McKesson Azure Lab 08: Demo3

Joan Imrich

>>

Demo3: Azure Service Bus Queues **November 30, 2017**

>>

.NET multi-tier application using Azure Service Bus Queues

REF: https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dotnet-multi-tier-app-using-service-bus-queues
REF: https://docs.microsoft.com/en-us/azure/storage/common/storage-use-emulator

Demo 3 walks you through the steps to create an application that uses multiple Azure resources running in your local environment. **Azure Service Bus Queues implemented with .NET multi-tier application SDK via Visual Studio**

You will learn the following:

- How to enable your computer for Azure development with a single download and install.
- How to use Visual Studio to develop for Azure.
- How to create a multi-tier application in Azure using web and worker roles.
- How to communicate between tiers using Service Bus queues.

In this tutorial you'll build and run the multi-tier application in an Azure cloud service. The front end is an ASP.NET MVC web role and the back end is a worker-role that uses a Service Bus queue. You can create the same multi-tier application with the front end as a web project, that is deployed to an Azure website instead of a cloud service. You can also try out the .NET on-premises/cloud hybrid application tutorial.

Scenario overview: inter-role communication

To submit an order for processing, the front-end UI component, running in the web role, must interact with the middle tier logic running in the worker role. This example uses Service Bus messaging for the communication between the tiers. Using Service Bus messaging between the web and middle tiers decouples the two components.

In contrast to direct messaging (that is, TCP or HTTP), the web tier does not connect to the middle tier directly; instead it pushes units of work, as messages, into Service Bus, which reliably retains them until the middle tier is ready to consume and process them.

Service Bus provides two entities to support brokered messaging: queues and topics. With queues, each message sent to the queue is consumed by a single receiver. Topics support the publish/subscribe pattern in which each published message is made available to a subscription registered with the topic. Each subscription logically maintains its own queue of messages. Subscriptions can also be configured with filter rules that restrict the set of messages passed to the subscription queue to those that match the filter. The following example uses Service Bus queues.

This communication mechanism has several advantages over direct messaging:

- **Temporal decoupling.** With the asynchronous messaging pattern, producers and consumers need not be online at the same time. Service Bus reliably stores messages until the consuming party is ready to receive them. This enables the components of the distributed application to be disconnected, either voluntarily, for example, for maintenance, or due to a component crash, without impacting the system as a whole. Furthermore, the consuming application might only need to come online during certain times of the day.

- **Load leveling.** In many applications, system load varies over time, while the processing time required for each unit of work is typically constant. Intermediating message producers and consumers with a queue means that the consuming application (the worker) only needs to be provisioned to accommodate average load rather than peak load. The depth of the queue grows and contracts as the incoming load varies. This directly saves money in terms of the amount of infrastructure required to service the application load.
- **Load balancing.** As load increases, more worker processes can be added to read from the queue. Each message is processed by only one of the worker processes. Furthermore, this pull-based load balancing enables optimal use of the worker machines even if the worker machines differ in terms of processing power, as they will pull messages at their own maximum rate. This pattern is often termed the *competing consumer* pattern.

The following sections discuss the code that implements this architecture.

Set up the development environment

Before you can begin developing Azure applications, get the tools and set up your development environment.

1. Install the Azure SDK for .NET from the SDK downloads page.
2. In the **.NET** column, click the version of Visual Studio you are using. The steps in this tutorial use Visual Studio 2015, but they also work with Visual Studio 2017.
3. When prompted to run or save the installer, click **Run**.
4. In the **Web Platform Installer**, click **Install** and proceed with the installation.
5. Once the installation is complete, you will have everything necessary to start to develop the app. The SDK includes tools that let you easily develop Azure applications in Visual Studio.

Create a namespace

The next step is to create a *namespace*, and obtain a Shared Access Signature (SAS) key for that namespace. A namespace provides an application boundary for each application exposed through Service Bus. A SAS key is generated by the system when a namespace is created. The combination of namespace name and SAS key provides the credentials for Service Bus to authenticate access to an application.

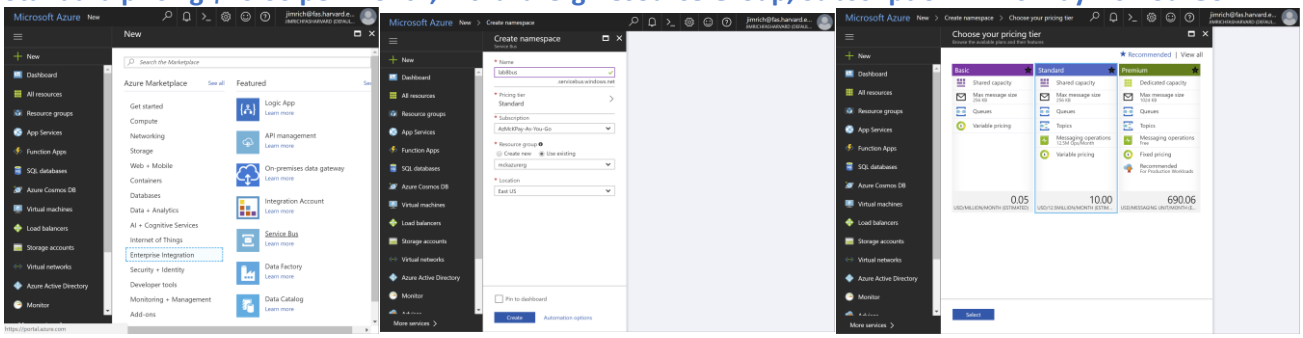
To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

Log on to the Azure portal.

In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**

Standard pricing $10.00 per Month, mckazurerg Resource Group, Subscription AzMcKPay-As-You-Go



In the **Create Namespace** dialog, enter a namespace name: **lab8bus**

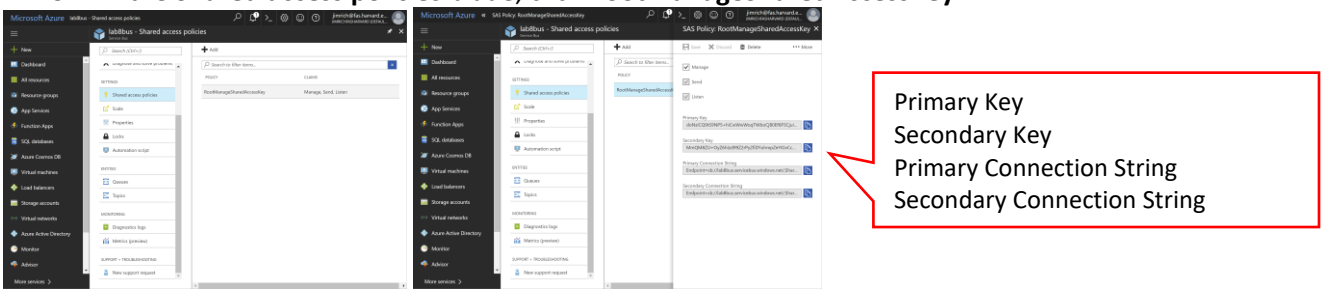
After making sure the namespace name is available, choose the pricing tier (Basic, **Standard**, or Premium).

In the **Subscription** field, choose an Azure subscription in which to create the namespace: **AzMcKPay-As-You-Go**
In the **Resource group** field, choose existing or create one in which namespace will live: **mckazurerg**
In **Location**, choose the country or region in which your namespace should be hosted:**eastus**
Click **Create**. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

Obtain the management credentials

Creating a new namespace will automatically generate an initial Shared Access Signature (SAS) rule with an associated pair of primary and secondary keys that grants full control over all aspects of the namespace. Refer to Service Bus authentication and authorization for how to create further rules with more constrained rights for regular senders and receivers. To copy the initial rule, follow these steps:

1. In the list of namespaces, click the newly created namespace name.
2. In the namespace blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**



4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use. Paste this value into Notepad or some other temporary location.Repeat the previous step, copying and pasting the value of **Primary key** to a temporary location for later use.

---

lab8stor = Storage account Namespace needed for Demo 2
mcarm = resource manager

Primary Key
CsMidVX0x9lv5y+n/dY6m8x8lXyjklwyr2BqiawH+x0X+RZWxCJjbZHarNj+rcdpR4+mitj4IJv/+2/oDipEJQ==
Primary Connection String
DefaultEndpointsProtocol=https;AccountName=lab8stor;AccountKey=CsMidVX0x9lv5y+n/dY6m8x8lXyjklwyr2BqiawH+x0X+RZWxCJjbZHarNj+rcdpR4+mitj4IJv/+2/oDipEJQ==;EndpointSuffix=core.windows.net

Key2
7WwbLtCoiLUg9voSL4+AljvldaKUrtm/HtGshJpAVxoC+QAz3egnSndByjD1BVt1lOSUORxSCuNTW7+OU2qUUQ==
Secondary Connection String
DefaultEndpointsProtocol=https;AccountName=lab8stor;AccountKey=7WwbLtCoiLUg9voSL4+AljvldaKUrtm/HtGshJpAVxoC+QAz3egnSndByjD1BVt1lOSUORxSCuNTW7+OU2qUUQ==;EndpointSuffix=core.windows.net

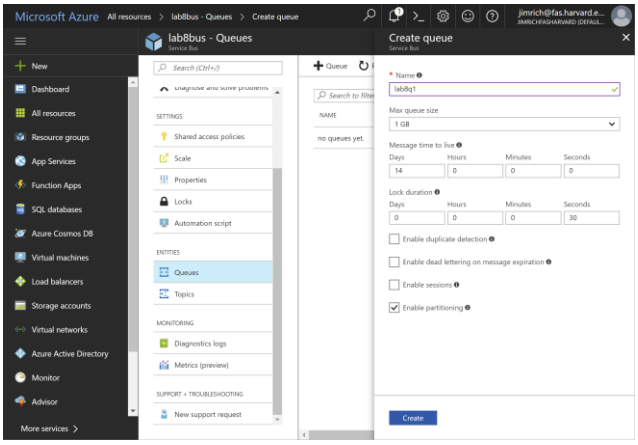
---

lab8bus = Service Bus Namespace … needed for Demo 3
mckazurerg = resource manager

Primary Key
doNsICQ9tS9NP5+hiCwWwWoqTWboCjB0Ef6P3Cjuiqk=
Primary Connection String
Endpoint=sb://lab8bus.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=doNsICQ9tS9NP5+hiCwWwWoqTWboCjB0Ef6P3Cjuiqk=

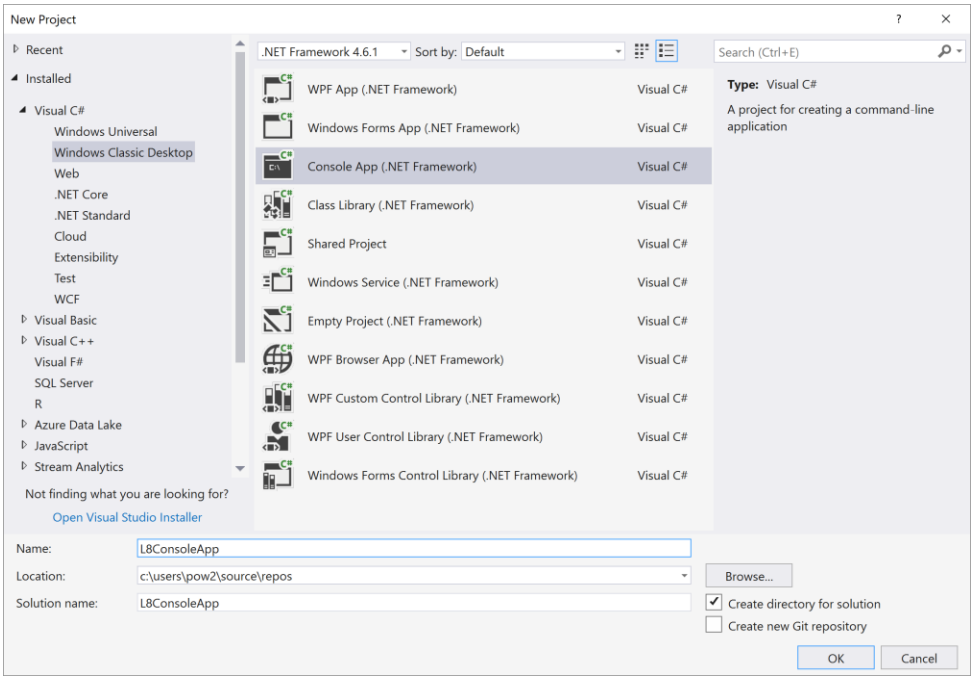
Secondary Key
MmQM8ZU+OyZ6hIzd99Z2rPy2fiDYuhmpZeYIGvCc9ro=
Secondary Connection String
Endpoint=sb://lab8bus.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=MmQM8ZU+OyZ6hIzd99Z2rPy2fiDYuhmpZeYIGvCc9ro=

If you first want to Test Create a Queue using the Azure portal … before delving into DEMO3 VS C# code …
OTHERWISE , If you have already created a Service Bus queue, jump to next section on creating worker role in VS …
Ensure that you have already created a Service Bus namespace …
In the left navigation pane of the portal, click **Service Bus** (if you don't see **Service Bus**, click **More services**).
Click the namespace in which you would like to create the queue. In this case, it is **lab8q1**
In the **Service Bus namespace** blade, select **Queues**, then click **Add queue**
Enter the **Queue Name** and leave the other values with their defaults: **lab8q1**



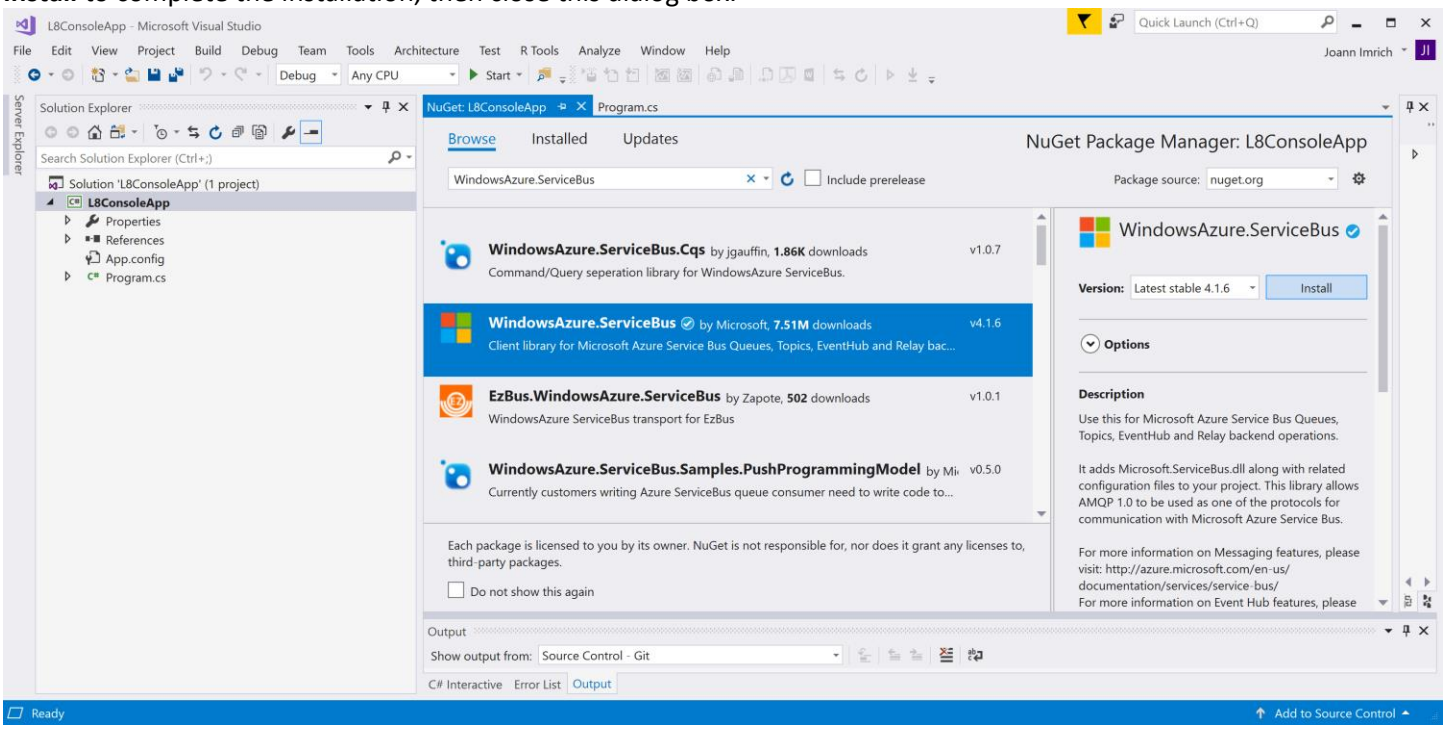
Test Sending messages to the queue using VS Console app

To send messages to the queue, we write a C# **console application** using Visual Studio.
Launch Visual Studio and create a new **Console app (.NET Framework)** project.
Console App is quick and easy way to check your setup before delving into the Lab8 Demo3 code
VS > File >New > Console App > L8ConsoleApp



Add the Service Bus NuGet package
Right-click the newly created project and select **Manage NuGet Packages**.

Click the **Browse** tab, search for **WindowsAzure.ServiceBus**, and then select the **WindowsAzure.ServiceBus** item. Click **Install** to complete the installation, then close this dialog box.



Write some code to Test Sending A Message to the Bus Queue

Add the following `using` statement to the top of the **Program.cs** file.

```
using Microsoft.ServiceBus.Messaging;
```

Add the following code to the `Main` method. Set the `connectionString` variable to the connection string that you obtained when creating the namespace, and set `queueName` to the queue name that you used when creating the queue.

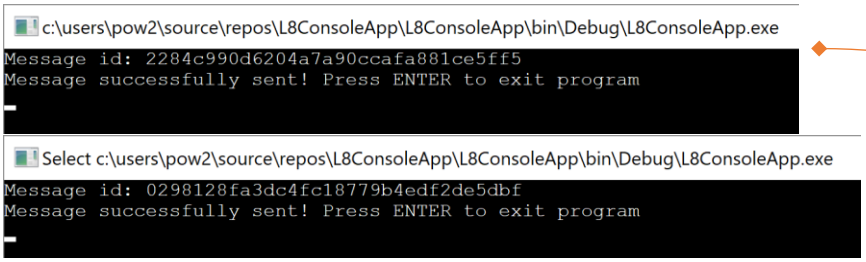
```
var connectionString = "<your connection string>";
var queueName = "<your queue name>";

var client = QueueClient.CreateFromConnectionString(connectionString, queueName);
var message = new BrokeredMessage("This is a test message!");

Console.WriteLine(String.Format("Message id: {0}", message.MessageId));
client.Send(message);

Console.WriteLine("Message successfully sent! Press ENTER to exit program");
Console.ReadLine();
```

Here is what your Program.cs file should look like.

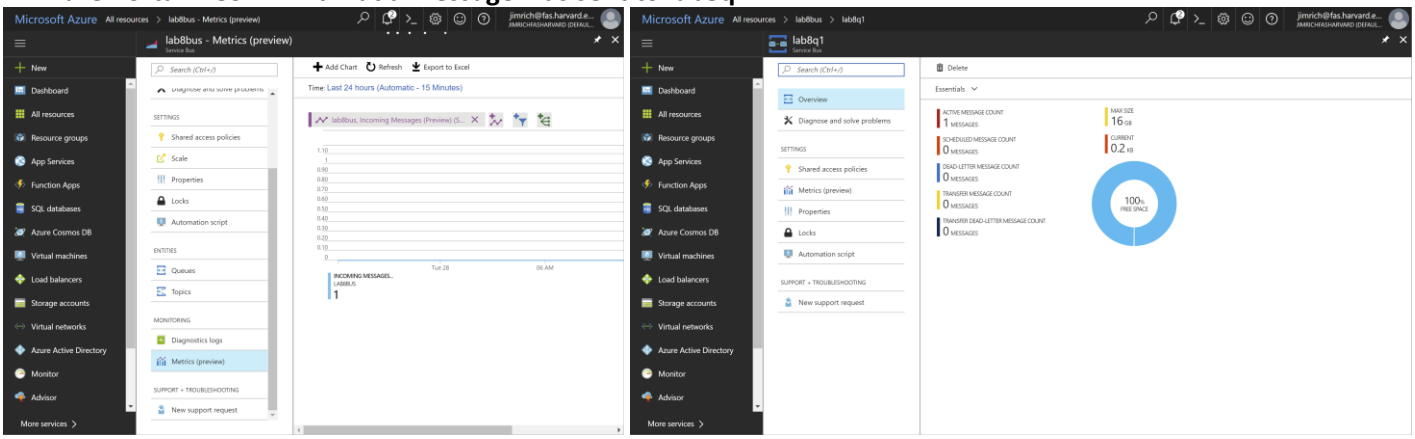
```
using Microsoft.ServiceBus.Messaging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace L8ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            var connectionString =
"Endpoint=sb://lab8bus.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAcce
ssKey=doNsICQ9tS9NP5+hiCwWWWoqTWboCjB0Ef6P3Cjuiqk=";
            var queueName = "lab8q1";

            var client = QueueClient.CreateFromConnectionString(connectionString, queueName);
            var message = new BrokeredMessage("This is lab8 DEMO3 Azure Bus test message!");

            Console.WriteLine(String.Format("Message id: {0}", message.MessageId));

            client.Send(message);

            Console.WriteLine("Message successfully sent! Press ENTER to exit program");
            Console.ReadLine();
        }
    }
}
```

Run the program, and check the Azure portal: click the name of your queue in the namespace **Overview** window. Click on queue, the **essentials window** is displayed. Notice that the **Active Message Count** value should now be 1. Each time you run the sender application without retrieving the messages, this value increases by 1. Also note that the current size of the queue increments each time the app adds a message to the queue.



In Azure Portal – Confirm th1at a message was sent to lab8q1



Test Receiving messages to the queue using VS Console app

To receive the messages you just sent, create a new console application and add a reference to the Service Bus NuGet package, similar to the previous sender application.
Add the following `using` statement to the top of the Program.cs file.

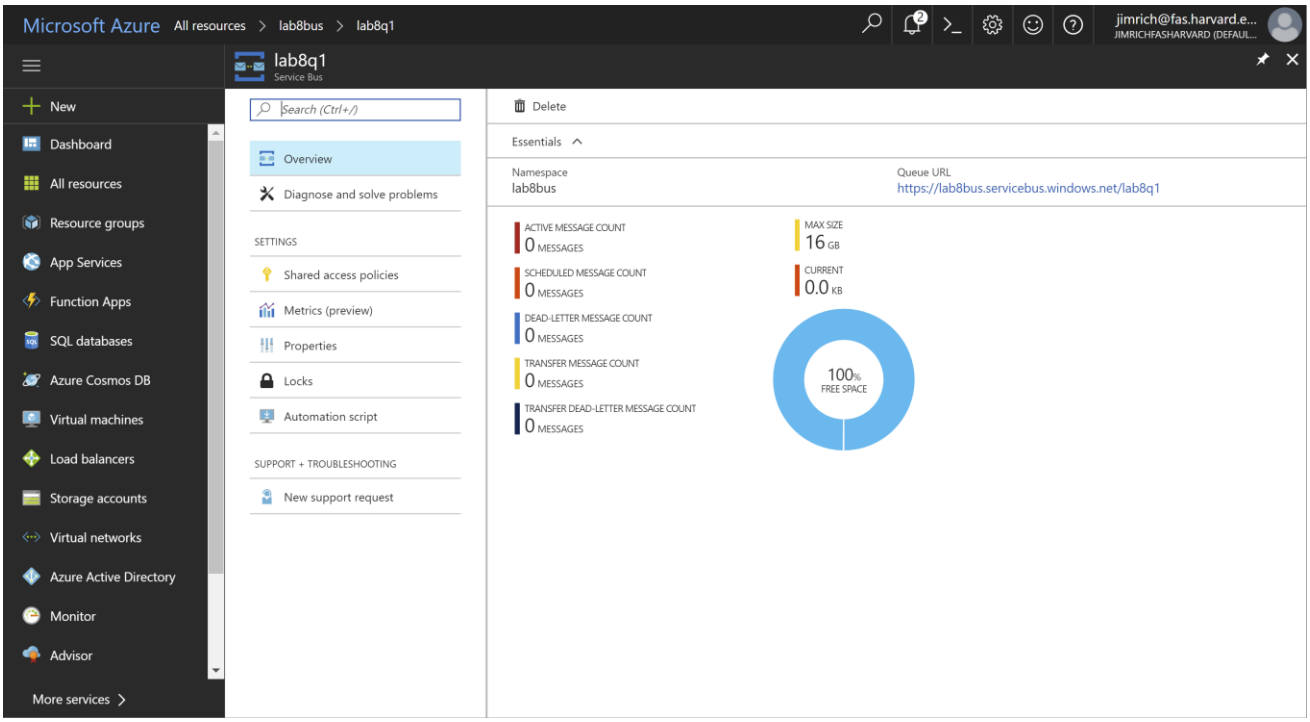
```
using Microsoft.ServiceBus.Messaging;
```

Add the following code to the `Main` method. Set the `connectionString` variable to the connection string that was obtained when creating the namespace, and set `queueName` to the queue name used when creating the queue.

```
var connectionString = "<your connection string>";
var queueName = "<your queue name>";

var client = QueueClient.CreateFromConnectionString(connectionString, queueName);

client.OnMessage(message =>
{
  Console.WriteLine(String.Format("Message body: {0}", message.GetBody<String>()));
  Console.WriteLine(String.Format("Message id: {0}", message.MessageId));
});

Console.WriteLine("Press ENTER to exit program");
Console.ReadLine();
```

Add the following code to the `Main` method. Set the `connectionString` variable to the connection string that was obtained when creating the namespace, and set `queueName` to the queue name that you used when creating the queue. Here is what your Program.cs file should look like:

```
using Microsoft.ServiceBus.Messaging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace L8ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            var connectionString =
"Endpoint=sb://lab8bus.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAc
cessKey=doNsICQ9tS9NP5+hiCwWWWoqTWboCjB0Ef6P3Cjuiqk=";
            var queueName = "lab8q1";

            var client = QueueClient.CreateFromConnectionString(connectionString, queueName);

            client.OnMessage(message =>
            {
                Console.WriteLine(String.Format("Message body: {0}", message.GetBody<String>()));
                Console.WriteLine(String.Format("Message id: {0}", message.MessageId));
            });

            Console.WriteLine("Press ENTER to exit program");
            Console.ReadLine();

        }
    }
}
```



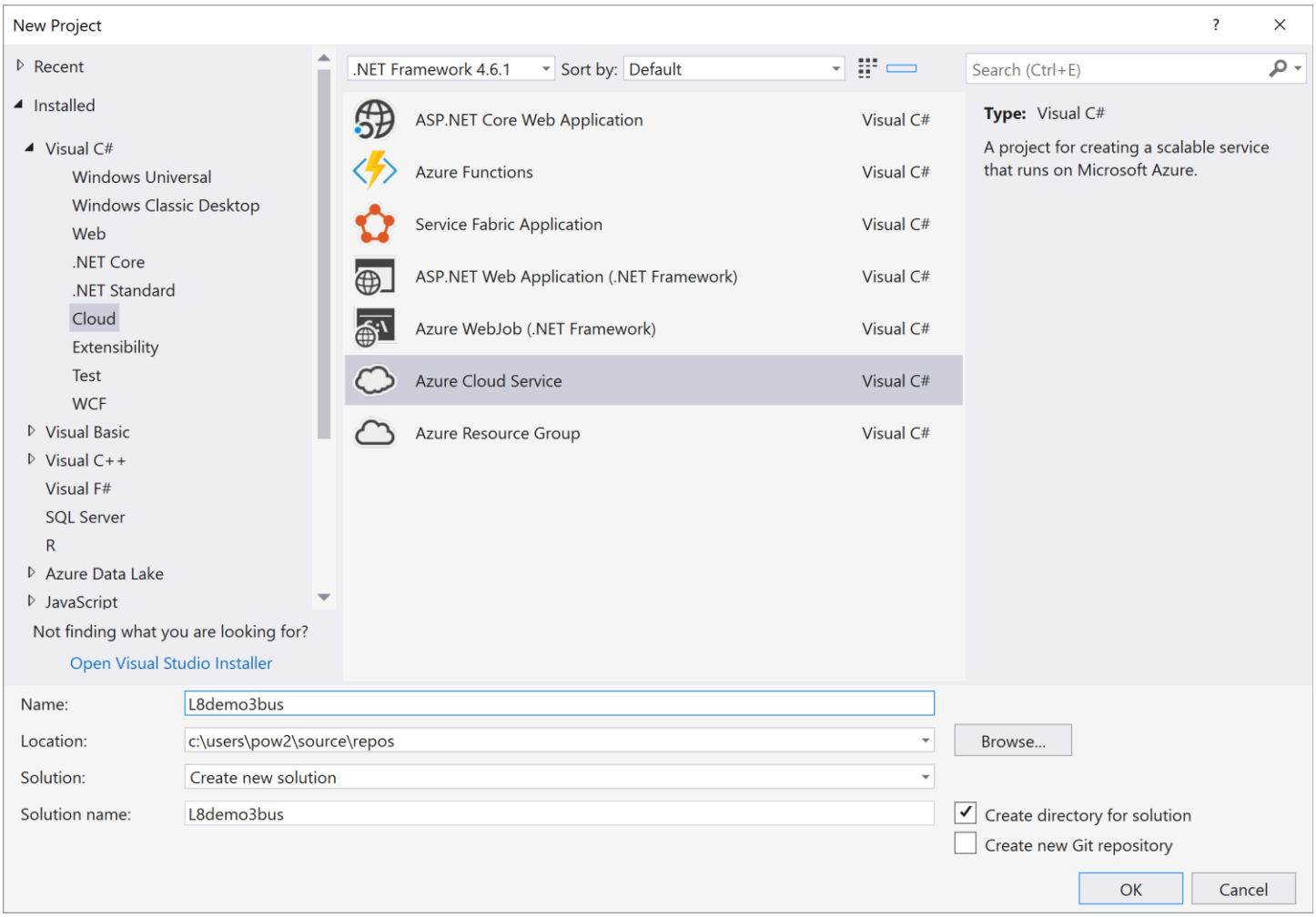
DEMO 3 ---- Begins here

Create a web role
In this section, you build the front end of your application. First, you create the pages that your application displays. After that, add code that submits items to a Service Bus queue and displays status information about the queue.

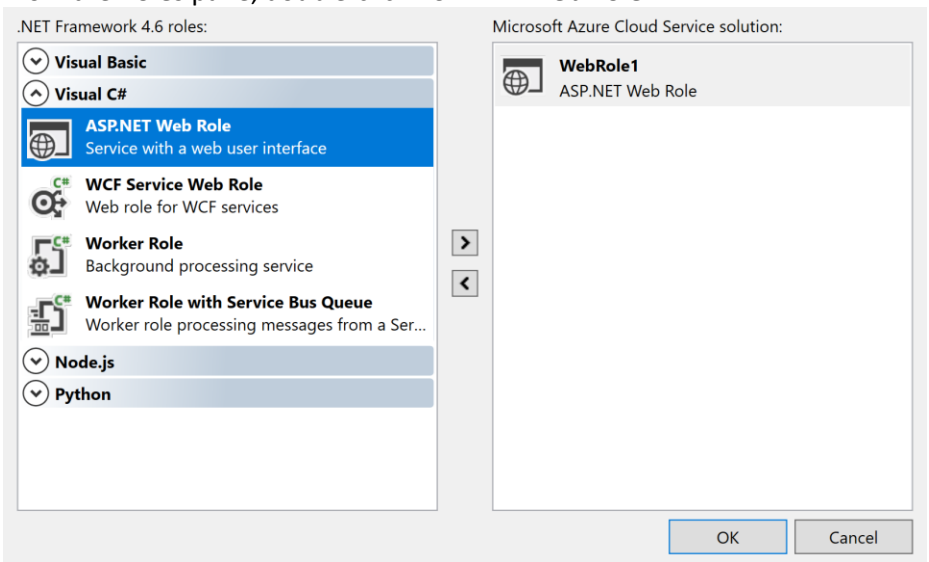
Create the project

Using administrator privileges, start Visual Studio: right-click the **Visual Studio** program icon, and then click **Run as administrator**. The Azure compute emulator, discussed later in this article, requires that Visual Studio be started with administrator privileges.

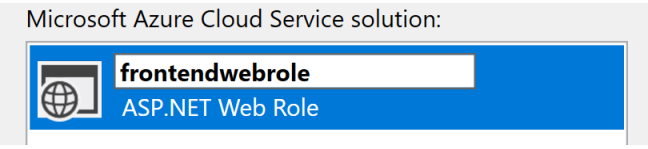
In Visual Studio, on the **File** menu, click **New**, and then click **Project**.
From **Installed Templates**, under **Visual C#**, click **Cloud** and then click **Azure Cloud Service**. Name the project **lab8demo3bus**. Then click **OK**.



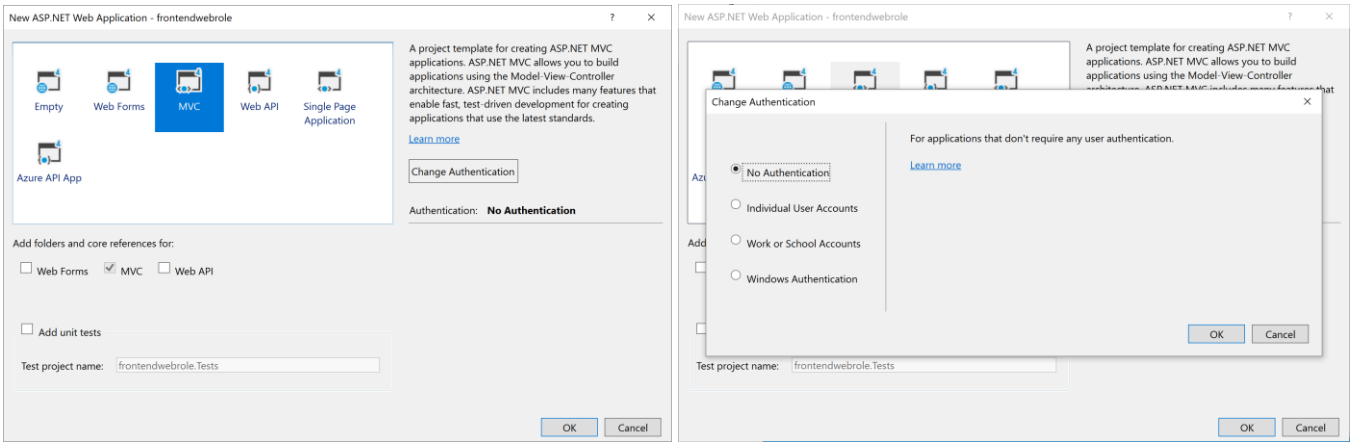
From the **Roles** pane, double-click **ASP.NET Web Role**.



Hover over **WebRole1** under **Azure Cloud Service solution**, click the pencil icon, and rename the web role to **frontendwebrole**. Then click **OK**. (Make sure you enter "frontend" with a lower-case 'e,' not "FrontEnd".)

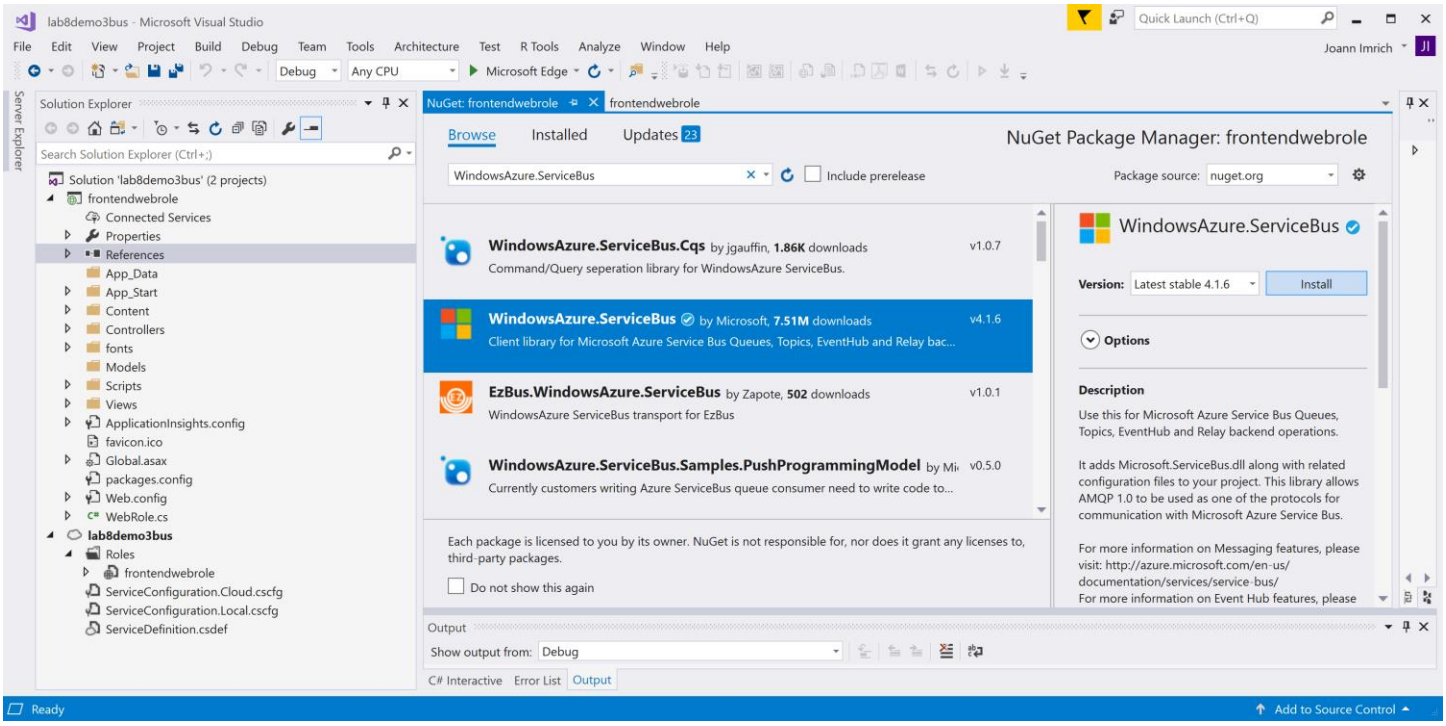


From the **New ASP.NET Project** dialog box, in the **Select a template** list, click **MVC**. Still in the **New ASP.NET Project** dialog box, click the **Change Authentication** button. In the **Change Authentication** dialog box, ensure that **No Authentication** is selected, and then click **OK**. For this tutorial, you're deploying an app that doesn't need a user login.

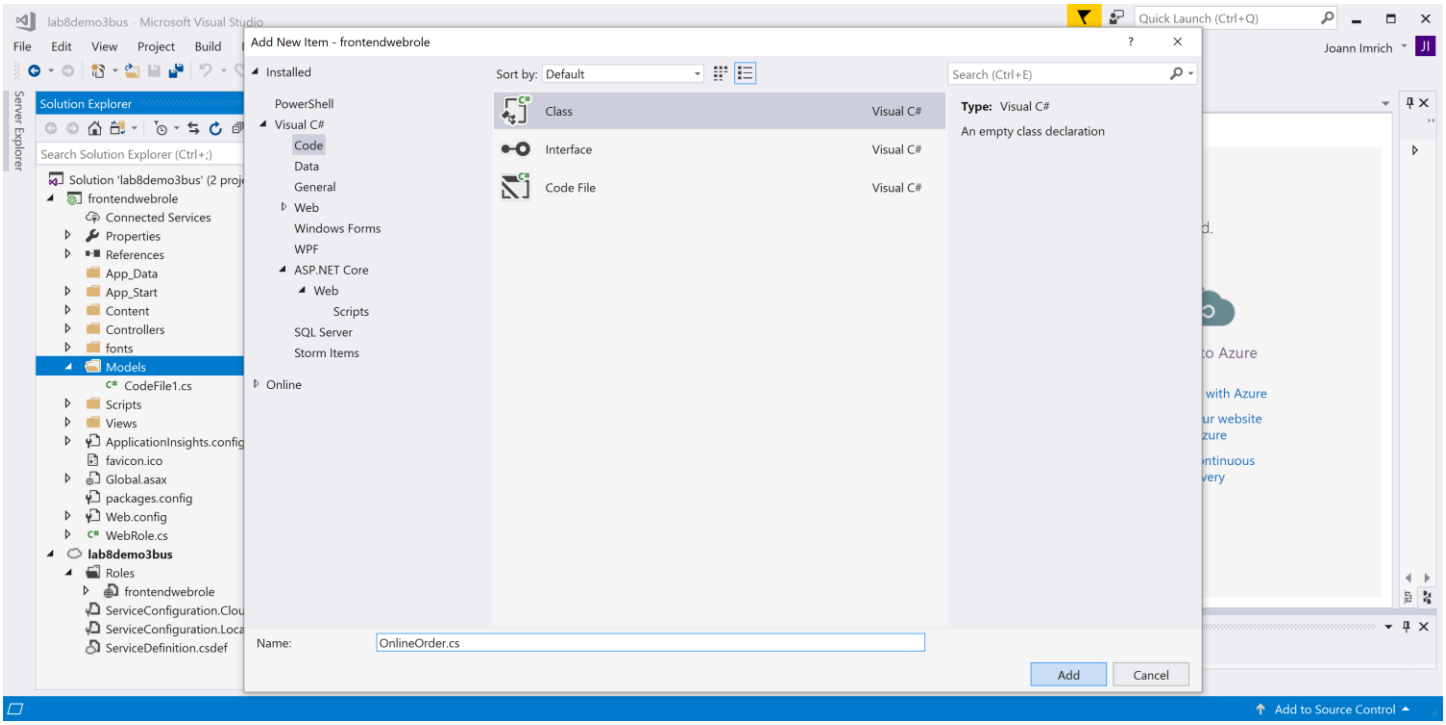


Back in the **New ASP.NET Project** dialog box, click **OK** to create the project.

In **Solution Explorer**, in the **frontendwebrole** project, right-click **References**, then click **Manage NuGet Packages**. Click the **Browse** tab, then search for **WindowsAzure.ServiceBus**. Select the **WindowsAzure.ServiceBus** package, click **Install**, and accept the terms of use.Note that the required client assemblies are now referenced and some new code files have been added.



In **Solution Explorer**, right-click **Models** and click **Add**, then click **Class**. In the **Name** box, type the name **OnlineOrder.cs**. Then click **Add**.



Write the code for your web role

In this section, you create the various pages that your application displays.

In the **OnlineOrder.cs** file in Visual Studio, replace the existing namespace definition with the following code:

```
namespace frontendwebrole.Models
{
    public class OnlineOrder
    {
        public string Customer { get; set; }
        public string Product  { get; set; }
    }
}
```

Solution Explorer, double-click **Controllers\HomeController.cs**. Add the following **using** statements at the top of the file to include the namespaces for the model you just created, as well as Service Bus.

```
using frontendwebrole.Models;
using Microsoft.ServiceBus.Messaging;
using Microsoft.ServiceBus;
```

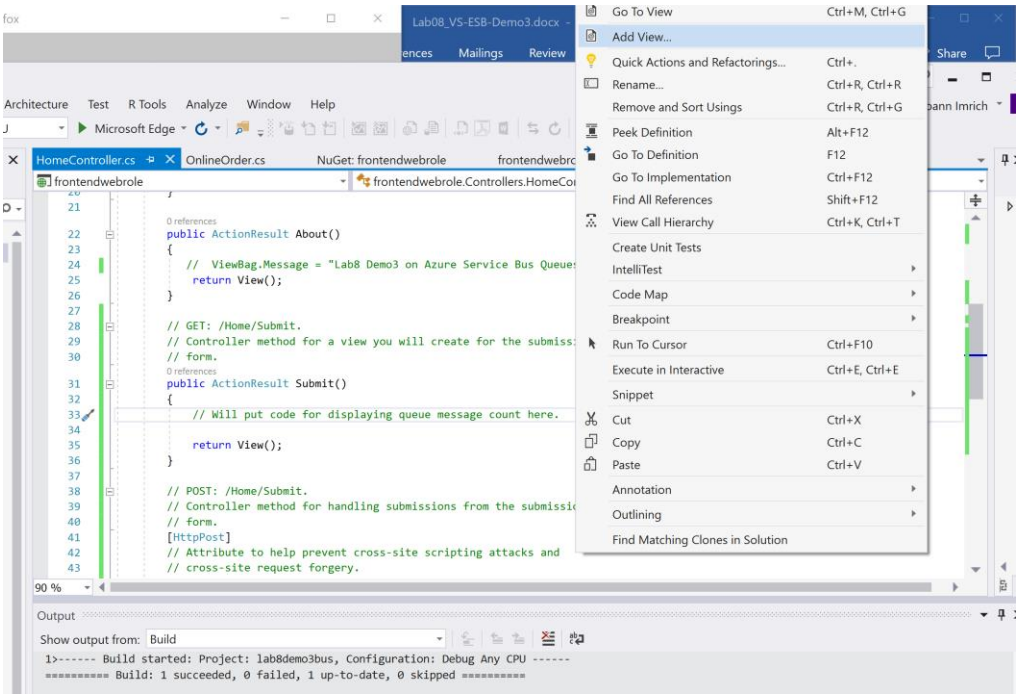
Also in the **HomeController.cs** file in Visual Studio, replace the existing namespace definition with the following code. This code contains methods for handling the submission of items to the queue.

```
namespace frontendwebrole.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
```

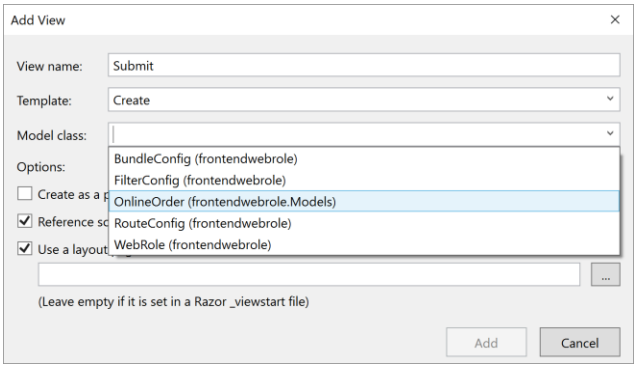
```
            // Simply redirect to Submit, since Submit will serve as the
            // front page of this application.
            return RedirectToAction("Submit");
        }

        public ActionResult About()
        {
            return View();
        }

        // GET: /Home/Submit.
        // Controller method for a view you will create for the submission
        // form.
        public ActionResult Submit()
        {
            // Will put code for displaying queue message count here.

            return View();
        }

        // POST: /Home/Submit.
        // Controller method for handling submissions from the submission
        // form.
        [HttpPost]
        // Attribute to help prevent cross-site scripting attacks and
        // cross-site request forgery.
        [ValidateAntiForgeryToken]
        public ActionResult Submit(OnlineOrder order)
        {
            if (ModelState.IsValid)
            {
                // Will put code for submitting to queue here.

                return RedirectToAction("Submit");
            }
            else
            {
                return View(order);
            }
        }
    }
}
```

From the **Build** menu, click **Build Solution** to test the accuracy of your work so far.

Now, create the view for the Submit() method you created earlier. Right-click within the Submit() method (the overload of Submit() that takes no parameters), and then choose **Add View**.



A dialog box appears for creating the view. In the **Template** list, choose **Create**. In the **Model class** list, select the **OnlineOrder** class. Click **Add**.



Now, change the displayed name of your application.

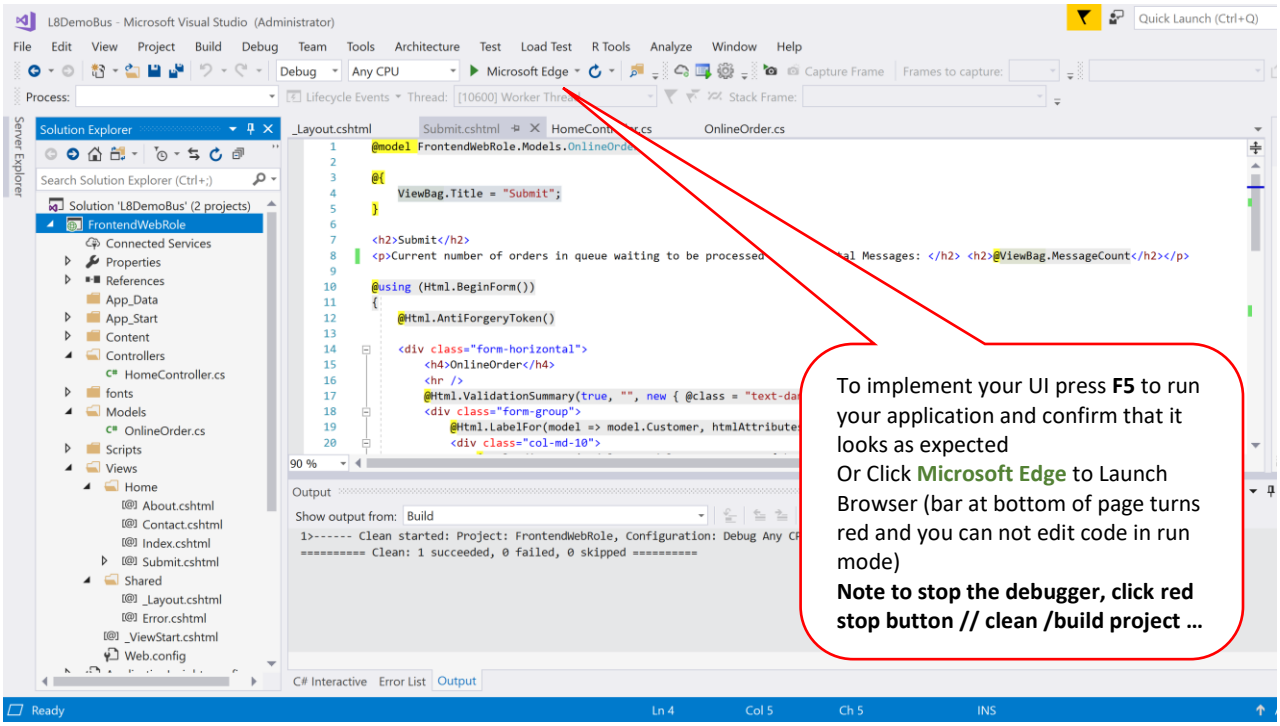
In **Solution Explorer**, double-click the **Views\Shared_Layout.cshtml** file to open it in the Visual Studio editor.

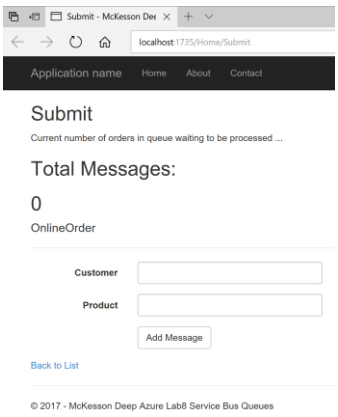
Replace all occurrences of **My ASP.NET Application** with **"Lab8 Demo3 Service Bus App"**

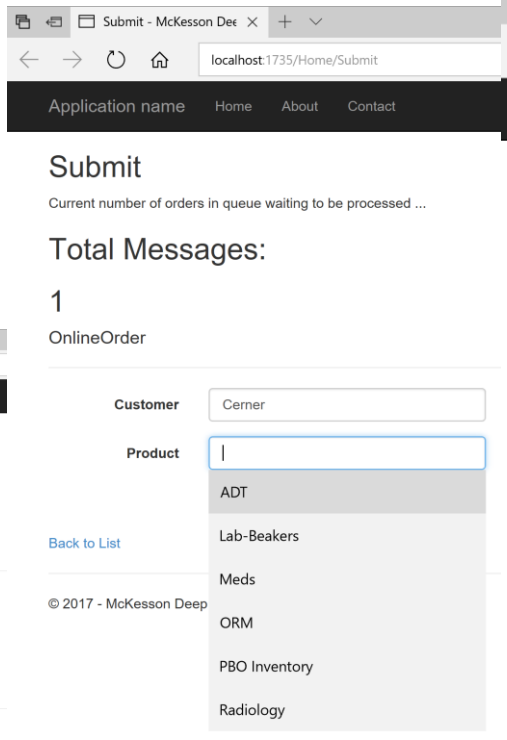
Remove the **Home**, **About**, and **Contact** links. Delete the highlighted code:

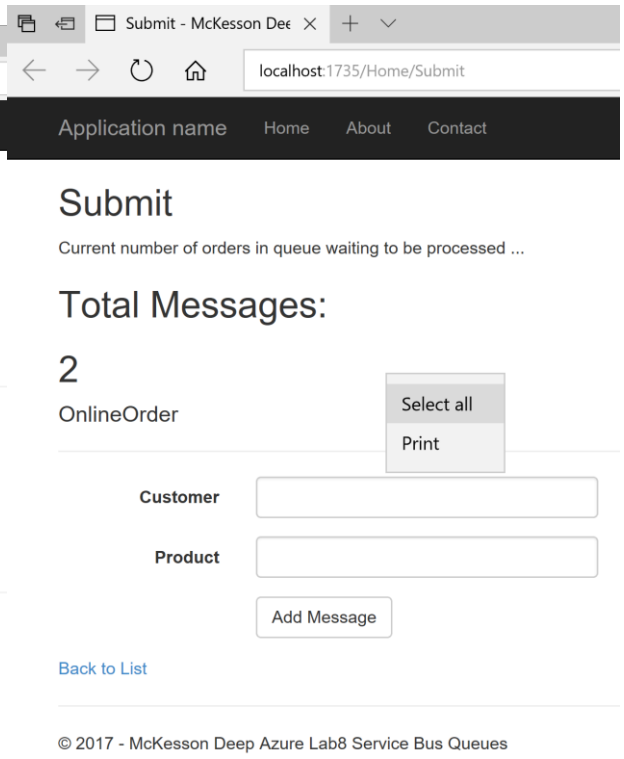
Finally, modify the submission page to include some information about the queue.
In **Solution Explorer**, double-click the **Views\Home\Submit.cshtml** file to open it in the Visual Studio editor. Add the following line after <h2>Submit</h2> at top of file after @{`ViewBag.Title = "Submit";`}`method.` For now, the ViewBag.MessageCount is empty. You will populate it later.

```
<p>Current number of orders in queue waiting to be processed: @ViewBag.MessageCount</p>
```





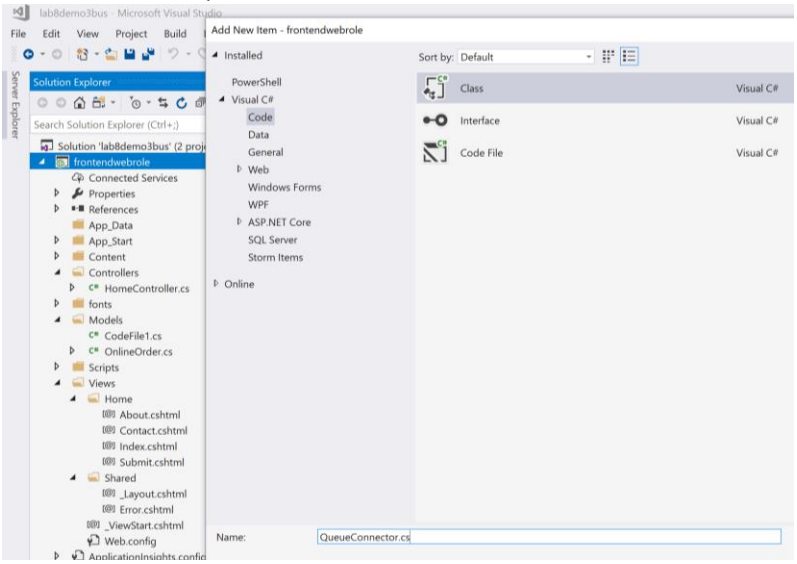




Write the code for submitting items to a Service Bus Queue

Now, add code for submitting items to a queue. First, you create a class that contains your Service Bus queue connection information. Then, initialize your connection from **Global.aspx.cs**. Finally, update the submission code you created earlier in **HomeController.cs** to actually submit items to a Service Bus queue.

In **Solution Explorer**, right-click **FrontendWebRole** (right-click the project, not the role). Click **Add**, and then click **Class**. Name the class **QueueConnector.cs**. Click **Add** to create the class.



Now, add code that encapsulates the connection information and initializes the connection to a Service Bus queue. Replace the entire contents of **QueueConnector.cs** with the following code, and enter values for your **Service Bus namespace** **lab8bus** and yourKey, which is the **primary key** you previously **obtained from the Azure portal.**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Microsoft.ServiceBus.Messaging;
using Microsoft.ServiceBus;

namespace FrontendWebRole
{
    public static class QueueConnector
    {
        // Thread-safe. Recommended that you cache rather than recreating it
        // on every request.
        public static QueueClient OrdersQueueClient;

        // Obtain these values from the portal.
        public const string Namespace = "your Service Bus namespace";

        // The name of your queue.
        public const string QueueName = "OrdersQueue";

        public static NamespaceManager CreateNamespaceManager()
        {
            // Create the namespace manager which gives you access to
            // management operations.
            var uri = ServiceBusEnvironment.CreateServiceUri(
                "sb", Namespace, String.Empty);
            var tP = TokenProvider.CreateSharedAccessSignatureTokenProvider(
                "RootManageSharedAccessKey", "yourKey");
```

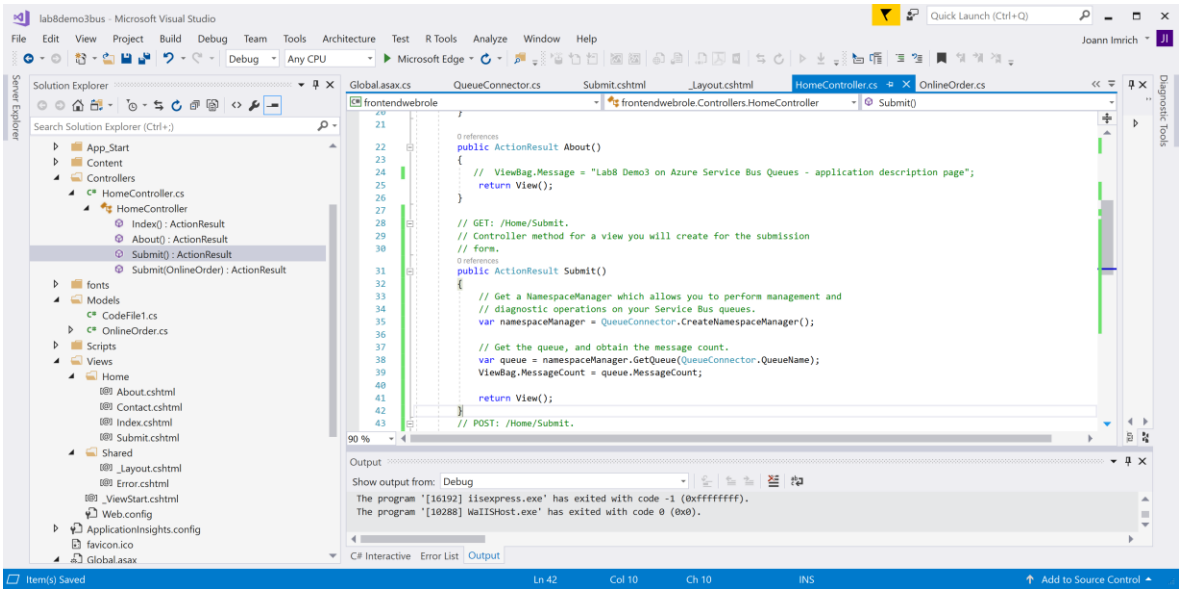
```
                return new NamespaceManager(uri, tP);
            }

            public static void Initialize()
            {
                // Using Http to be friendly with outbound firewalls.
                ServiceBusEnvironment.SystemConnectivity.Mode =
                    ConnectivityMode.Http;

                // Create the namespace manager which gives you access to
                // management operations.
                var namespaceManager = CreateNamespaceManager();

                // Create the queue if it does not exist already.
                if (!namespaceManager.QueueExists(QueueName))
                {
                    namespaceManager.CreateQueue(QueueName);
                }

                // Get a client to the queue.
                var messagingFactory = MessagingFactory.Create(
                    namespaceManager.Address,
                    namespaceManager.Settings.TokenProvider);
                OrdersQueueClient = messagingFactory.CreateQueueClient(
                    "OrdersQueue");
            }
        }
    }
```

Now, ensure that your **Initialize** method gets called. In **Solution Explorer**, double-click **Global.asax\Global.asax.cs**.
Add the following line of code at the end of the **Application_Start** method.

```
FrontendWebRole.QueueConnector.Initialize();
```

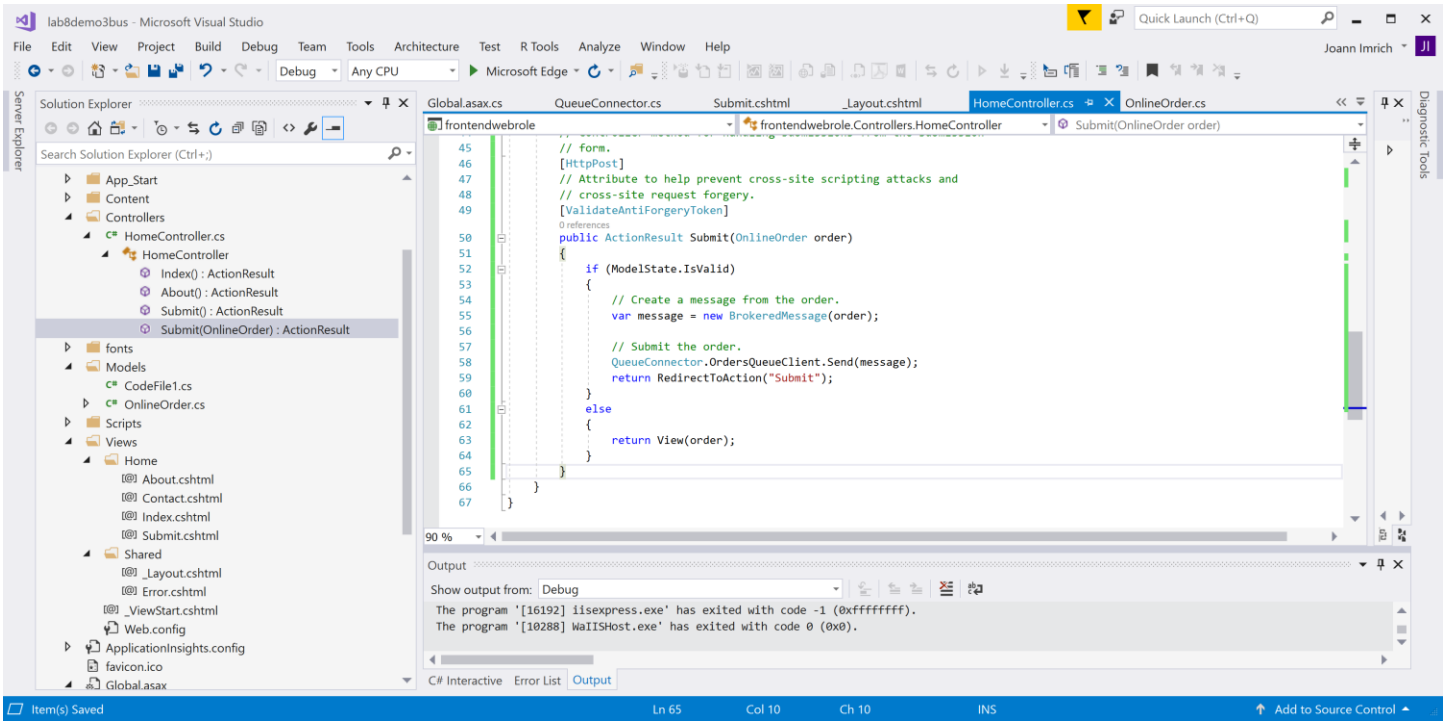
Finally, update the web code you created earlier, to submit items to the queue. In **Solution Explorer**, double-click **Controllers\HomeController.cs**.
Update the Submit() method (the overload that takes no parameters) as follows to get the message count for the queue.

```
public ActionResult Submit()
{
    // Get a NamespaceManager which allows you to perform management and
    // diagnostic operations on your Service Bus queues.
    var namespaceManager = QueueConnector.CreateNamespaceManager();

    // Get the queue, and obtain the message count.
    var queue = namespaceManager.GetQueue(QueueConnector.QueueName);
    ViewBag.MessageCount = queue.MessageCount;

    return View();
}
```



Update the Submit(OnlineOrder order) method (the overload that takes one parameter) as follows to submit order information to the queue.

```
public ActionResult Submit(OnlineOrder order)
{
    if (ModelState.IsValid)
    {
        // Create a message from the order.
        var message = new BrokeredMessage(order);

        // Submit the order.
        QueueConnector.OrdersQueueClient.Send(message);
        return RedirectToAction("Submit");
    }
    else
    {
        return View(order);
    }
}
```



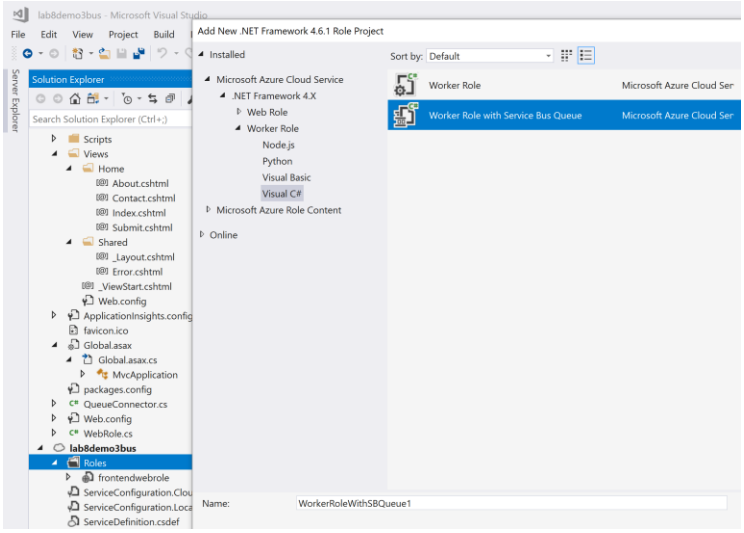
You can now run the application again. Each time you submit an order, the message count increases.

Create the worker role

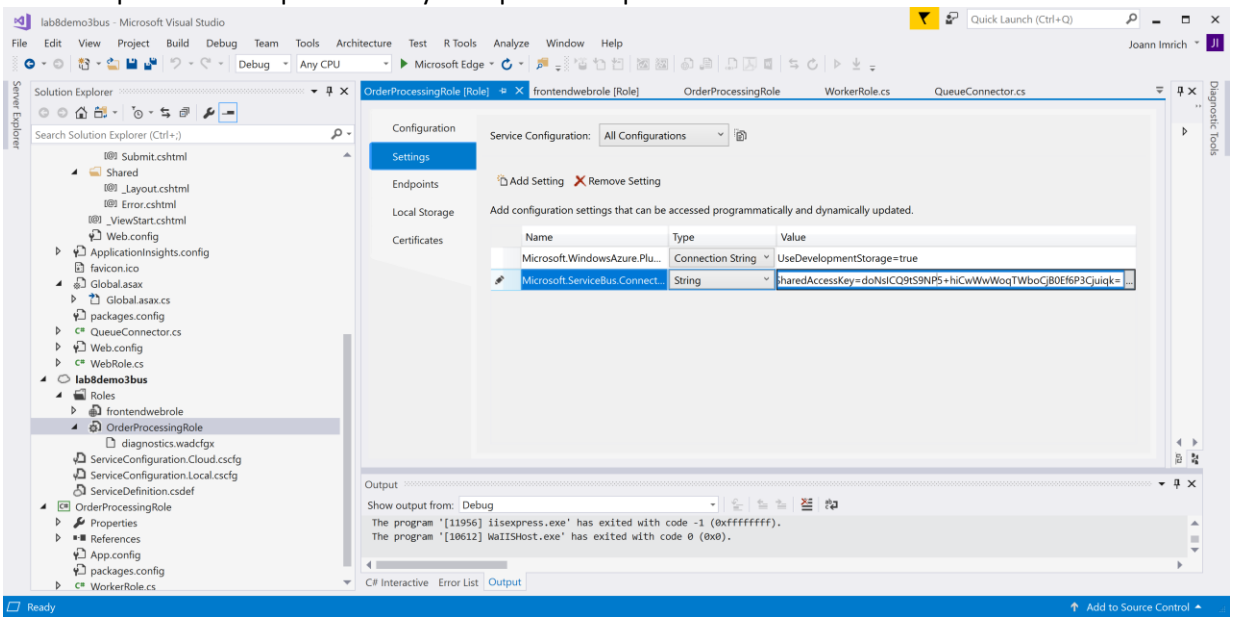
You will now create the worker role that processes the order submissions. This example uses the **Worker Role with Service Bus Queue** Visual Studio project template. You already obtained the required credentials from the portal. Make sure you have connected Visual Studio to your Azure account.

In Visual Studio, in **Solution Explorer** right-click the **Roles** folder under the **MultiTierApp** project. Click **Add**, and then click **New Worker Role Project**. The **Add New Role Project** dialog box appears.

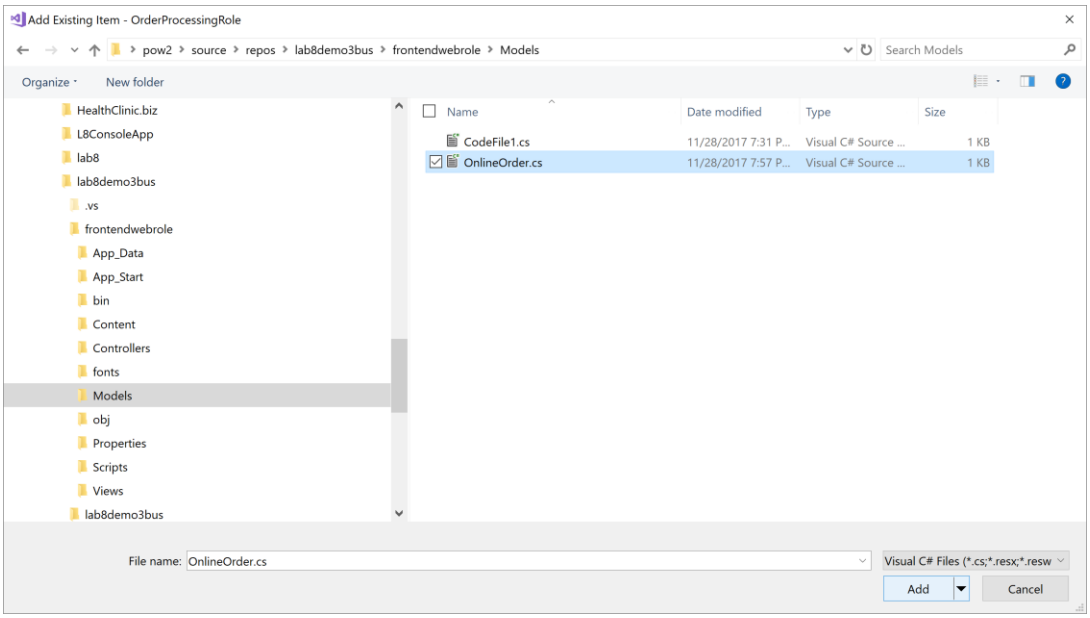
In the **Add New Role Project** dialog box, click **Worker Role with Service Bus Queue**.



In the **Name** box, name the project **OrderProcessingRole**. Then click **Add**.
Copy the connection string that you obtained in step 9 of the "Create a Service Bus namespace" section to the clipboard.
In **Solution Explorer**, right-click the **OrderProcessingRole** you created in step 5 (make sure that you right-click **OrderProcessingRole** under **Roles**, and not the class). Then click **Properties**.
On the **Settings** tab of the **Properties** dialog box, click inside the **Value** box for **Microsoft.ServiceBus.ConnectionString**, and then paste the endpoint value you copied in step 6.

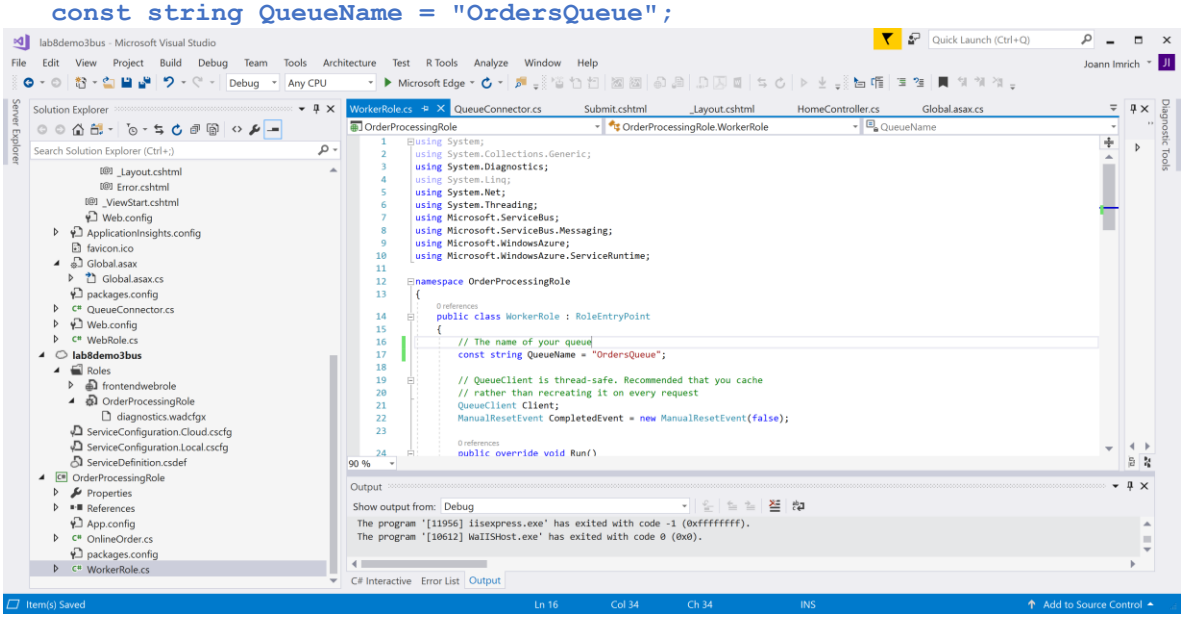


Create an **OnlineOrder** class to represent the orders as you process them from the queue. You can reuse a class you have already created. In **Solution Explorer**, right-click the **OrderProcessingRole** class (right-click the class icon, not the role). Click **Add**, then click **Existing Item**. Browse to the subfolder for **FrontendWebRole\Models**, and then double-click **OnlineOrder.cs** to add it to this project.



In **WorkerRole.cs**, change the value of the **QueueName** variable from "ProcessingQueue" to "OrdersQueue" as shown in the following code.

```
// The name of your queue.
const string QueueName = "OrdersQueue";
```

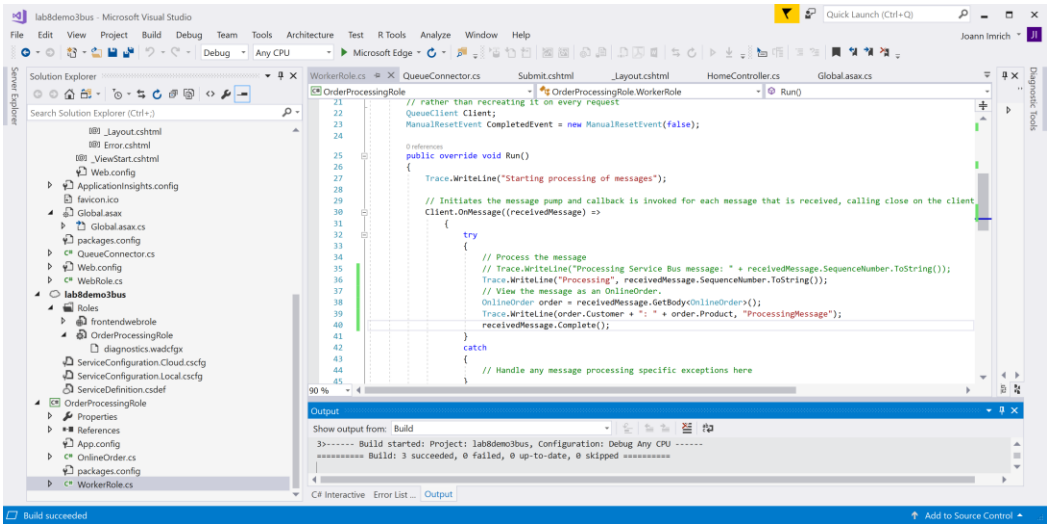


Add the following using statement at the top of the **WorkerRole.cs** file.

```
using frontendwebrole.Models;
```

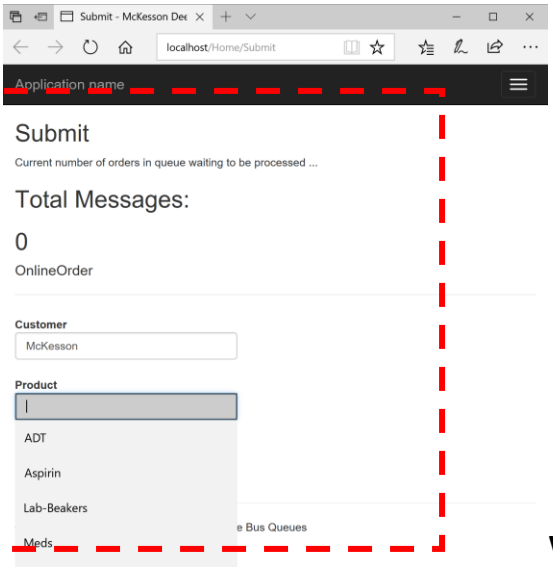
In the Run() function, inside the OnMessage() call, replace the contents of the try clause with the following code.

```
Trace.WriteLine("Processing", receivedMessage.SequenceNumber.ToString());
// View the message as an OnlineOrder.
OnlineOrder order = receivedMessage.GetBody<OnlineOrder>();
Trace.WriteLine(order.Customer + ": " + order.Product, "ProcessingMessage");
receivedMessage.Complete();
```

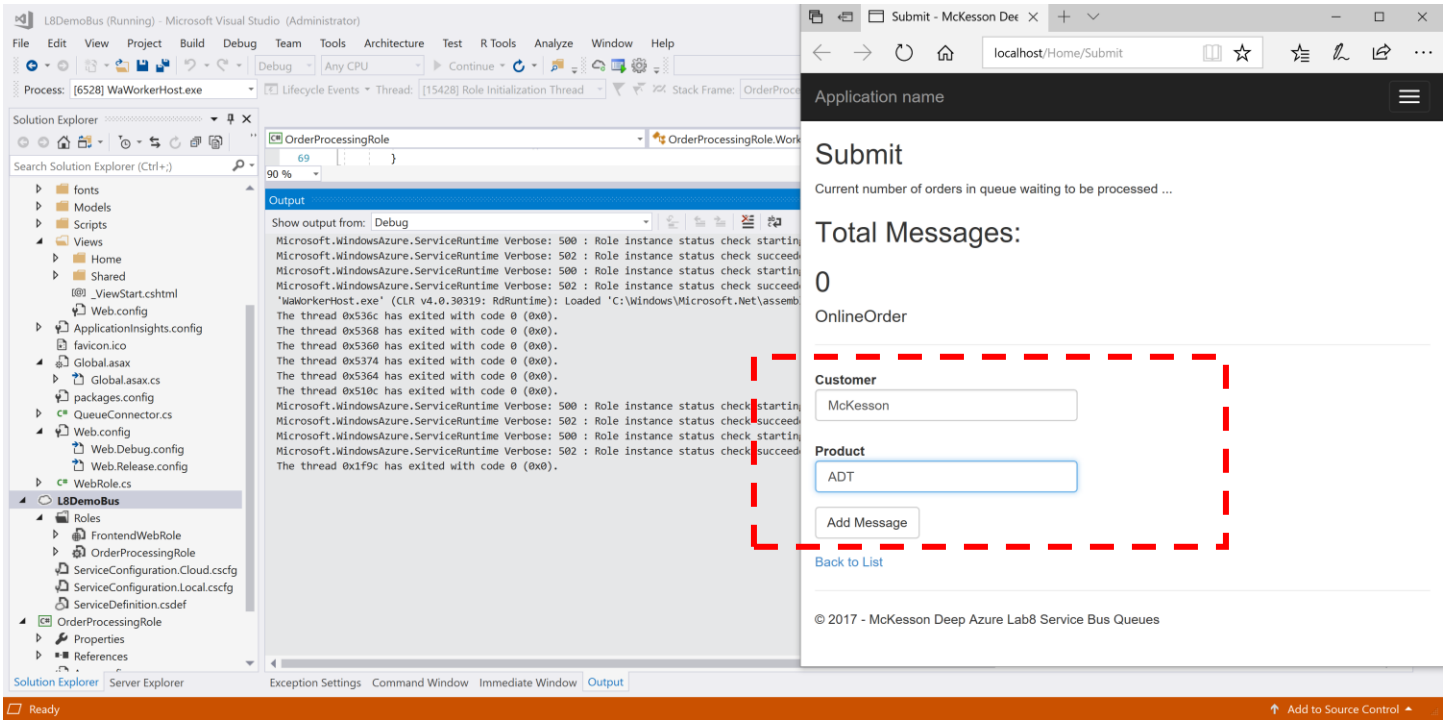


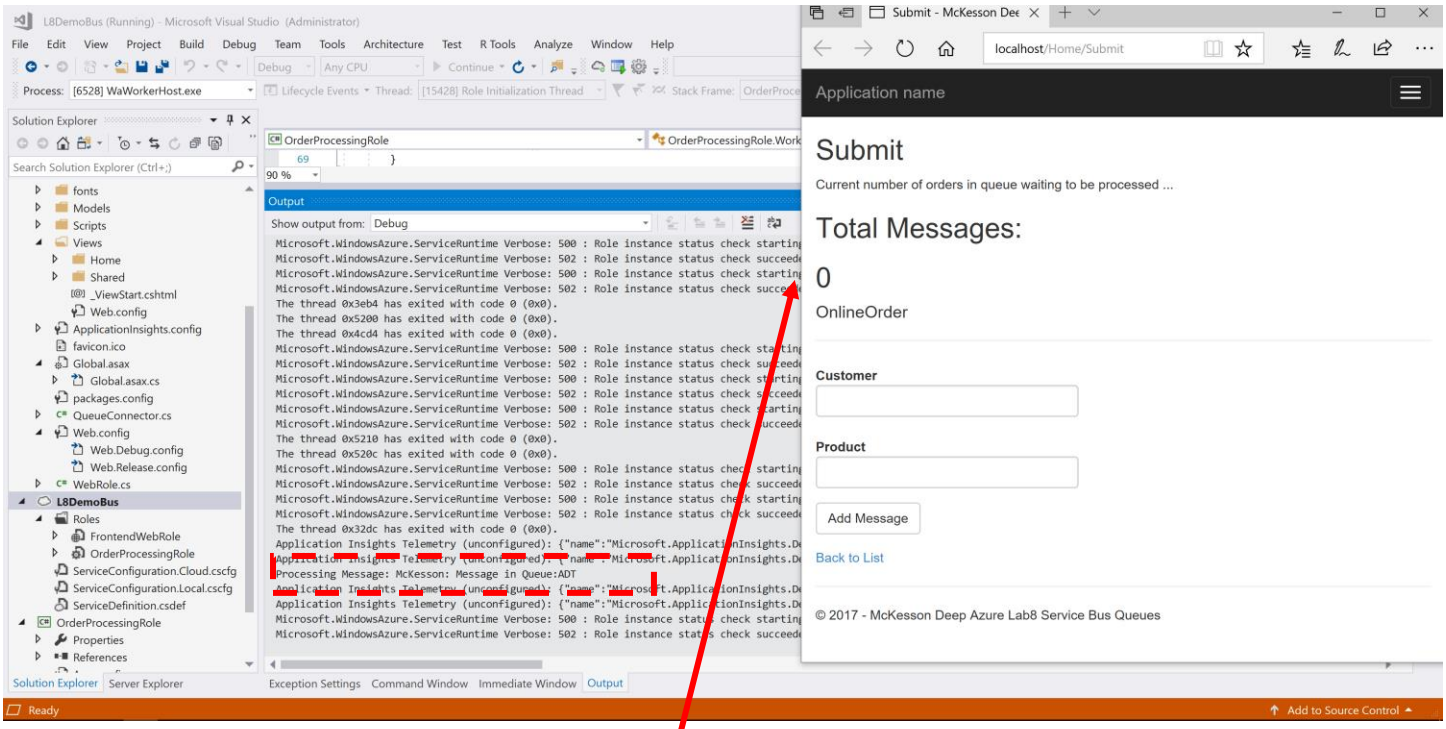
You have completed the application. You can test the full application by right-clicking the **lab8demo3bus** project in Solution Explorer, selecting **Set as Startup Project**, and then pressing F5.

From Browser:



Web Role





Worker Role

Note that the message count does not increment, because the worker role processes items from the queue and marks them as complete. You can see the trace output of your worker role by viewing the Azure Compute Emulator UI.

You can do this by right-clicking the emulator icon in the notification area of your taskbar and selecting **Show Compute Emulator UI**.
https://docs.microsoft.com/en-us/azure/storage/common/storage-use-emulator

References
https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dotnet-multi-tier-app-using-service-bus-queues

For those that prefer to use the Azure Portal to Create Service Bus Queues …
A good that tutorial covers the following steps:

1. Create a Service Bus namespace, using the Azure portal.
2. Create a Service Bus queue, using the Azure portal.
3. Write a console application to send a message.
4. Write a console application to receive the messages sent in the previous step

https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dotnet-get-started-with-queues