# Building and Deploying a WebJob using the WebJobs SDK

The goal of this lab is to give you a step by step walk through of creating and deploying a WebJob using C# and Visual Studio 2015.

In this hands-on lab, you will explore the basics of WebJobs SDK and the WebJobs SDK Extensions by creating a simple console application that mimics the general steps of an order processing system.

This lab starts with building a simple WebJob, and then make several improvements using some of the customization points in the WebJobs SDK to accomplish things that a real production WebJob will need to do.

**Prerequisites**

* Visual Studio 2015 Installed
* Azure Subscription
* Send Grid API Key
* Internet connection

**Assumptions**

* You have experience developing C# applications

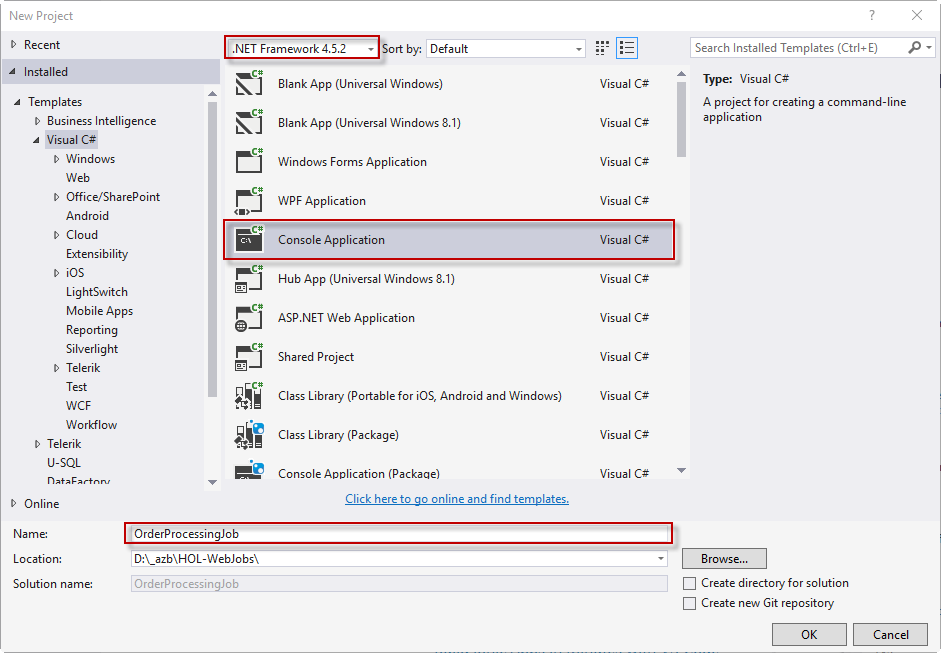
This lab includes the following sections:

1. Create a Console Application project for the WebJob functionality
2. Setup a storage account to use with the WebJob
3. Add a function to the WebJob to watch a queue for Orders and then save to Blob storage
4. Add a test function to verify the queue and blob logic works
5. Modify the logic to use a custom NameResolver
6. Modify the logic to use a custom trace writer
7. Add functionality to save the Products to a storage table
8. Add functionality to send an email once the order is processed
9. Add a function to handle poison messages on the Orders queue to send out an email
10. Add a function that emails when a certain number of error have happened
11. Deploy the WebJob to Azure

## Create a Console Application project for the WebJob functionality

In this section we will create the console application that we will be adding functionality to throughout this hands-on lab.

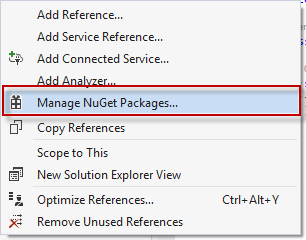
1. Open **Visual Studio 2015** and click the **New Project** link on the start page (or File -> New -> Project.
2. Create a new **Console Application** using **C#, .NET Framework 4.5.2** and name it **OrderProcessingJob**.



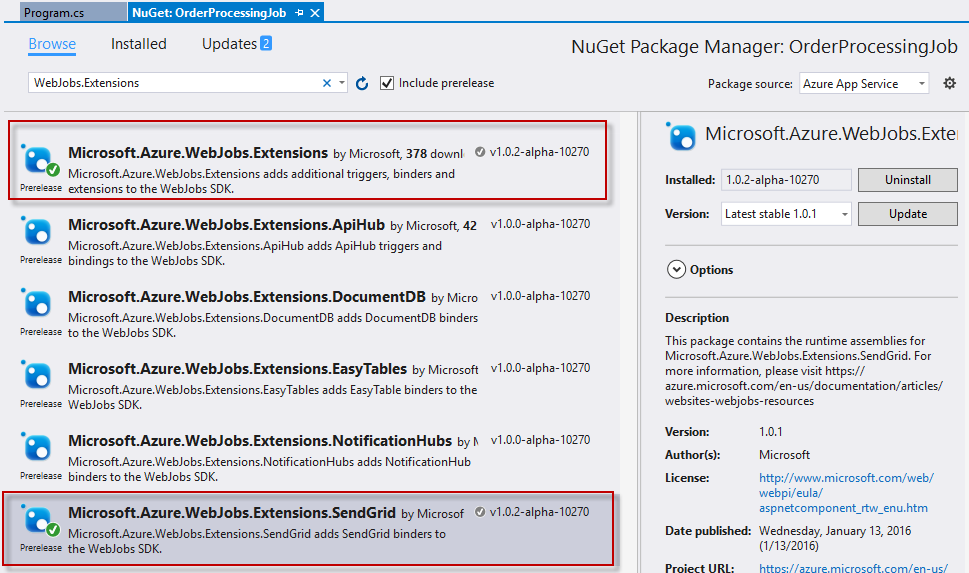
Before starting the WebJobs logic, we need to add a NuGet package (which will add all the other dependencies we need along with it):

* Microsoft.Azure.WebJobs.Extensions.SendGrid

1. In the **Solution Explorer**, **right-click the References** node and select **Manage NuGet Packages** …



1. In the **NuGet window**, select the **Browse** tab, check the **Include prerelease** checkbox and type **WebJobs.Extensions** in the search box.
2. Select **Microsoft.Azure.WebJobs.Extensions.SendGrid** and click **Install** button. This will ask you to confirm some license boxes



You should now see the green check by the NuGet packages that were installed.

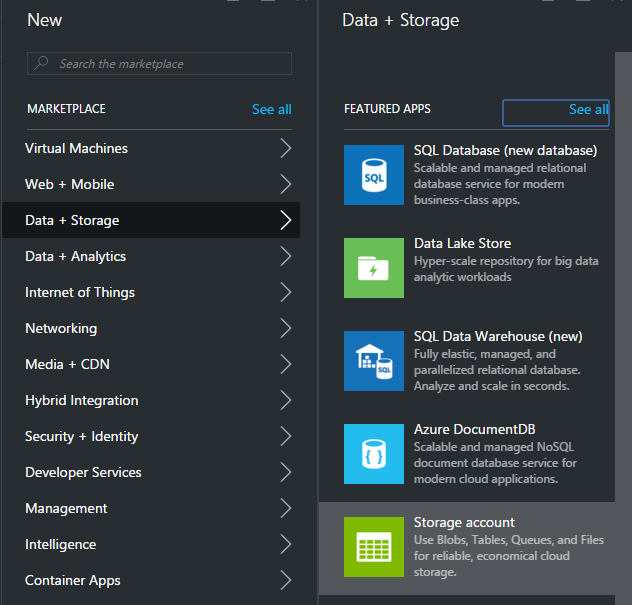
1. In the Solution Explorer, right-click on the **OrderProcessingJob project** and select **Add -> Class**. Name it **Order.cs** and click **Add**.
2. Open the **Order** class and insert the following C#

|  |
| --- |
| using System;  using System.Collections.Generic;    namespace OrderProcessingJob  {      public class Order      {          public string Id { get; set; }          public string BuyerName { get; set; }          public string BuyerEmail { get; set; }          public string BuyerCell { get; set; }          public DateTime OrderPlacedUtc { get; set; }          public List<Product> Products { get; set; }      }        public class Product      {          public string Id { get; set; }          public string Name { get; set; }          public int Quantity { get; set; }          public double Price { get; set; }      }  } |

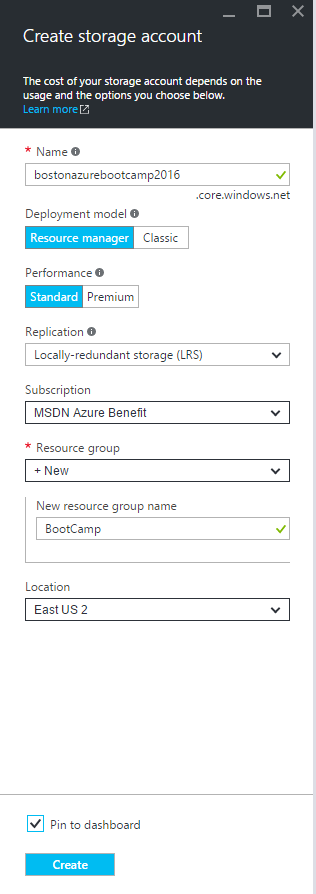
## Setup a storage account to use with the WebJob

Since the WebJob is going to use queues, tables and blobs, we need to create a storage account to use.

1. Open a browser and go to [**https://portal.azure.com**](https://portal.azure.com) and sign in with your Azure credentials.
2. On the top left, click the **+ New button**



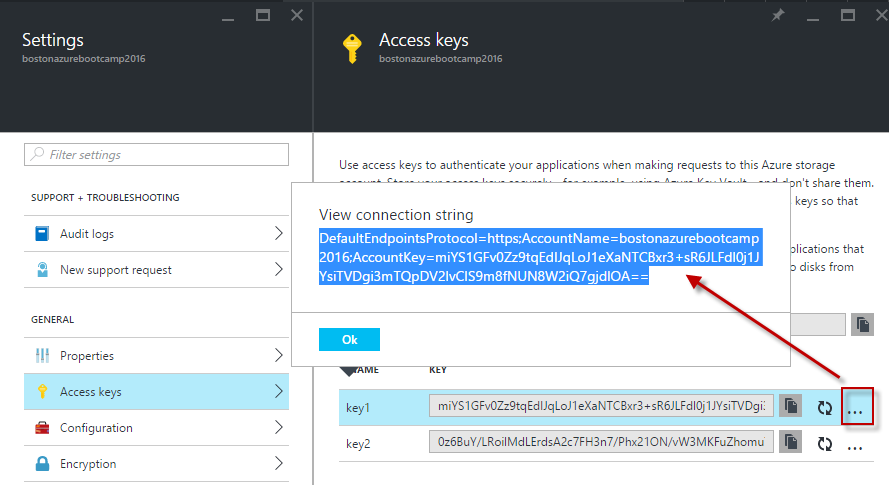
1. On the Create blade, select **Data + Storage ->** **Storage account**



1. On the **Create storage account** blade, enter a **unique name** for the storage account
2. Give the storage account a **resource name**
3. Select **Pin to dashboard** (this will save you a few steps later)
4. Click the **Create** button

Once the storage account creates, the panel should open for you. If not, you can click the storage account tile on the dashboard.

1. In the storage account panel -> **All Settings** (blade should be open). Select **Access Keys**



1. Click on the … beside the key1 and **copy the connection string** from the dialog to the **clipboard** (Ctrl + C)
2. Back in Visual Studio, open the **app.config** file
3. Add two connection strings named AzureWebJobsDashboard and AzureWebJobsStorage and set their connectionString to the value you copied from the portal.

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <configuration>    <connectionStrings>      <add name="AzureWebJobsDashboard" connectionString="DefaultEndpointsProtocol=https;AccountName=bostonazurebootcamp2016;AccountKey=miYS1GFv0Zz9tqEdIJqLoJ1eXaNTCBxr3+sR6JLFdl0j1JYsiTVDgi3mTQpDV2lvCIS9m8fNUN8W2iQ7gjdlOA==" />      <add name="AzureWebJobsStorage" connectionString="DefaultEndpointsProtocol=https;AccountName=bostonazurebootcamp2016;AccountKey=miYS1GFv0Zz9tqEdIJqLoJ1eXaNTCBxr3+sR6JLFdl0j1JYsiTVDgi3mTQpDV2lvCIS9m8fNUN8W2iQ7gjdlOA==" />    </connectionStrings> |

Make sure you use your storage account - the key I show above will be changed by the time you read this ☺

## Add a function to the WebJob to watch a queue for Orders and then save to Blob storage

You are now ready to create the WebJob functionality to use a QueueTrigger and a Blob binding. Since we want the blob to work with the Order type (not a Stream type or one of the other options) we will need to also create an implementation of the ICloudBlobStreamBinder to create the custom serialization/deserialization of an Order type.

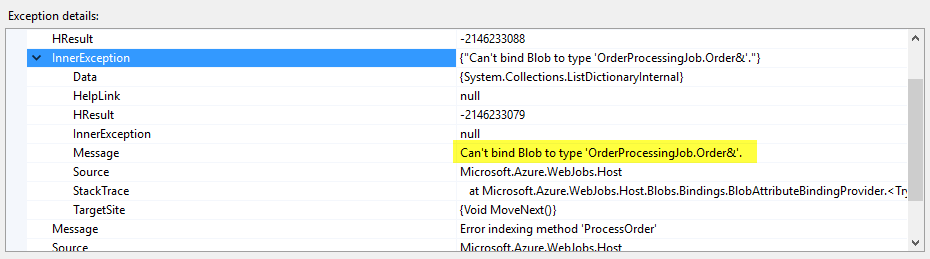
1. Open the **Program.cs** file and insert the following C#

|  |
| --- |
| using Microsoft.Azure.WebJobs;    namespace OrderProcessingJob  {      class Program      {          static void Main(string[] args)          {              var config = new JobHostConfiguration();                config.UseDevelopmentSettings();                var host = new JobHost(config);              host.RunAndBlock();          }      }  } |

1. In the Solution Explorer, right-click on the **OrderProcessingJob project** and select **Add -> Class**. Name it **Functions.cs** and click **Add**.
2. Open the **Functions** class and insert the following C#

|  |
| --- |
| using System.IO;  using Microsoft.Azure.WebJobs;    namespace OrderProcessingJob  {      public class Functions      {          public static void ProcessOrder(             [QueueTrigger("orders")] Order order,             [Blob("orders/{Id}")] out Order orderToSave,             TextWriter log             )          {              // Save the order in blob storage              orderToSave = order;                // log order processed          log.WriteLine($"Processed {order.Products.Count} products");          }      }  } |

If you try and run the console application now, you will get an exception:

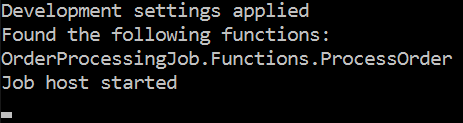


This is due to the Blob binding not knowing how we want the Order type serialized to the blob storage.

1. In the Solution Explorer, right-click on the **OrderProcessingJob project** and select **Add -> Class**. Name it **OrderBlobBinder.cs** and click **Add**.
2. Open the **OrderBlobBinder**class and insert the following C#

|  |
| --- |
| using System.IO;  using System.Threading;  using System.Threading.Tasks;  using Microsoft.Azure.WebJobs;  using Newtonsoft.Json;    namespace OrderProcessingJob  {      public class OrderBlobBinder : ICloudBlobStreamBinder<Order>      {          public Task<Order> ReadFromStreamAsync(Stream input,              CancellationToken cancellationToken)          {              using (StreamReader reader = new StreamReader(input))              {                  var jsonString = reader.ReadToEnd();                  var order = JsonConvert.DeserializeObject<Order>(jsonString);                  return Task.FromResult(order);              }          }            public Task WriteToStreamAsync(Order value, Stream output,              CancellationToken cancellationToken)          {              var serializer = new JsonSerializer();              using (var writer = new StreamWriter(output))              using (var jsonTextWriter = new JsonTextWriter(writer))              {                  serializer.Serialize(jsonTextWriter, value);              }              return Task.FromResult<object>(null);          }      }  } |

Now if you run the console application you will see the job starts up fine.



# Add a test function to verify the queue and blob logic works

We now need to create a function that will add an Order object to the orders queue so we can verify the WebJob functionality is working as designed so far. This new test function will use the NoAutomaticTrigger attribute to allow us to use the WebJobs SDK bindings without needing a trigger.

1. Open the **Functions** class and insert the following C#

|  |
| --- |
| [NoAutomaticTrigger]  public static void PutTestOrderInQueue(      [Queue("orders")] out Order testOrder,      TextWriter log)  {      testOrder = new Order()      {          Id = Guid.NewGuid().ToString("N"),          BuyerEmail = "<your email here>",          BuyerName = "<your name here>",          OrderPlacedUtc = DateTime.UtcNow,          Products = new List<Product>          {              new Product()              {                  Id = "Product1",                  Name = "Product #1",                  Price = 1.5,                  Quantity = 5              },              new Product()              {                  Id = "Product2",                  Name = "Product #2",                  Price = 2.1,                  Quantity = 2              }          }      };        // log message      log.WriteLine("Queued test order");  } |

1. You may need to add the following using statement:

|  |
| --- |
| using System; |

Now we need to add a call to invoke this function when the WebJob starts up

1. Open the **Program.cs** file and add the highlighted line to your code:

|  |
| --- |
| static void Main(string[] args)  {      var config = new JobHostConfiguration();        config.UseDevelopmentSettings();        var host = new JobHost(config);        host.Call(typeof(Functions).GetMethod("PutTestOrderInQueue"));        host.RunAndBlock();  } |

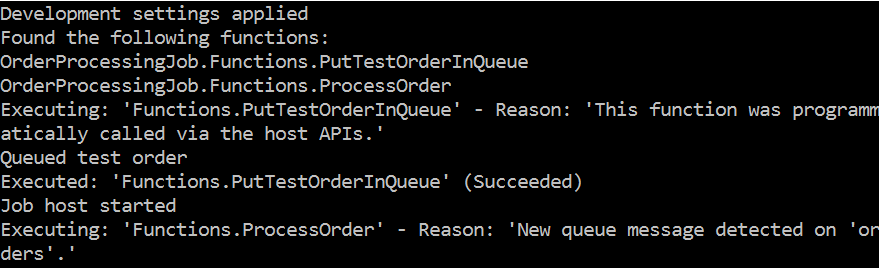
One more thing to do before testing it out:

1. Open the **Functions** class and put a breakpoint in the first line off the ProcessOrder method.

|  |
| --- |
| public static void ProcessOrder(      [QueueTrigger("orders")] Order order,      [Blob("orders/{Id}")] out Order orderToSave,      TextWriter log      )  {      // Save the order in blob storage      orderToSave = order;        // log order processed      log.WriteLine($"Processed {order.Products.Count} products");  } |

1. Run the application

Once the WebJob starts, you should see the logging in your console that looks like this:



The debugger should now be stopped in the ProcessOrder method.

1. Hit **F5** to **continue execution** of the method

Once the method has completed the console should add the following lines:



At this point the logic has completed, so you can stop the debugger.

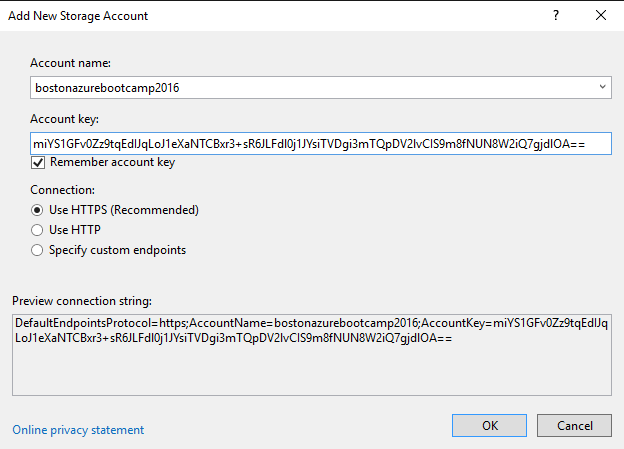
Verify the blob was saved to storage

1. Open **Server Explorer** and click on the **Azure** node. Login to your account if you need to
2. Locate the **Storage** node and look for your **WebJob’s storage account** you setup earlier

If you don’t see the storage account under the storage node:

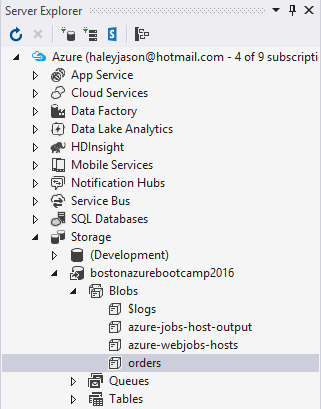
1. Right-click the **Storage node** -> **Attach External Storage**

This will show the Add New Storage Account dialog



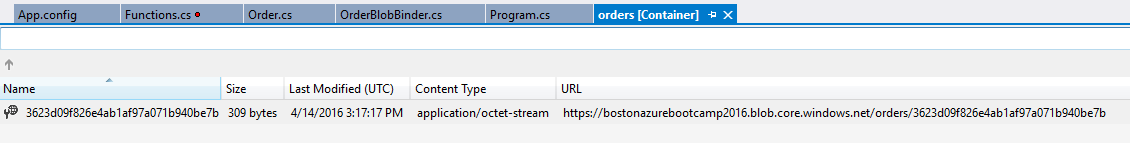
1. Enter your **storage account name** and **account key** (these are in the connection string you added to the app.config earlier).
2. Click Ok

Your storage account should now show.



1. Click on the **orders** blob container to see the contents.

You should see one item in the container that was uploaded recently.



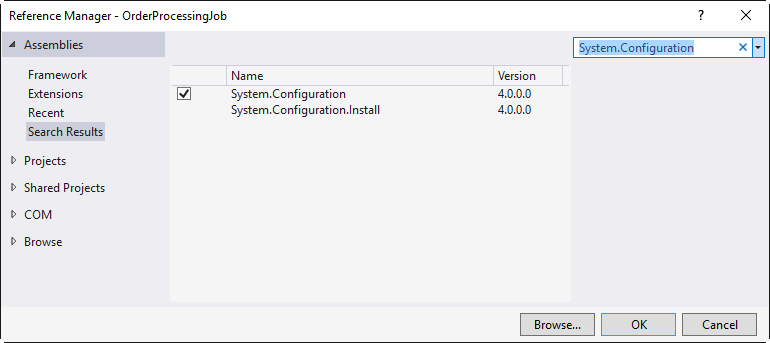
## Modify the logic to use a custom NameResolver

One thing that you may notice in the ProcessOrder function, is the QueueTrigger and Blob attributes have hard coded queue and container names. This can be limiting when you have multiple environments and may want those queue and/or blob containers to differ between the environments. The solution to this is to create a custom implementation of the INameResolver interface.

1. In the Solution Explorer, right-click on the **OrderProcessingJob project** and select **Add -> Class**. Name it **NameResolver.cs** and click **Add**.
2. Open the **NameResolver** class and insert the following C#

|  |
| --- |
| using System;  using System.Configuration;  using Microsoft.Azure.WebJobs;    namespace OrderProcessingJob  {      public class NameResolver : INameResolver      {          public string Resolve(string name)          {              var settingValue = ConfigurationManager.AppSettings[name];              if (string.IsNullOrEmpty(settingValue))              {                  settingValue = Environment.GetEnvironmentVariable(name);              }              return settingValue;          }      }  } |

1. In the Solutions Explorer, right-click on References -> Add Reference
2. Select Assemblies, enter System.Configuration in the search box and check the box to add the reference.



1. Click OK

Now that you have a custom name resolver, you need to configure it, add application settings and modify the code to use the name resolver.

1. Open the Program.cs file and add the highlighted line of C#

|  |
| --- |
| static void Main(string[] args)  {      var config = new JobHostConfiguration();        config.NameResolver = new NameResolver();        config.UseDevelopmentSettings();        var host = new JobHost(config);        host.Call(typeof(Functions).GetMethod("PutTestOrderInQueue"));        host.RunAndBlock();  } |

1. Open the Functions class and modify the attributes to use the special syntax that indicates what name to resolve as highlighted below:

|  |
| --- |
| public static void ProcessOrder(      [QueueTrigger("%OrdersQueueName%")] Order order,      [Blob("%OrdersContainerName%/{Id}")] out Order orderToSave,      TextWriter log      )  {      // Save the order in blob storage      orderToSave = order;        // log order processed      log.WriteLine($"Processed {order.Products.Count} products");  } |

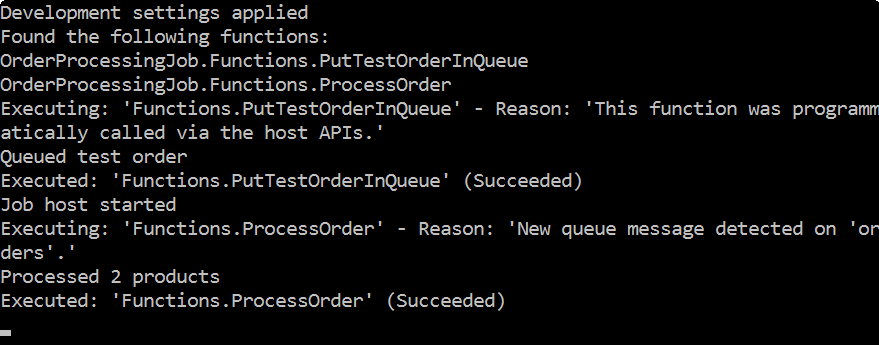
1. Open App.config and add an appSettings section and two keys named:

* OrdersQueueName
* OrdersContainerName

|  |
| --- |
| <configuration>    <connectionStrings>      <add name="AzureWebJobsDashboard" connectionString="…" />      <add name="AzureWebJobsStorage" connectionString="…" />    </connectionStrings>    <appSettings>      <add key="OrdersQueueName" value="orders" />      <add key="OrdersContainerName" value="orders" />    </appSettings> |

1. Run the application

You should get the same console output as before – but this time the code is getting the queue and container names from the configuration file.



## Modify the logic to use a custom trace writer

Sometimes you may find you want to customize your logging. For instance, you may want to log certain things to a database or log4net or some other location besides the WebJobs dashboard.

Logging can be customized by creating and configuring your own implementation of a TraceWriter.

For this lab, we will just create a trace writer that writes to the console in a different color to differentiate the messages coming from our logger vs. the WebJobs logging.

1. In the Solution Explorer, right-click on the **OrderProcessingJob project** and select **Add -> Class**. Name it **ColorConsoleTraceWriter.cs** and click **Add**.
2. Open the **ColorConsoleTraceWriter** class and insert the following C#

|  |
| --- |
| using System;  using System.Diagnostics;  using Microsoft.Azure.WebJobs.Host;    namespace OrderProcessingJob  {      public class ColorConsoleTraceWriter : TraceWriter      {          public ColorConsoleTraceWriter(TraceLevel level)              : base(level)          {          }            public override void Trace(TraceEvent traceEvent)          {              var holdColor = Console.ForegroundColor;                Console.ForegroundColor = ConsoleColor.Yellow;              Console.WriteLine(traceEvent.Message);                Console.ForegroundColor = holdColor;          }      }  } |

1. Open the Program.cs file and add in the highlighted line of C#

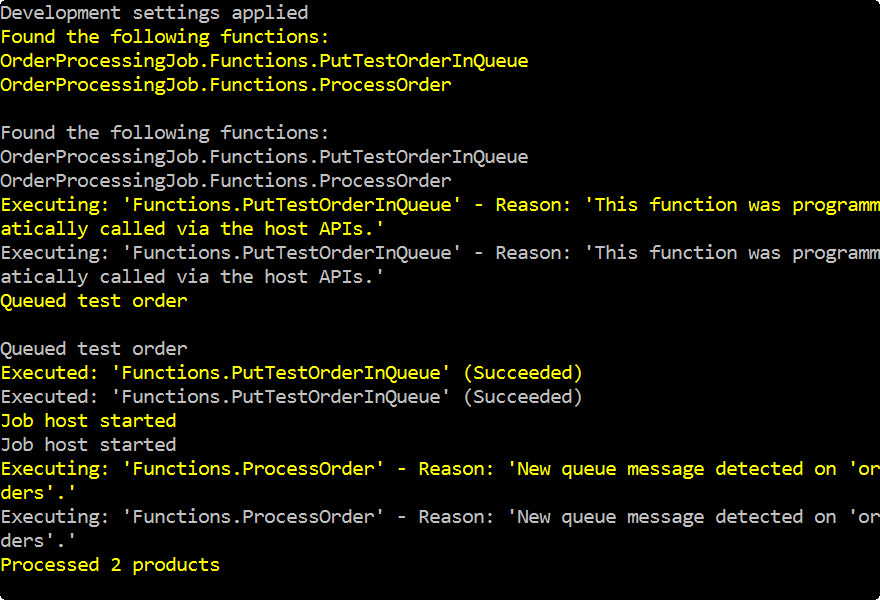
|  |
| --- |
| static void Main(string[] args)  {      var config = new JobHostConfiguration();        config.NameResolver = new NameResolver();      config.Tracing.Tracers.Add(new ColorConsoleTraceWriter(TraceLevel.Info));        config.UseDevelopmentSettings();        var host = new JobHost(config);      host.Call(typeof(Functions).GetMethod("PutTestOrderInQueue"));      host.RunAndBlock();  } |

1. You may need to add the following using statement:

|  |
| --- |
| using System.Diagnostics; |

1. Run the application

You should now see duplicated messages in the console in yellow – those are the ones being logged by the custom trace writer … which means in real life you could be putting those into the database or some other location instead of the console window.



## Add functionality to Save the Order Products to a Storage Table

Sometimes you want to save things in an Azure storage table. When you use the WebJobs SDK, it is just as easy as saving the Order to blob storage. In this exercise we’ll add a Table binding to the ProcessOrder function to save the individual products on the Order to a table.

1. In the Solution Explorer, right-click on the **OrderProcessingJob project** and select **Add -> Class**. Name it **ProductsOrderedEntity.cs** and click **Add**.
2. Open the **ProductsOrderedEntity** class and insert the following C#

|  |
| --- |
| using System;  using Microsoft.WindowsAzure.Storage.Table;    namespace OrderProcessingJob  {      public class ProductsOrderedEntity : TableEntity      {          public ProductsOrderedEntity()          { }            public ProductsOrderedEntity(string orderId,              Product product,              DateTime orderedUtc)          {              SetRowKey(orderId, product, orderedUtc);                OrderId = orderId;              Id = product.Id;              Name = product.Name;              Quantity = product.Quantity;              Price = product.Price;          }            public string OrderId { get; set; }          public string Id { get; set; }          public string Name { get; set; }          public int Quantity { get; set; }          public double Price { get; set; }          public void SetRowKey(string orderId,              Product product,              DateTime orderedUtc)          {              // Use a reverse data schema to keep rows sorted chronologically              long ticks = DateTimeOffset.MaxValue.Ticks - orderedUtc.Ticks;                RowKey = $"{ticks}\_{orderId}\_{product.Id}";              PartitionKey = orderId;          }      }  } |

Since we created a customer name resolver, we should add a new application setting for the name of the storage table.

1. Open app.config and add the highlighted app setting:

|  |
| --- |
| <appSettings>      <add key="OrdersQueueName" value="orders" />      <add key="OrdersContainerName" value="orders" />      <add key="ProductsTableName" value="OrderedProducts"/>    </appSettings> |

1. Open the Functions class and locate the ProcessOrder method

With the table, we want to be able to add multiple rows inside the ProcessOrder method. We do this by using a parameter that is an ICollector.

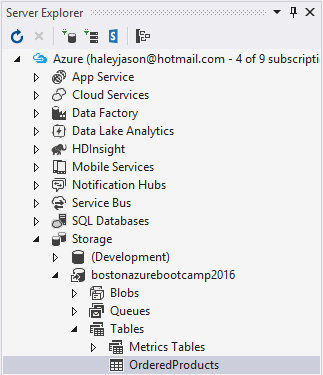
1. Add the highlighted changes to the ProcessOrder method:

|  |
| --- |
| public static void ProcessOrder(      [QueueTrigger("%OrdersQueueName%")] Order order,      [Blob("%OrdersContainerName%/{Id}")] out Order orderToSave,      [Table("%ProductsTableName%")] ICollector<ProductsOrderedEntity> productsOrderedTable,      TextWriter log      )  {      // Save the order in blob storage      orderToSave = order;        // Save the ordered product quantities to table storage      foreach (var product in order.Products)      {          productsOrderedTable.Add(              new ProductsOrderedEntity(order.Id, product, order.OrderPlacedUtc));      }        // log order processed      log.WriteLine($"Processed {order.Products.Count} products");  } |

1. Build and Run the application

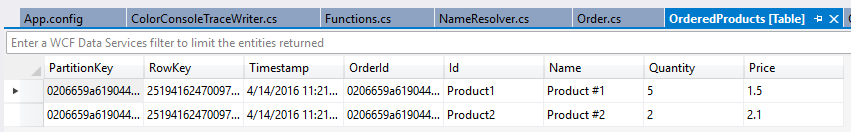
The console output will look the same as last time. In order to verify the table was created and rows were added, we need to look at Azure storage.

1. Open the Server Explorer and select Azure -> Storage. Find your storage account for the WebJob and expand the Tables node underneath. You should see the OrderedProducts table



1. Double click the OrderedProducts table name.

This will open the ProductsTable, where you should see two rows.



## Add Functionality to Send an Email Once the Order is Processed

In this exercise you will use the SendGrid binding from the WebJobs SDK Extensions to send an email inside the WebJob.

The assumption is that you already have a SendGridApiKey by this point. If you do not, you can go here to sign up for a free one: <https://sendgrid.com/free>

1. Open app.config and create a new setting to hold your SendGridApiKey and an email to have in the from address.

|  |
| --- |
| <appSettings>      <add key="OrdersQueueName" value="orders" />      <add key="OrdersContainerName" value="orders" />      <add key="ProductsTableName" value="OrderedProducts"/>      <add key="AdminEmail" value="<your email here>"/>      <add key="AzureWebJobsSendGridApiKey" value="…"/>  </appSettings> |

Now you need to configure the SendGrid extension.

1. Open Program.cs and add the highlighted changes to your C#

|  |
| --- |
| using System.Configuration;  using System.Diagnostics;  using System.Net.Mail;  using Microsoft.Azure.WebJobs;  using Microsoft.Azure.WebJobs.Extensions.SendGrid;    namespace OrderProcessingJob  {      class Program      {          static void Main(string[] args)          {              var config = new JobHostConfiguration();                config.NameResolver = new NameResolver();              config.Tracing.Tracers.Add(new ColorConsoleTraceWriter(TraceLevel.Info)  );  config.UseSendGrid(new SendGridConfiguration()              {                  FromAddress = new MailAddress(                      ConfigurationManager.AppSettings["AdminEmail"])              });                config.UseDevelopmentSettings();                var host = new JobHost(config);              host.Call(typeof(Functions).GetMethod("PutTestOrderInQueue"));              host.RunAndBlock();          }      }  } |

1. Open the Functions class, and add the highlighted C#

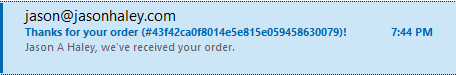
|  |
| --- |
| public static void ProcessOrder(      [QueueTrigger("%OrdersQueueName%")] Order order,      [Blob("%OrdersContainerName%/{Id}")] out Order orderToSave,      [Table("%ProductsTableName%")] ICollector<ProductsOrderedEntity> productsOrderedTable,      [SendGrid] SendGridMessage message,      TextWriter log      )  {      // Save the order in blob storage      orderToSave = order;        // Save the ordered product quantities to table storage      foreach (var product in order.Products)      {          productsOrderedTable.Add(              new ProductsOrderedEntity(order.Id, product, order.OrderPlacedUtc));      }        // Send and email to the buyer      message.AddTo(order.BuyerEmail);      message.Subject = $"Thanks for your order (#{order.Id})!";      message.Text = $"{order.BuyerName}, we've received your order.";        // log order processed      log.WriteLine($"Processed {order.Products.Count} products");  } |

1. You will need to make sure you have the following using statement as well:

|  |
| --- |
| using SendGrid; |

1. In the PutTestOrderInQueue method of the Functions class, make sure your BuyerEmail and BuyerName have good values (like your name and your email address).
2. Run the application

The console output will again be the same, however in a minute or two you should receive and email.



## Add a function to handle poison messages on the Orders queue to send out an email

Sometimes messages can be full of unexpected data and cause a function to fail. This is managed in WebJobs by moving the message to a poison queue once the message has hit the MaxDequeueCount attemps. The poison queue has the same name as the other queue, but with a ‘-poison’ appended to it. Example: ‘orders-poison’

In this exercise we will create a function to handle and poison messages and also write a function to test it.

1. Open the Functions class and add the following C# to it:

|  |
| --- |
| public static void HandlePoisonMessage(      [QueueTrigger("%OrdersQueueName%-poison")] string orderJson,      [SendGrid] SendGridMessage message,      TextWriter log)  {      // Send and email to the admin      message.AddTo(ConfigurationManager.AppSettings["AdminEmail"]);      message.Subject = "Order Failed Notification";      message.Text = orderJson;        // log poison message      log.WriteLine($"Poison Order: {orderJson}");  } |

1. Add the following using statement to the top:

|  |
| --- |
| using System.Configuration; |

This method will watch for messages on the orders-poison queue. When a message gets added to it, the function will send an email with text contents of the poison message.

1. Add the following method to the Functions class

|  |
| --- |
| [NoAutomaticTrigger]  public static void PutPoisonOrderInQueue(      [Queue("orders")] out Order testOrder,      TextWriter log)  {      testOrder = new Order()      {          Id = Guid.NewGuid().ToString("N"),          BuyerEmail = "<your email here>",          BuyerName = "<your name here>",          OrderPlacedUtc = DateTime.UtcNow      };        // log message      log.WriteLine("Queued order");  } |

This Order doesn’t contain a Products list, which will cause errors and the removal of the message after 5 attempts it will get moved to the poison queue and run the method we wrote earlier.

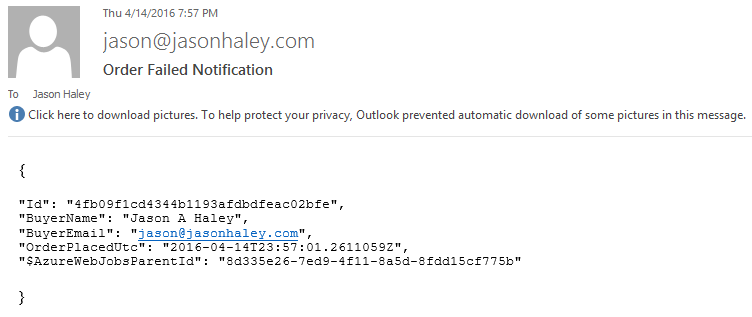
1. Open Program.cs and add the highlighted line to the Main method:

|  |
| --- |
| static void Main(string[] args)  {      var config = new JobHostConfiguration();        config.NameResolver = new NameResolver();      config.Tracing.Tracers.Add(new ColorConsoleTraceWriter(TraceLevel.Info));      config.UseSendGrid(new SendGridConfiguration()      {          FromAddress = new MailAddress(                  ConfigurationManager.AppSettings["AdminEmail"])      });        config.UseDevelopmentSettings();        var host = new JobHost(config);      host.Call(typeof(Functions).GetMethod("PutTestOrderInQueue"));      host.Call(typeof(Functions).GetMethod("PutPoisonOrderInQueue"));      host.RunAndBlock();  } |

This new line will invoke the test method you just created.

1. Run the application.

This time you should see quite a bit of red messages and eventually an email will get sent.



## Add a function that emails when a certain number of error have happened

In the last exercise, we dealt with the poison message scenario. In this exercise we will create a function that will use the ErrorTrigger to cause a function to run once a certain error threshold has been reached – in our case 10 errors in 5 minutes.

1. Open the Functions class and add the following C# method to it:

|  |
| --- |
| public static void HandleErrors(      [ErrorTrigger("00:05:00", 10)] TraceFilter filter,      [SendGrid] ref SendGridMessage message,      TextWriter log)  {      // Send and email to the admin      message.AddTo(ConfigurationManager.AppSettings["AdminEmail"]);      message.Subject = "WebJob Error Notification";      message.Text = filter.GetDetailedMessage(5);        // log last 5 detailed errors to the Dashboard      log.WriteLine(filter.GetDetailedMessage(5));    } |

This function will execute when 10 errors have been reached in a 5 minute period of time, then send an email with the last 5 errors to the administrator.

1. Add the following test method to the Functions class:

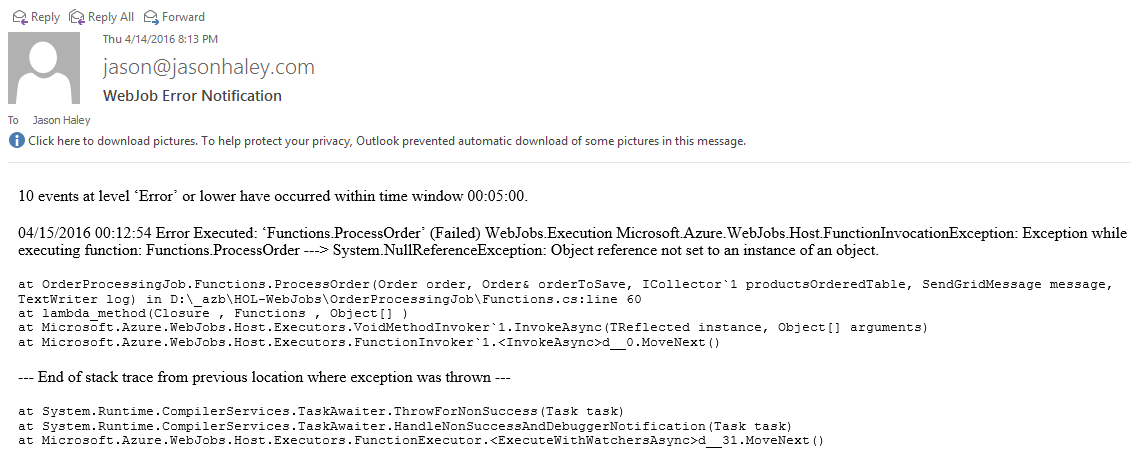
|  |
| --- |
| [NoAutomaticTrigger]  public static void PutTwoPoisonOrdersInQueue(      [Queue("orders")] ICollector<Order> testOrders,      TextWriter log)  {      for (int i = 0; i < 2; i++)      {          testOrders.Add(new Order()          {              Id = Guid.NewGuid().ToString("N"),              BuyerEmail = "<your email here>",              BuyerName = "<your name here>",              OrderPlacedUtc = DateTime.UtcNow          });      }        // log message      log.WriteLine("Queued order");  } |

1. Open Program.cs and add the highlighted changes

|  |
| --- |
| static void Main(string[] args)  {      var config = new JobHostConfiguration();        config.NameResolver = new NameResolver();      config.Tracing.Tracers.Add(new ColorConsoleTraceWriter(TraceLevel.Info));      config.UseSendGrid(new SendGridConfiguration()      {          FromAddress = new MailAddress(                  ConfigurationManager.AppSettings["AdminEmail"])      });      config.UseCore();        config.UseDevelopmentSettings();        var host = new JobHost(config);      host.Call(typeof(Functions).GetMethod("PutTestOrderInQueue"));      host.Call(typeof(Functions).GetMethod("PutPoisonOrderInQueue"));      host.Call(typeof(Functions).GetMethod("PutTwoPoisonOrdersInQueue"));      host.RunAndBlock();  } |

1. Run the application

Eventually you will start getting emails from both the poison queue and the error handler. Once you’ve proven the error handler works, you can stop the debugging.



## Deploy the WebJob to Azure

Now that you have a WebJob and have proven the functionality works by testing it locally, it is time to deploy it to Azure.

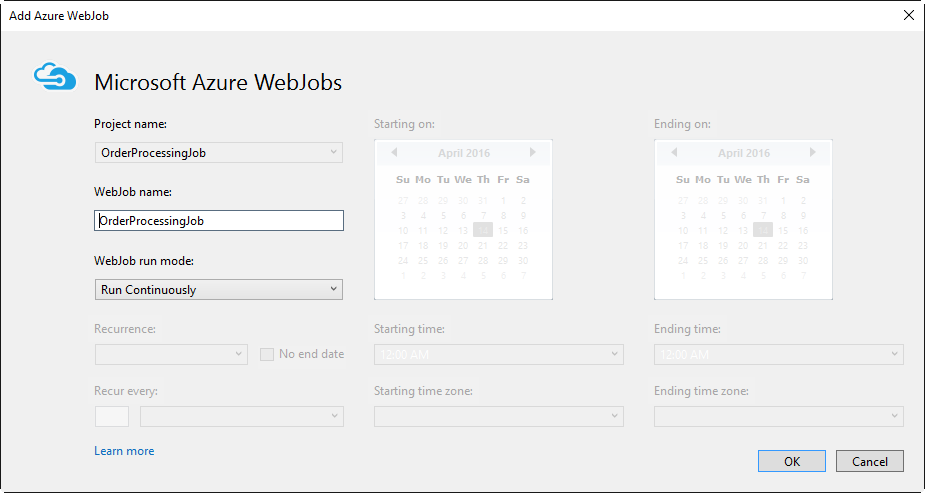
Before deploying it to Azure, you need to disable the testing methods.

1. Open Program.cs and delete or comment out the highlighted lines:

|  |
| --- |
| static void Main(string[] args)  {      var config = new JobHostConfiguration();        config.NameResolver = new NameResolver();      config.Tracing.Tracers.Add(new ColorConsoleTraceWriter(TraceLevel.Info));      config.UseSendGrid(new SendGridConfiguration()      {          FromAddress = new MailAddress(                  ConfigurationManager.AppSettings["AdminEmail"])      });      config.UseCore();        config.UseDevelopmentSettings();        var host = new JobHost(config);      //host.Call(typeof(Functions).GetMethod("PutTestOrderInQueue"));      //host.Call(typeof(Functions).GetMethod("PutPoisonOrderInQueue"));      //host.Call(typeof(Functions).GetMethod("PutTwoPoisonOrdersInQueue"));      host.RunAndBlock();  } |

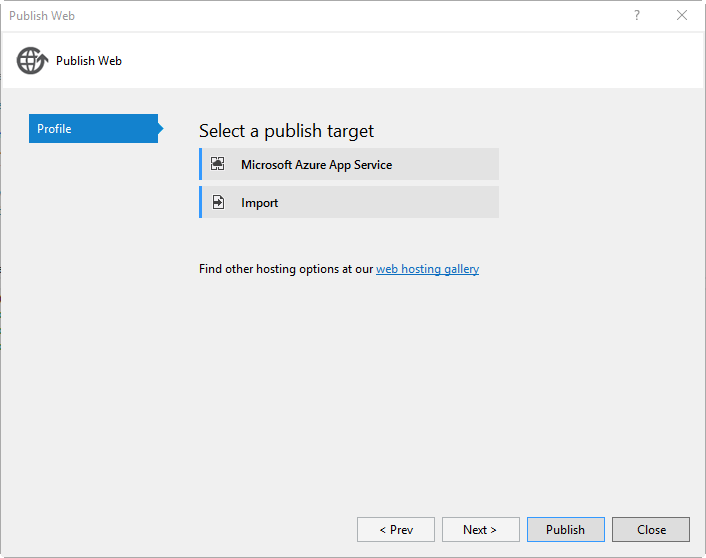
1. Right-click the OrderProcessingJob and select Publish as Azure Web Job…

This will open the Add Azure WebJob dialog

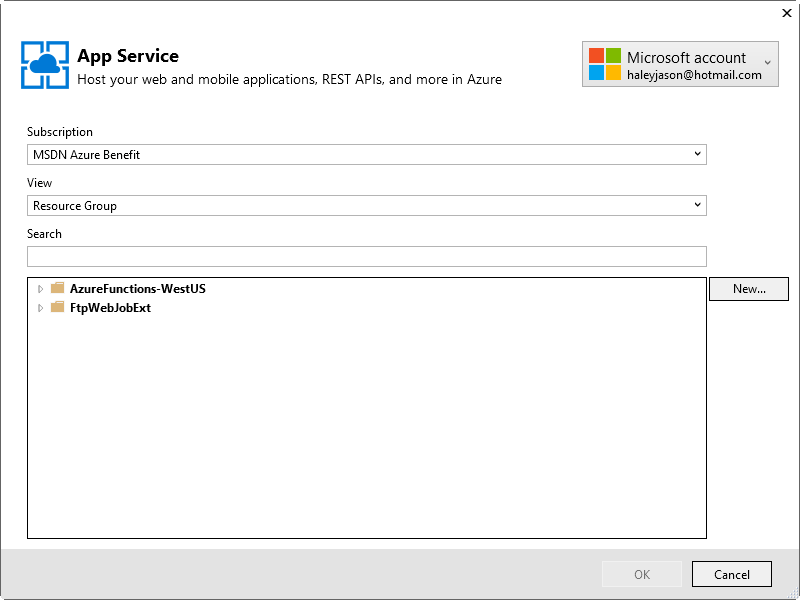


1. Click OK

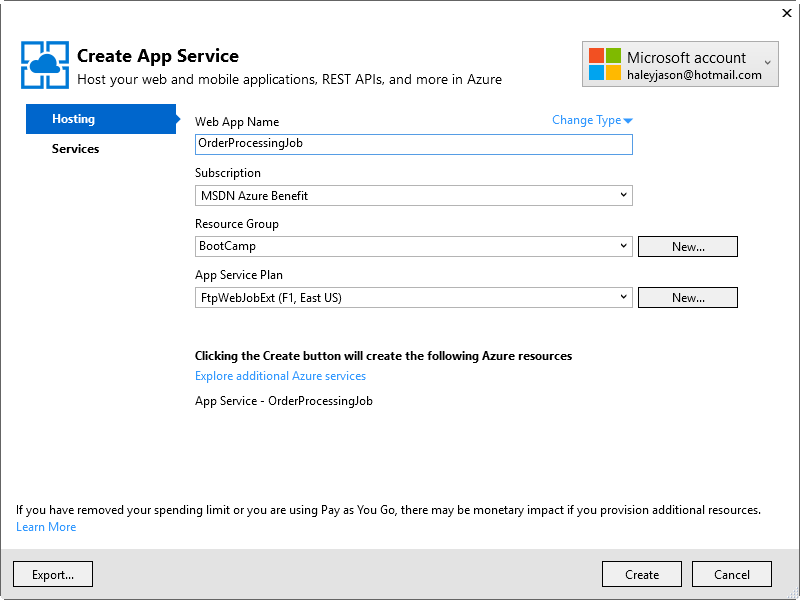
This will open the Publish Web dialog



1. Click the Microsoft Azure App Service button



1. Select your subscription from the dropdown, click the New… button on the right

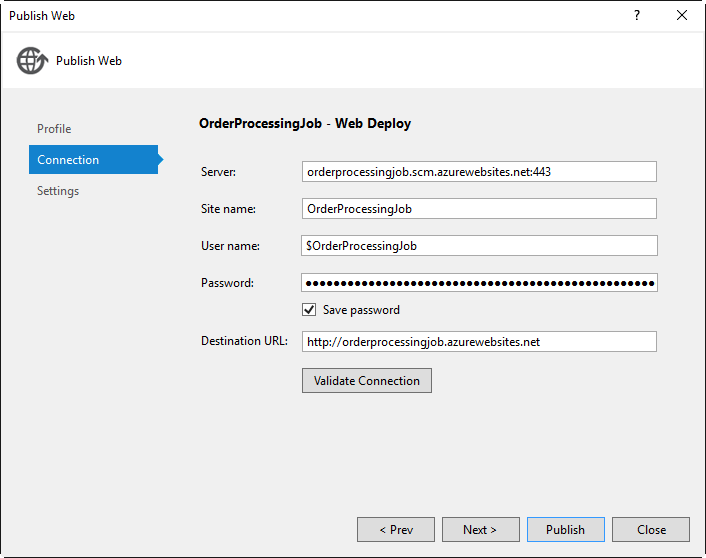


1. Fill in your Hosting details:

* Web App Name: OrderProcessingJob
* Subscription:
* Resource Group: select an existing or click the New button to create a new one
* App Service Plan: select an existing or click the New button to create a new one

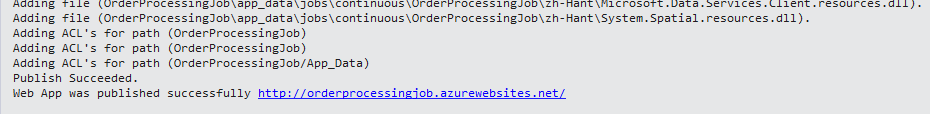
1. Click Create

The dialog will go out to Azure and create an App Service to deploy the WebJob to. Once it is ready, the Publish Web dialog will show with all the Connection information populated.



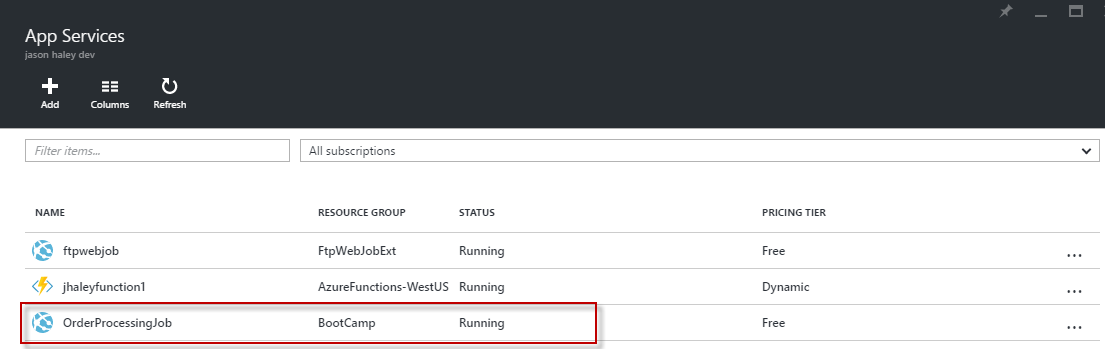
1. Click Publish

This will package and deploy the webjob out to the site.



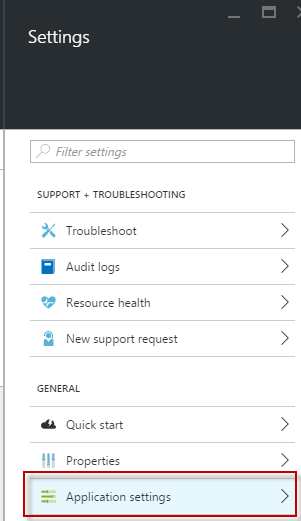
1. Open the Azure portal (<https://portal.azure.com>)
2. Select App Services from the menu

You should see your app service in the listing

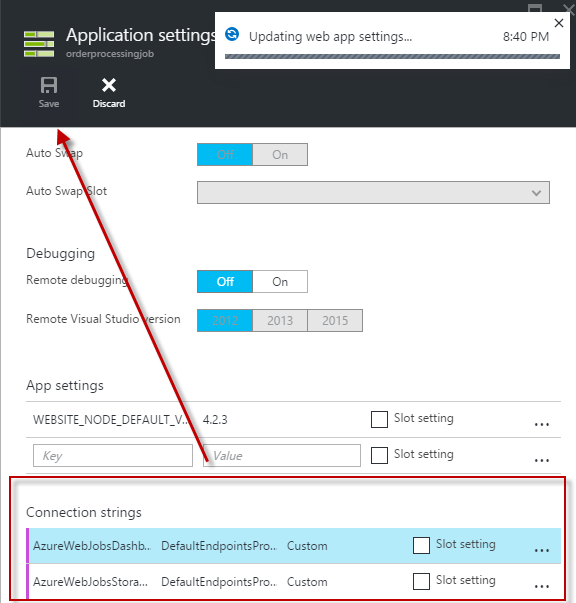


The WebJobs dashboard needs the two app settings that you added to your config file added to the website in order to open up all the functionality it provides.

1. Select the App Service. Once the Web app panel is open and the Settings blade, scroll to the Application settings menu item and click it.

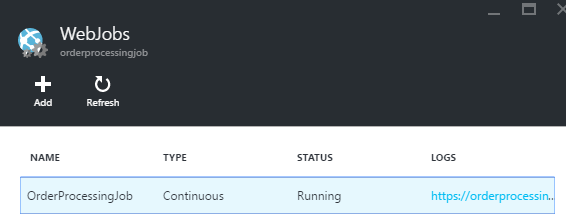


1. Scroll to the Connection Strings area of the Application Settings panel
2. In Visual Studio, open the app.config file and copy the AzureWebJobsDashboard and AzureWebJobsStorage over to Connection Strings in the panel.
3. Click the Save button at the top



1. In the Settings menu, scroll down to the WebJobs menu and click it.

This will show the list of WebJobs running.



1. Double click the link in the logs column.

This will take you to the WebJobs dashboard, where you can navigate and see all the logging information for the different runs of the WebJobs functions and their status.

