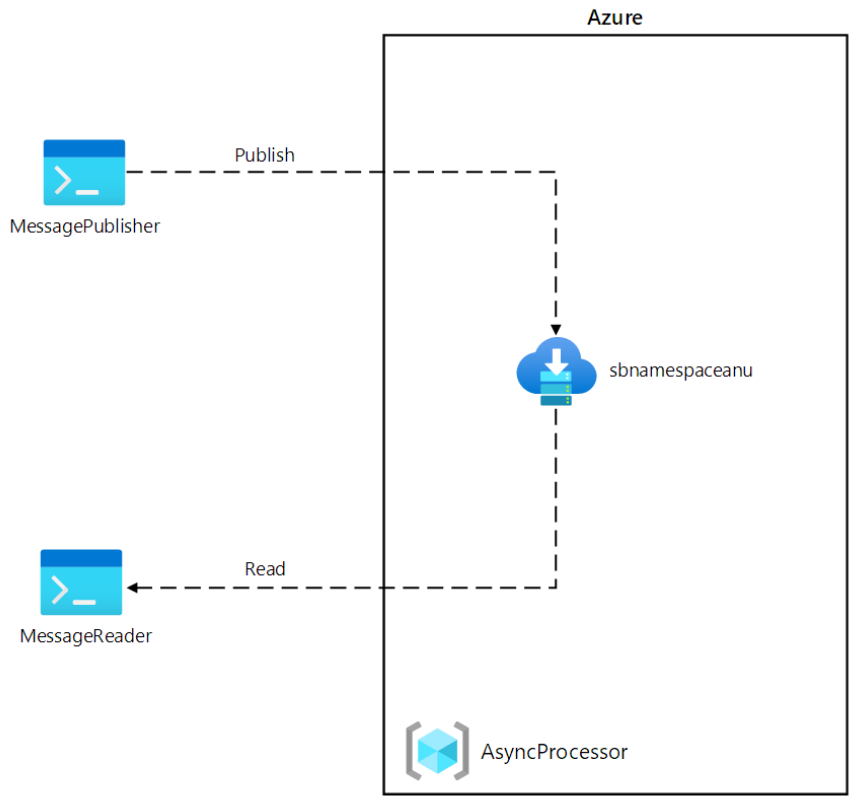
Lab 01 -Asynchronously process messages by using Azure Service Bus Queues

Architecture diagram



Exercise 1: Create Azure resources

Task 1: Open the Azure portal

1. On the taskbar, select the **Microsoft Edge** icon.
2. In the browser window, browse to the Azure portal (portal.azure.com) and sign in with the account you'll be using for this lab.

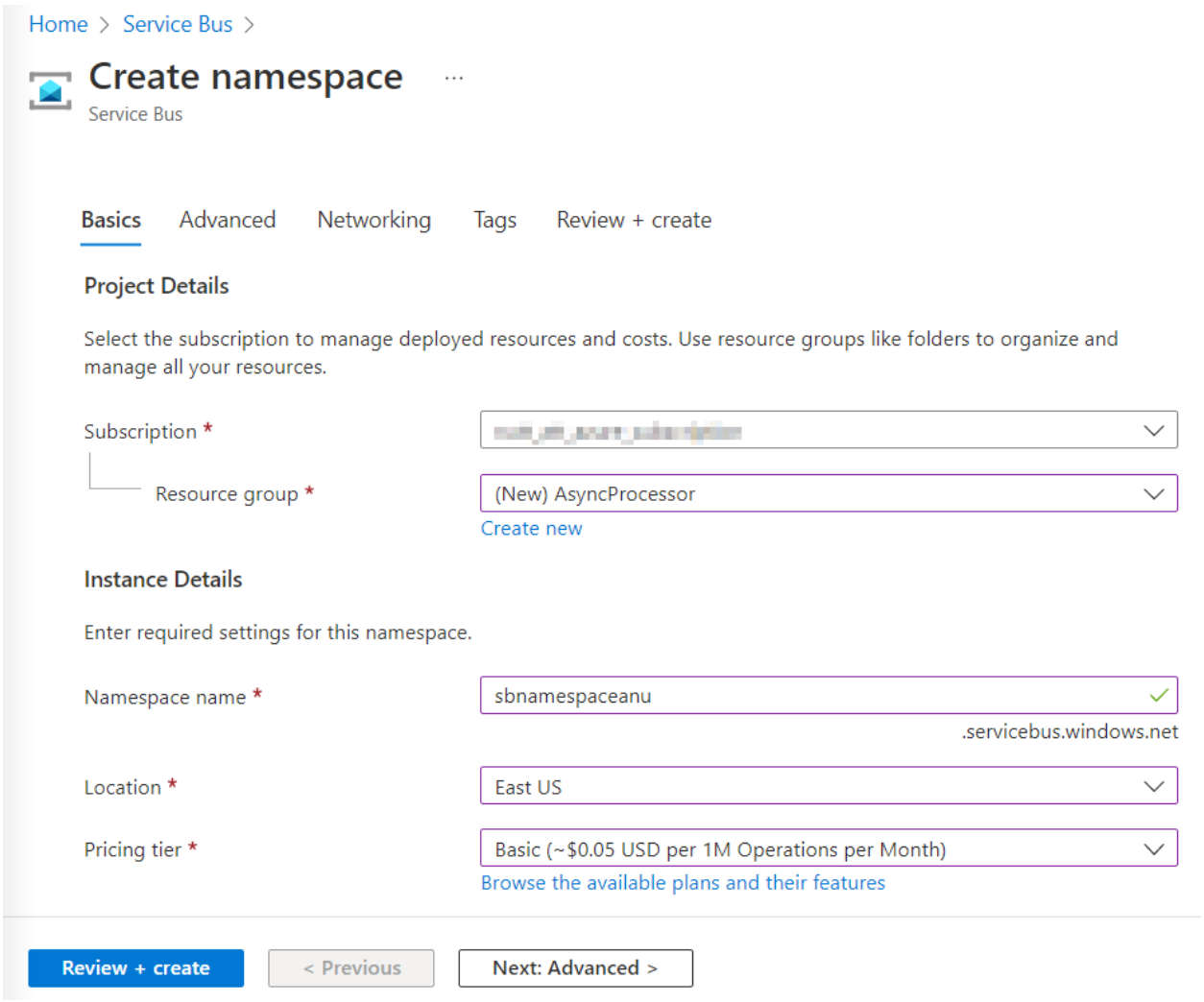
   Note: If this is your first time signing in to the Azure portal, you'll be offered a tour of the portal. Select **Get Started** to skip the tour and begin using the portal.

Task 2: Create an Azure Service Bus queue

1. In the Azure portal, use the **Search resources, services, and docs** text box to search for **Service Bus** and then, in the list of results, select **Service Bus**.
2. On the **Service Bus** blade, select **+ Create**.
3. On the **Create namespace** blade, on the **Basics** tab, perform the following actions, and select **Review + create**:

Setting	Action
Subscription drop-down list	Retain the default value
Resource group section	Select **Create new**, enter **AsyncProcessor**, and then select **OK**
Namespace name text box	Enter **sbnamespace***[yourname]*
Region drop-down list	Select any Azure region in which you can deploy an Azure Service Bus
Pricing tier drop-down list	Select **Basic**

4. The following screenshot displays the configured settings on the **Basics** tab on the **Create namespace** blade.

5. 

6. On the **Review + create** tab, review the options that you selected during the previous steps.
7. Select **Create** to create the **Service Bus** namespace by using your specified configuration.

   Note: Wait for the creation task to complete before you proceed with this lab.

8. On the **Deployment** blade, select the **Go to resource** button to navigate to the blade of the newly created **Service Bus** namespace.
9. On the **Service Bus** namespace blade, in the **Settings** section, select **Shared access policies**.
10. In the list of policies, select **RootManageSharedAccessKey**.
11. On the **SAS Policy: RootManageSharedAccessKey** pane, next to the **Primary Connection String** entry, select the **Copy to clipboard** button, and record the copied value. You'll use it later in this lab.

    Note: It doesn't matter which of the two available keys you choose. They are interchangeable.

12. On the **Service Bus** namespace blade, in the **Entities** section, select **Queues**, and then select **+ Queue**.
13. On the **Create queue** blade, review the available settings, in the **Name** text box, enter **messagequeue**, and then select **Create**.
14. Select **messagequeue** to display the properties of the **Service Bus** queue.
15. Leave the browser window open. You'll use it again later in this lab.

Review

In this exercise, you created an Azure **Service Bus** namespace and a **Service Bus** queue that you'll use through the remainder of the lab.

Exercise 2: Create a .NET Core project to publish messages to a Service Bus queue

Task 1: Create a .NET Core project

1. From the lab computer, start Visual Studio Code.
2. In Visual Studio Code, in the **File** menu, select **Open Folder**.
3. In the **Open Folder** window, browse to **Allfiles (F):\Allfiles\Labs\10\Starter\MessagePublisher**, and then select **Select Folder**.
4. In the **Visual Studio Code** window, activate the shortcut menu, and then select **Open in Integrated Terminal**.
5. At the terminal prompt, run the following command to create a new .NET project named **MessagePublisher** in the current folder:

```
dotnet new console --framework net6.0 --name MessagePublisher --output .
```

> **Note**: The **dotnet new** command will create a new **console** project in a folder with the same name as the project.

6. Run the following command to import version 7.8.1 of the **Azure.Messaging.ServiceBus** package from NuGet:

```
dotnet add package Azure.Messaging.ServiceBus --version 7.8.1
```

> **Note**: The **dotnet add package** command will add the **Azure.Messaging.ServiceBus** package from NuGet. For more information, go to Azure.Messaging.ServiceBus.

7. At the terminal prompt, run the following command to build the .NET Core console application:

```
dotnet build
```

8. Select **Kill Terminal** (the **Recycle Bin** icon) to close the terminal pane and any associated processes.

Task 2: Publish messages to an Azure Service Bus queue

1. In the **Explorer** pane of the **Visual Studio Code** window, open the **Program.cs** file.
2. On the code editor tab for the **Program.cs** file, delete all the code in the existing file.
3. Add the following lines of code to facilitate the use of the built-in namespaces that will be referenced in this file:

```
using System;
using System.Threading.Tasks;
```

4. Add the following code to import the **Azure.Messaging.ServiceBus** namespace included in the **Azure.Storage.Queues** package imported from NuGet:

```
using Azure.Messaging.ServiceBus;
```

5. Enter the following code to create a new **Program** class in the MessagePublisher namespace:

```
namespace MessagePublisher
{
   public class Program
   {
   }
}
```

6. In the **Program** class, enter the following code to create a string constant named **storageConnectionString**:

```
private const string storageConnectionString = "";
```

7. Update the **storageConnectionString** string constant by setting its value to **Primary Connection String** of the Service Bus namespace you recorded earlier in this lab.
8. Enter the following code to create a string constant named **queueName** with a value of **messagequeue**, matching the name of the Service Bus queue you created earlier in this exercise.

```
private const string queueName = "messagequeue";
```

9. Enter the following code to create an integer constant which stores the number of messages to be sent to the target queue:

```
private const int numOfMessages = 3;
```

10. Enter the following code to create a Service Bus client that will own the connection to the target queue:

```
static ServiceBusClient client;
```

11. Enter the following code to create a Service Bus sender that will be used to publish messages to the target queue:

```
static ServiceBusSender sender;
```

12. Enter the following code to create an asynchronous **Main** method:

```
public static async Task Main(string[] args)
{
}
```

13. Review the **Program.cs** file, which should now include the following code. Note that the `<storage-connection-string>` placeholder represents the connection string to the target Azure Service Bus namespace:

```
using System;
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;
namespace MessagePublisher
{
    public class Program
    {
```

```
        private const string storageConnectionString = "<storage-connection-
string>";
        private const string queueName = "messagequeue";
        private const int numOfMessages = 3;
        static ServiceBusClient client = default!;
        static ServiceBusSender sender = default!;
        public static async Task Main(string[] args)
        {
        }
    }
}
```

14. In the **Main** method, add the following code to initialize *client* of type **ServiceBusClient** that will provide connectivity to the Service Bus namespace and **sender** that will be responsible for sending messages:

```
client = new ServiceBusClient(storageConnectionString);
sender = client.CreateSender(queueName);
```

> **Note**: The Service Bus client is safe to cache and use as a singleton for the lifetime of the application. This is considered one of the best practices when publishing and reading messages on a regular basis.

15. In the **Main** method, add the following code to create a **ServiceBusMessageBatch** object that will allow you to combine multiple messages into a batch by using the **TryAddMessage** method:

```
using ServiceBusMessageBatch messageBatch = await
sender.CreateMessageBatchAsync();
```

16. In the **Main** method, add the following lines of code to add messages to a batch and throw an exception if a message size exceeds the limits supported by the batch:

```
for (int i = 1; i <= numOfMessages; i++)
{
    if (!messageBatch.TryAddMessage(new ServiceBusMessage($"Message {i}")))
    {
        throw new Exception($"The message {i} is too large to fit in the
batch.");
    }
}
```

17. In the **Main** method, add the following lines of code to create a try block, with **sender** asynchronously publishing messages in the batch to the target queue:

```
try
{
    await sender.SendMessagesAsync(messageBatch);
    Console.WriteLine($"A batch of {numOfMessages} messages has been published
to the queue.");
}
```

18. In the **Main** method, add the following lines of code to create a finally block that asynchronously disposes of the **sender** and **client** objects, releasing any network and unmanaged resources:

```
finally
{
    await sender.DisposeAsync();
    await client.DisposeAsync();
}
```

19. Review the **Main** method, which should now consist of the following code:

```
public static async Task Main(string[] args)
{
    client = new ServiceBusClient(storageConnectionString);
    sender = client.CreateSender(queueName);
    using ServiceBusMessageBatch messageBatch = await
sender.CreateMessageBatchAsync();
    for (int i = 1; i <= numOfMessages; i++)
    {
        if (!messageBatch.TryAddMessage(new ServiceBusMessage($"Message
{i}")))
        {
            throw new Exception($"The message {i} is too large to fit in the
batch.");
        }
    }
    try
    {
        await sender.SendMessagesAsync(messageBatch);
        Console.WriteLine($"A batch of {numOfMessages} messages has been
published to the queue.");
    }
    finally
    {
        await sender.DisposeAsync();
        await client.DisposeAsync();
    }
}
```

20. Save the **Program.cs** file.

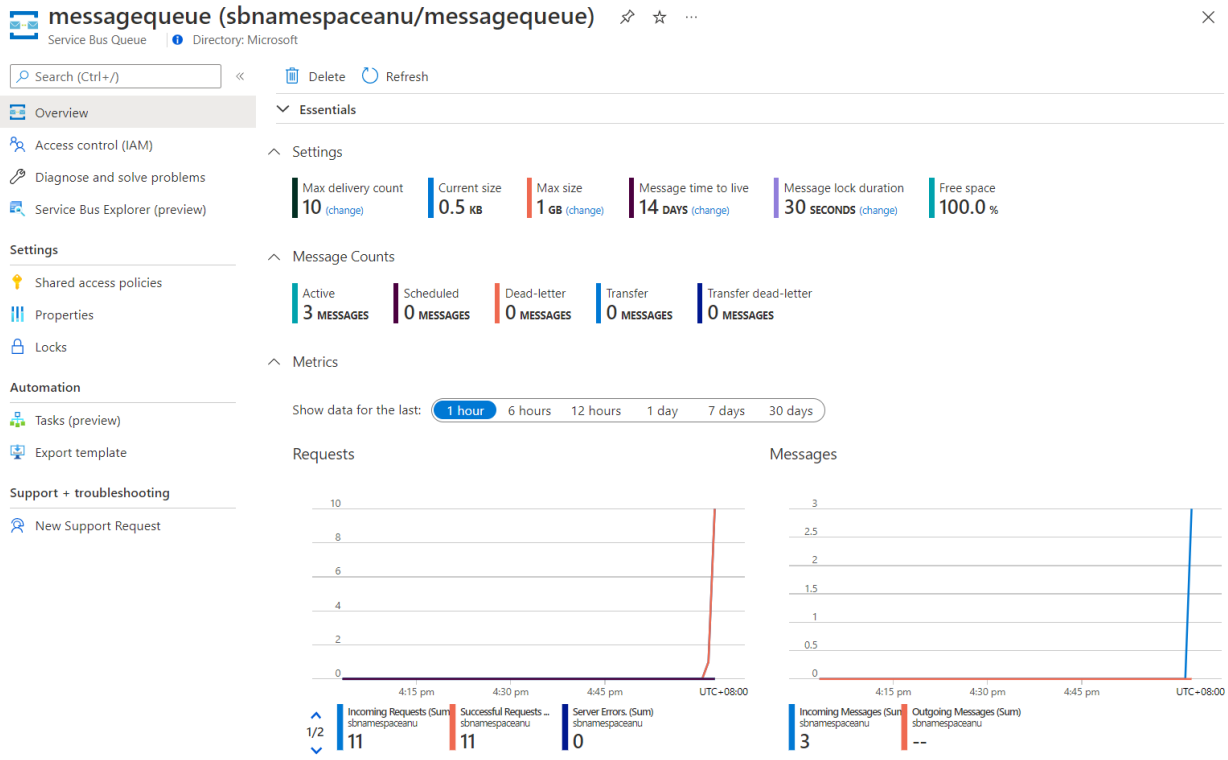
21. In the **Visual Studio Code** window, activate the shortcut menu, and then select **Open in Integrated Terminal**.
22. At the terminal prompt, run the following command to launch the .NET Core console app:

```
dotnet run
```

> **Note**: If you encounter any errors, review the **Program.cs** file in the **Allfiles (F):\Allfiles\Labs\10\Solution\MessagePublisher** folder.

23. Verify that the console message displayed at the terminal prompt states that a batch of three messages has been published to queue.
24. Select **Kill Terminal** (the **Recycle Bin** icon) to close the terminal pane and any associated processes.
25. Switch to the Microsoft Edge browser displaying the Service Bus queue **messagequeue** in the Azure portal.
26. Review the **Essentials** pane and note that the queue contains three active messages.

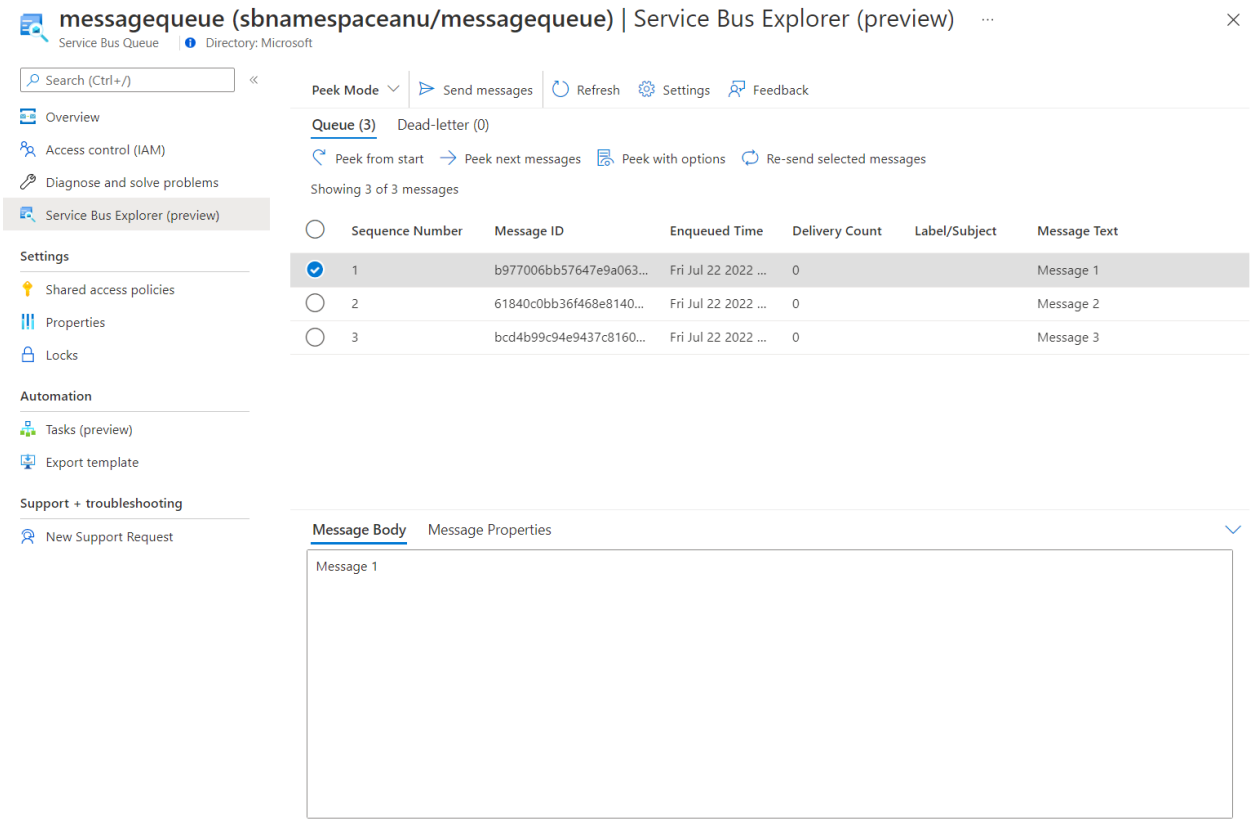
    The following screenshot displays the Service Bus queue metrics and message count.



27. Select **Service Bus Explorer (preview)** blade.

28. On the **Peek Mode** tab header and, on the **Queue** tab, select the **Peek from start** button.
29. Verify that the queue contains three messages.
30. Select the first message and review its content in the **Message** pane.

    The following screenshot displays the first message's content.



31. Close the **Message** pane.

Review

In this exercise, you configured your .NET project that published messages into an Azure Service Bus queue.

Exercise 3: Create a .NET Core project to read messages from a Service Bus queue

Task 1: Create a .NET project

1. From the lab computer, start Visual Studio Code.
2. In Visual Studio Code, in the **File** menu, select **Open Folder**.
3. In the **Open Folder** window, browse to **Allfiles (F):\Allfiles\Labs\10\Starter\MessageReader**, and then select **Select Folder**.
4. In the **Visual Studio Code** window, activate the shortcut menu, and then select **Open in Integrated Terminal**.
5. At the terminal prompt, run the following command to create a new .NET project named **MessageReader** in the current folder:

```
dotnet new console --framework net6.0 --name MessageReader --output .
```

6. Run the following command to import version 7.8.1 of the **Azure.Messaging.ServiceBus** package from NuGet:

```
dotnet add package Azure.Messaging.ServiceBus --version 7.8.1
```

7. At the terminal prompt, run the following command to build the .NET Core console application:

```
dotnet build
```

8. Select **Kill Terminal** (the **Recycle Bin** icon) to close the terminal pane and any associated processes.

Task 2: Read messages from an Azure Service Bus queue

1. In the **Explorer** pane of the **Visual Studio Code** window, open the **Program.cs** file.
2. On the code editor tab for the **Program.cs** file, delete all the code in the existing file.
3. Add the same code which was included in the Program.cs file to allow for interaction with Azure Service Bus queues, but set the namespace to **MessageReader**:

```
using System;
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;
namespace MessageReader
{
   public class Program
   {
```

```
        private const string storageConnectionString = "";
        static string queueName = "messagequeue";
        static ServiceBusClient client = default!;
    }
}
```

4. As before, update the **storageConnectionString** string constant by setting its value to **Primary Connection String** of the **Service Bus** namespace you recorded earlier in this lab.
5. Enter the following code to create a ServiceBusProcessor that will be used to process messages from the queue:

```
static ServiceBusProcessor processor = default!;
```

6. Enter the following code to create a static async **MessageHandler** task that displays the body of messages in the queue as they are being processed and deletes them after the processing completes:

```
static async Task MessageHandler(ProcessMessageEventArgs args)
{
    string body = args.Message.Body.ToString();
    Console.WriteLine($"Received: {body}");
    await args.CompleteMessageAsync(args.Message);
}
```

7. Enter the following code to create a static async **ErrorHandler** task that manages any exceptions encountered during message processing:

```
static Task ErrorHandler(ProcessErrorEventArgs args)
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
}
```

8. Enter the following code to create an asynchronous **Main** method:

```
static async Task Main(string[] args)
{
}
```

9. Review the **Program.cs** file, which should now include the following code. The `<storage-connection-string>` placeholder represents the connection string to the target Azure Service Bus namespace:

```
using System;
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;
namespace MessageReader
{
    class Program
    {
        static string storageConnectionString = "<storage-connection-
string>`";
        static string queueName = "messagequeue";
        static ServiceBusClient client;
        static ServiceBusProcessor processor;
        static async Task MessageHandler(ProcessMessageEventArgs args)
        {
            string body = args.Message.Body.ToString();
            Console.WriteLine($"Received: {body}");
            await args.CompleteMessageAsync(args.Message);
        }
        static Task ErrorHandler(ProcessErrorEventArgs args)
        {
            Console.WriteLine(args.Exception.ToString());
            return Task.CompletedTask;
        }
        static async Task Main()
        {
        }
    }
}
```

10. In the **Main** method, add the following code to initialize *client* of type **ServiceBusClient** that will provide connectivity to the Service Bus namespace and **processor** that will be responsible for processing of messages:

```
client = new ServiceBusClient(storageConnectionString);
processor = client.CreateProcessor(queueName, new
ServiceBusProcessorOptions());
```

> **Note**: As mentioned earlier, the Service Bus client is safe to cache and use as a singleton for the lifetime of the application. This is considered one of the best practices when publishing and reading messages on a regular basis.

11. In the **Main** method, add the following lines of code to create a try block, which first implements a message and error processing handler, initiates message processing, and stops processing following a user input:

```
try
{
    processor.ProcessMessageAsync += MessageHandler;
    processor.ProcessErrorAsync += ErrorHandler;
```

```
    await processor.StartProcessingAsync();
    Console.WriteLine("Wait for a minute and then press any key to end the
processing");
    Console.ReadKey();
    Console.WriteLine("\nStopping the receiver...");
    await processor.StopProcessingAsync();
    Console.WriteLine("Stopped receiving messages");
}
```

12. In the **Main** method, add the following lines of code to create a finally block that asynchronously disposes of the **processor** and **client** objects, releasing any network and unmanaged resources:

```
finally
{
    await processor.DisposeAsync();
    await client.DisposeAsync();
}
```

13. Review the **Main** method, which should now consist of the following code:

```
static async Task Main()
{
    client = new ServiceBusClient(storageConnectionString);
    processor = client.CreateProcessor(queueName, new
ServiceBusProcessorOptions());
    try
    {
        processor.ProcessMessageAsync += MessageHandler;
        processor.ProcessErrorAsync += ErrorHandler;

        await processor.StartProcessingAsync();
        Console.WriteLine("Wait for a minute and then press any key to end the
processing");
        Console.ReadKey();

        Console.WriteLine("\nStopping the receiver...");
        await processor.StopProcessingAsync();
        Console.WriteLine("Stopped receiving messages");
    }
    finally
    {
        await processor.DisposeAsync();
        await client.DisposeAsync();
    }
}
```

14. Save the **Program.cs** file.

15. In the **Visual Studio Code** window, activate the shortcut menu, and then select **Open in Integrated Terminal**.
16. At the terminal prompt, run the following command to launch the .NET Core console app:

```
dotnet run
```

> **Note**: If you encounter any errors, review the **Program.cs** file in the **Allfiles (F):\Allfiles\Labs\10\Solution\MessageReader** folder.

17. Verify that the console message displayed at the terminal prompt states that each of the three messages in the queue has been received.
18. At the terminal prompt, press any key to stop the receiver and terminate the app execution.
19. Select **Kill Terminal** (the **Recycle Bin** icon) to close the terminal pane and any associated processes.
20. Switch back to the Microsoft Edge browser displaying the Service Bus queue **messagequeue** in the Azure portal.
21. On the **Service Bus Explorer (preview)** blade, select **Peek from start**, and note that the number of active messages in the queue has changed to **0**.