# Azure Discovery Days 2019

## Data Analytics & Near Real Time Intelligence with Azure - Hands-On Lab Guide

## Lab 3: Stream Enrichment

### Summary

In this hands-on lab, you will:

1. Set up two stream ingestion endpoints
2. Set up a streaming event simulator that will send events to the first stream ingestion endpoint
3. Enrich the stream with insights from a pre-trained AI Cognitive Service
4. Send the enriched stream on to the second stream ingestion endpoint

### About this Lab

The streaming event simulator is meant to simulate the flow of data from taxis in the future, to include both trip data as well as user feedback, which will include both a numerical score (good/bad) as well as free-form comments about the trip.

### References

### General Notes

### Architecture for this Lab

The tasks in this lab cover the following components of the overall architecture.



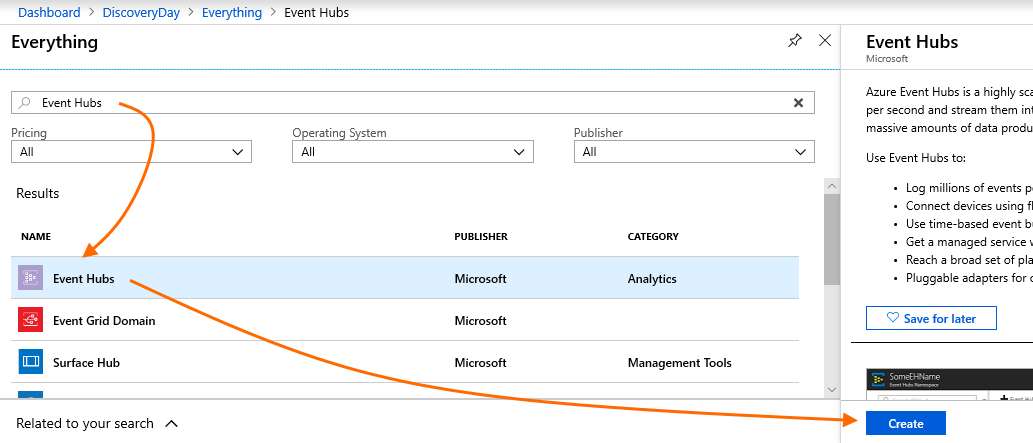
### Task 1 – Set up two stream ingestion endpoints

In this lab, we will use Azure Event Hubs for stream ingestion.

The first stream ingestion endpoint will receive inbound trip messages from taxi devices. The stream will then be enriched (see Task 2), and then forwarded to a second stream ingestion endpoint. (Lab 4 includes further work with the enriched stream after the second stream ingestion endpoint.)

Begin in the Resource Group where you have been working so far, and click “+ Add”, similarly to previous deployments, to deploy new resources.

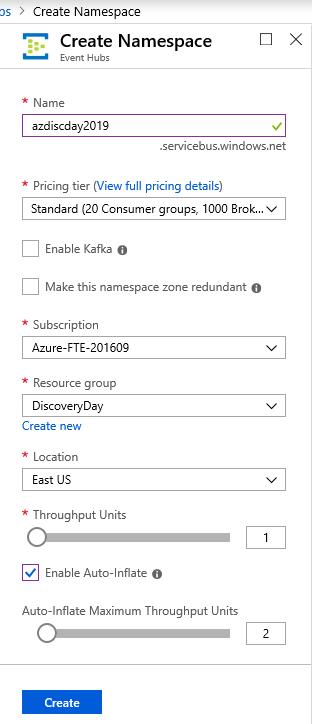
In the search box, type “Event Hubs” followed by Enter. Click on the Event Hubs entry, then “Create” on its product blade.



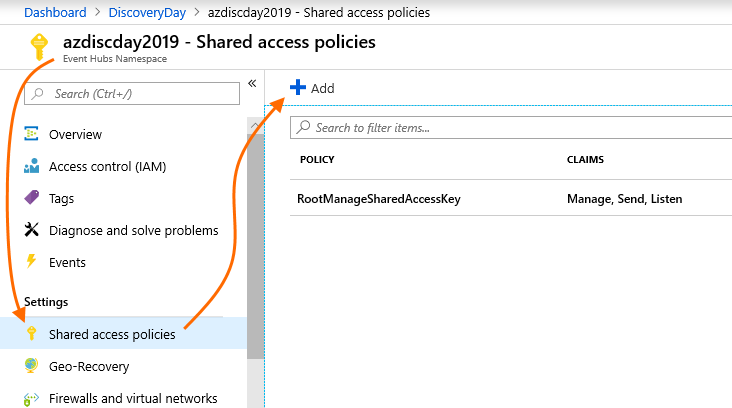
Provide appropriate values on the blade to create an Event Hubs Namespace. Specifically:

* Pricing Tier: Standard
* Subscription, Resource Group and Location: choose the ones you have been using so far.
* Throughput Units (TUs):
  + Set to 1
  + Enable Auto-Inflate: yes (checked)
  + Auto-Inflate Maximum TUs: 2
  + Each TU provides 1MB/sec or 1,000 events/sec ingress. These settings will be ample for this lab.

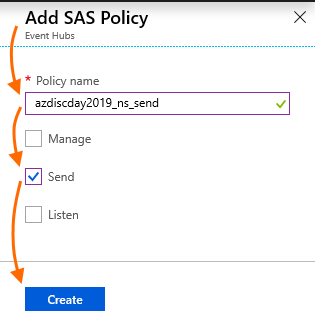
Then click Create.



When deployment completes, click on the new Event Hubs Namespace resource in your Resource Group. Then click on “Shared access policies”, and “+ Add” to add a new policy.

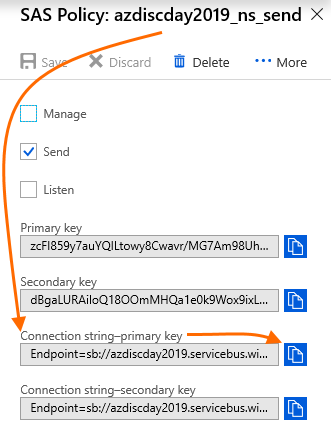


On the create blade, provide an appropriate name for the policy and ensure only “Send” is checked, then click “Create”.



This policy will be used by the taxi device simulator to send in messages; it does not need “Manage” or “Listen” capabilities (as we do not want our taxi devices to be able to manage our stream ingestion endpoints, or access sent-in messages).

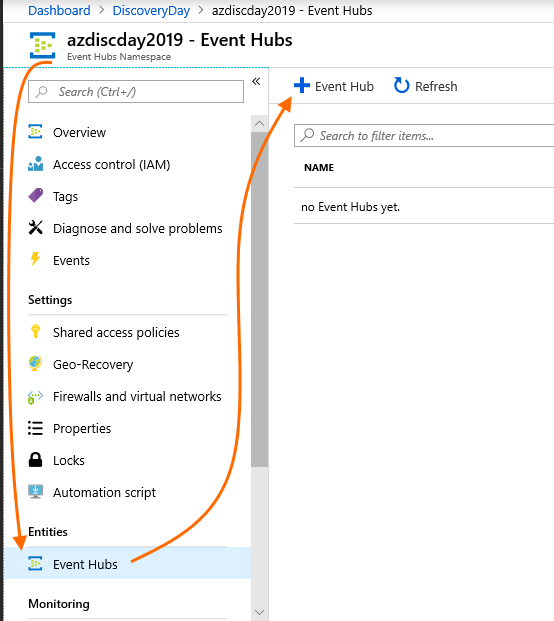
After creation completes, you will see the new policy in the Shared access policies view where you clicked “+Add”. Click on the new policy to view its properties. Copy one of its connection strings to a scratch area; you will need this later to configure the device simulator.



Close the SAS Policy blade to return to the Event Hubs Namespace view.

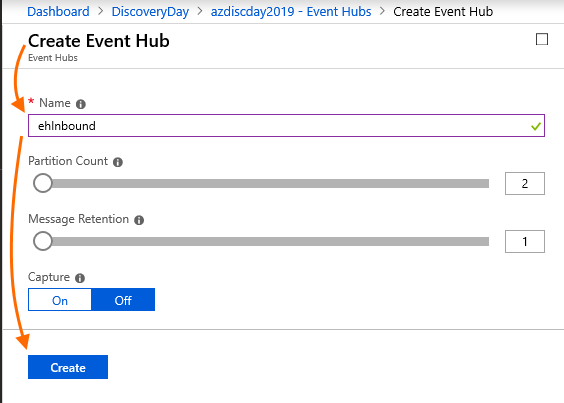
You will now create two Event Hubs in this Event Hubs Namespace. This first Event Hub will be the endpoint to which the taxi device simulator will send messages.

To start, click on “Entities/Event Hubs”. On the Event Hubs view, click “+ Event Hub”.

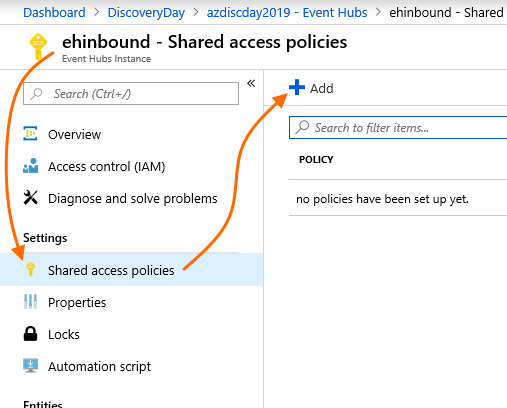


In the “Create Event Hub” view, provide a name for this Event Hub. Other inputs are good with their default values.

Optionally, you can configure Capture so that inbound messages, in addition to being routed to the Azure Function you will create later in this lab, are also stored in Azure Storage. However, Capture is not required for this or other labs. Then click “Create”.

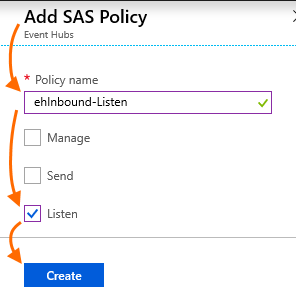


After the Event Hub is created, you will see it in the Event Hubs list on the view where you just clicked “+ Event Hub”. Now click the Event Hub you just created, then click “Shared access policies”, then “+ Add”.



Specify a name for the policy and check only “Listen”, then click “Create”.

This policy will allow the Azure Function you will create in a later task to listen for event messages sent to this endpoint by the taxi device simulator. As with the namespace-level policy you created earlier, in this case the Azure Function will only need to Listen for messages – it does not need to Send messages to this endpoint, nor does it need to Manage it.

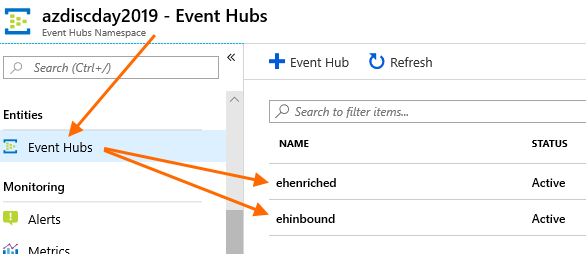


After the policy is created, return to the Event Hubs Namespace “Event Hubs” view, where your first Event Hub is shown in the list.

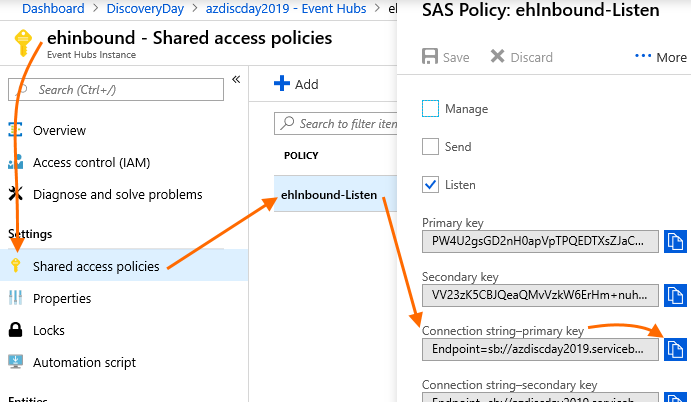
Now, create another Event Hub. This will be the second messaging endpoint shown in the architecture diagram (see page 1 of this document). The Azure Function you will create in a later task will send the enriched message stream to this second Event Hub.

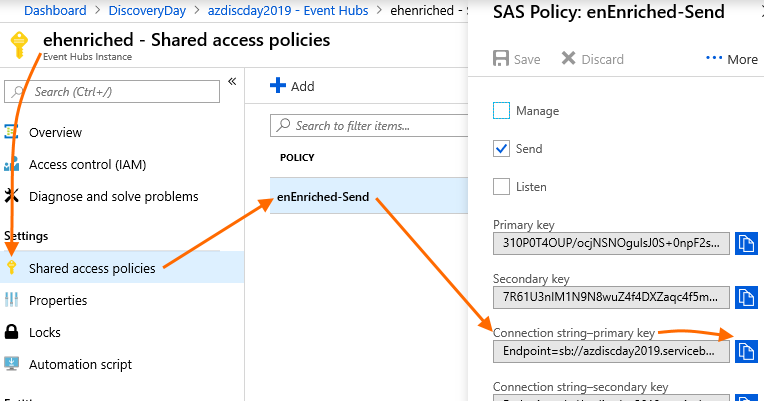
**Proceed as with the first Event Hub, but with this important difference:** create a Shared access policy for this second Event Hub, but with Send permission only. The Azure Function will use this policy to send enriched messages onward, and Listen and Manage are not needed.

When you are done, you should see two Event Hubs listed in the Event Hubs Namespace list. Each should have one SAS Policy.



To conclude this task, click into each of the two Event Hubs. In each Event Hub, click “Shared access policies”. Then click the policy you created to show its properties. Copy the policy’s connection string and save it in a scratch pad area for later use (or, of course, you can always come back to this view when you need the connection string).





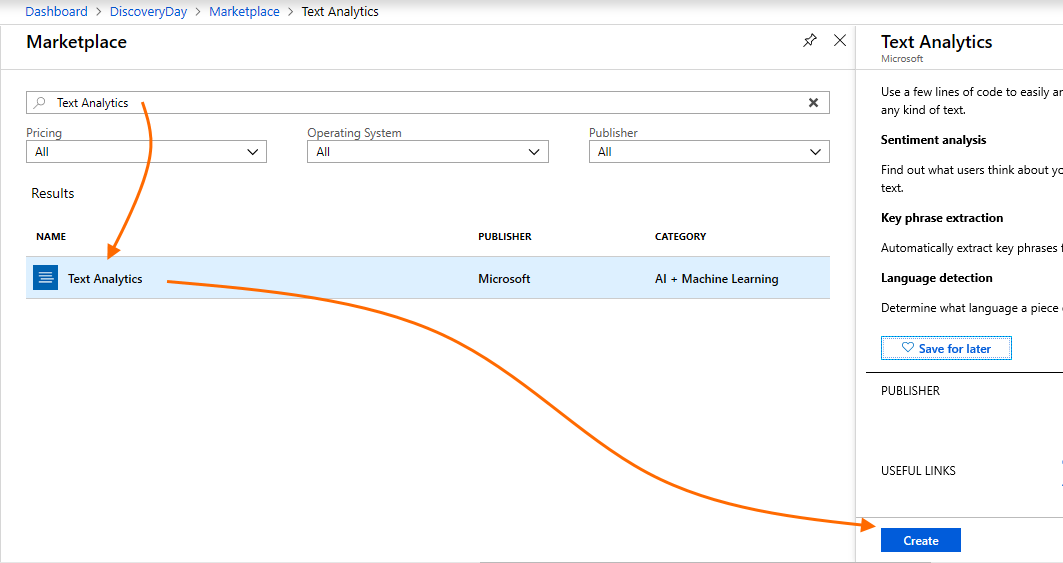
In summary, when you have successfully deployed the following, then this task is complete:

1. Event Hubs Namespace
   1. Send Shared Access Policy for the Namespace, to be used by the taxi device simulator to send in messages.
2. Event Hub for inbound taxi device messages
   1. Listen Shared Access Policy for the Event Hub, to be used by the Azure Function to listen for incoming messages.
3. Event Hub to forward enriched taxi device messages
   1. Send Shared Access Policy for the Event Hub, to be used by the Azure Function to send on messages it has enriched.

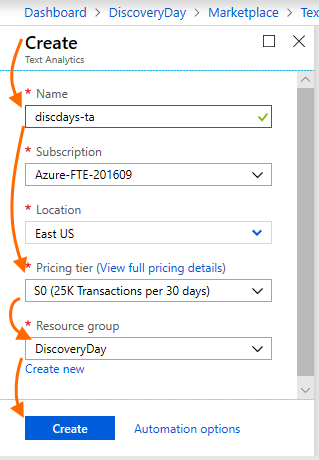
### Task 2 – Deploy Text Analytics Cognitive Service

In this task, you will deploy an Azure Text Analytics Cognitive Service. This will be used by the Azure Function (which you will deploy in task 3) to perform text analytics on customer comments that are part of the messages sent in by the taxi devices.

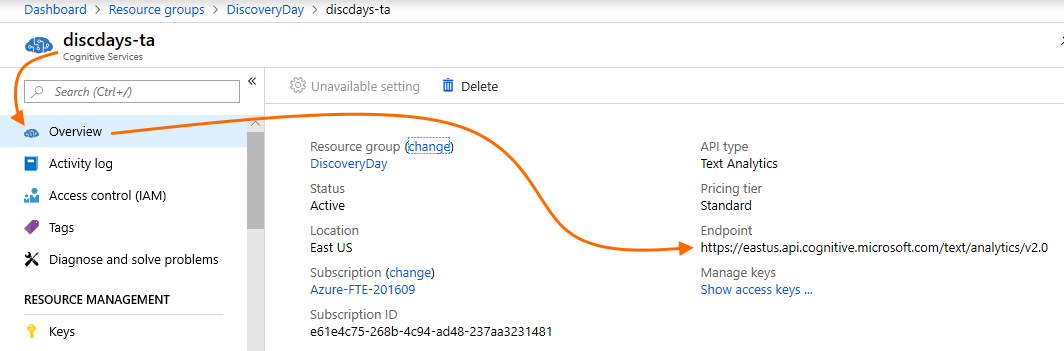
In your Resource Group, click “+ Add” again to add a new resource. Type “Text Analytics” in the search box, then Enter.

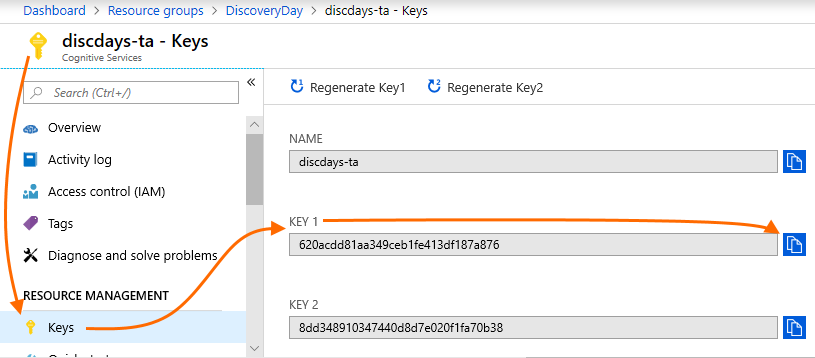


On the Create blade, provide a name. Select the S0 pricing tier, and ensure that your Resource Group is selected. Then click “Create”.



After creation completes, locate the new cognitive service resource in your Resource Group and click on it. Begin on the “Overview” blade. Locate the Endpoint and copy its value to a scratchpad area. Then click on the “Keys” blade and copy either API key value to a scratchpad area. You will need both pieces of information when creating the Azure Function that will use this cognitive service.





When you have obtained both the API Endpoint URL and the API key, this task is complete. Please return to your Resource Group for the next task.

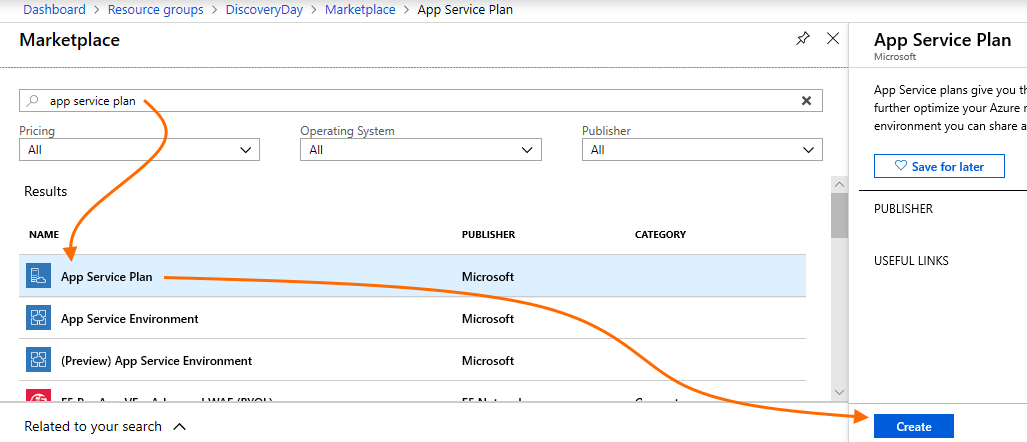
### Task 3 – Deploy Azure Function to Process and Enrich Taxi Messages

In this task, you will deploy the Azure Function shown in the architecture diagram on page 1.

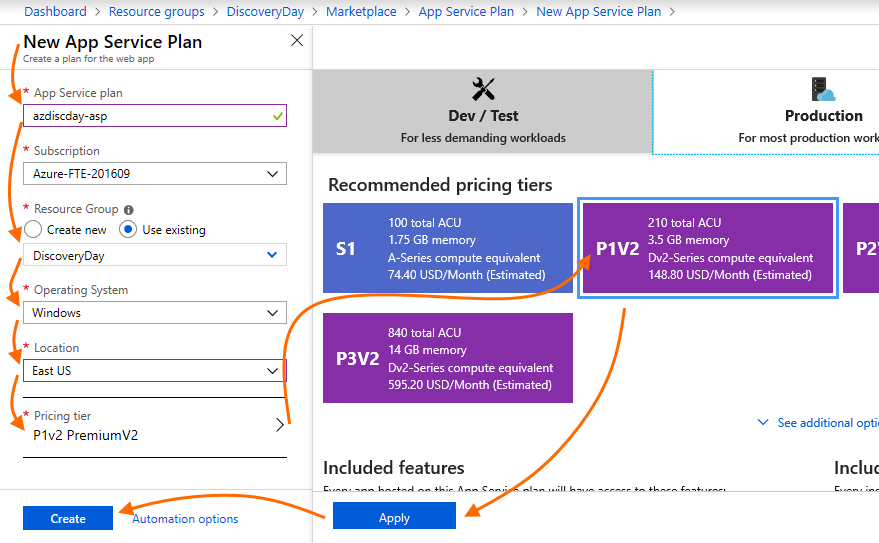
This Function will be triggered to run every time a taxi device message is received by the first Event Hub you deployed in task 1. It will then get the customer’s comments from the message, and run those comments through the text analytics cognitive service you created in task 2. Then, the text analytics results will be added to the taxi message – i.e. the inbound message is enriched with text analytics results. Lastly, the enriched message is sent to the second Event Hub you deployed, where it will be further processed in lab 4.

Azure Functions can run either in “Consumption” plans or in App Service Plans. You will deploy an App Service Plan, which provides the ability to keep a Function running (“Always On”), which is consistent with our scenario of taxi devices sending in messages around the clock. After the App Service Plan is deployed, you will then deploy the Azure Function that uses it.

In your Resource Group, click “+ Add” and type “App Service Plan” into the search box, then Enter. Select “App Service Plan”, then click “Create”.

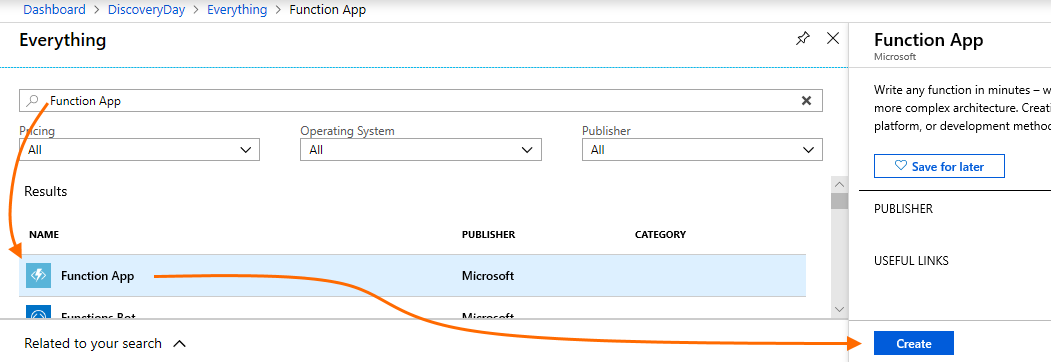


On the create blade, provide a name for the App Service Plan. Then, ensure your Resource Group is selected; leave Operating System at Windows; set the Location to the Azure region you have been using so far; then click Pricing Tier. On that view, select P1V2 and click “Apply”. Then, click “Create”.



Return to your Resource Group. When the App Service Plan has completed deployment (reminder – use the bell glyph at the top of the portal view to monitor deployment status and other events) and you can see the new App Service Plan in your Resource Group (you may need to click “Refresh”), click “+ Add” again.

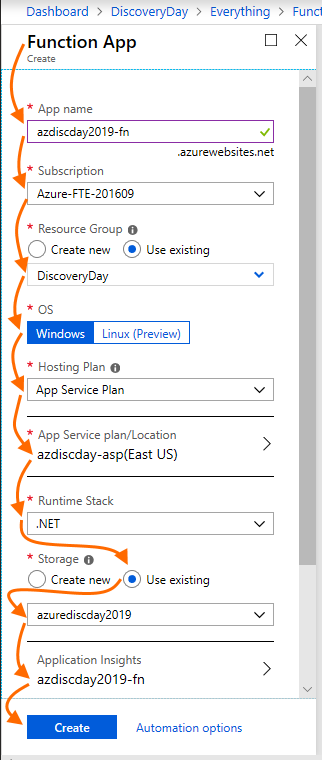
Type “Function App” into the search box. Click “Function App” in the search results, then click “Create” on its info blade.



On the Function App’s create blade, ensure that all the following are correctly entered!

* App Name: enter a name for your Azure Function App
* Resource Group: ensure the Resource Group you have been using so far is selected
* OS: ensure Windows is selected
* Hosting Plan: select “App Service Plan”, then select the App Service Plan you just deployed
* Runtime Stack: ensure .NET is selected
* Storage: select “Use existing” and select the storage account you deployed in lab 1
* Application Insights: this is an optional Application Performance Monitoring solution that is helpful in issue analysis and debugging. You can leave this set to deploy, or you can click and disable deployment if desired (it does not add materially to deployment time).

When all information is correctly entered, click “Create”. Return to your Resource Group while deployment proceeds.



When the Function App has completed deployment, find it in your Resource Group. Remember to monitor notifications and to click “Refresh” in the Resource Group view. Click on the Function App you just deployed and work through the following steps.

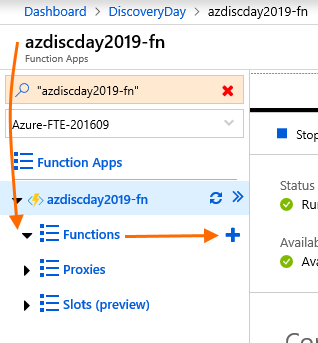
There are several ways to develop and deploy Azure Functions, including from Visual Studio and Visual Studio Code as well as the Azure portal. In this lab, we will use the Azure portal and deploy a C# script Function, but you are encouraged to learn about the other ways to develop and deploy Functions.

This task has several steps:

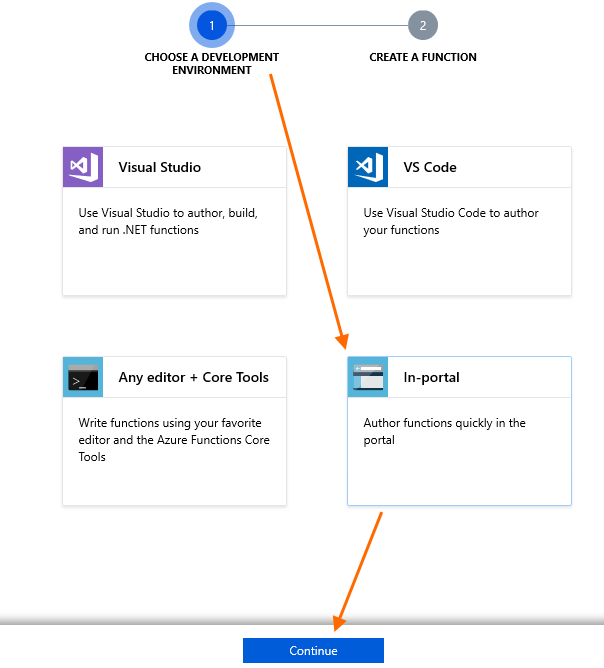
1. You will create a new Function and add an appropriate trigger binding that will cause the Function to run
2. You will add an output binding, which will let your Function pass the results of its work to the next step in the pipeline you are building in this lab
3. You will import packages from Nuget, to provide functionality you will need in the Function code
4. You will add Application Settings to store values from other resources you deployed earlier in this lab so that your Function code can access these values
5. You will conclude by adding Function code to process, enrich, and pass on the inbound messages. You will also test your Function before the taxi device simulator is live.

#### Create a Function and add a Trigger binding

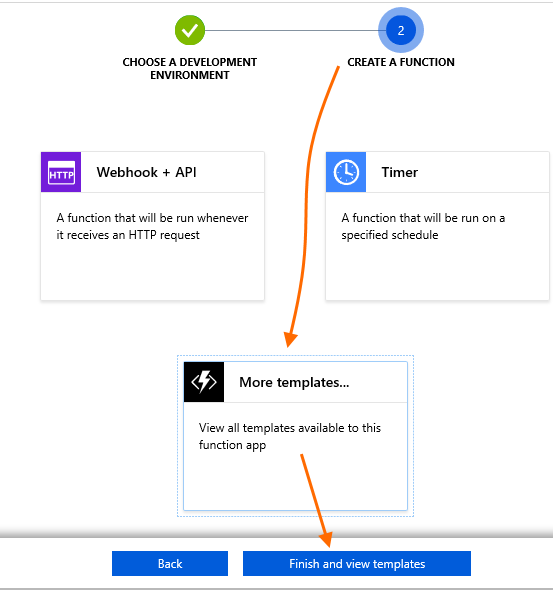
First, create a new Function. To do this, in the Function App left navigation bar, locate “Functions” and click “+” next to it.



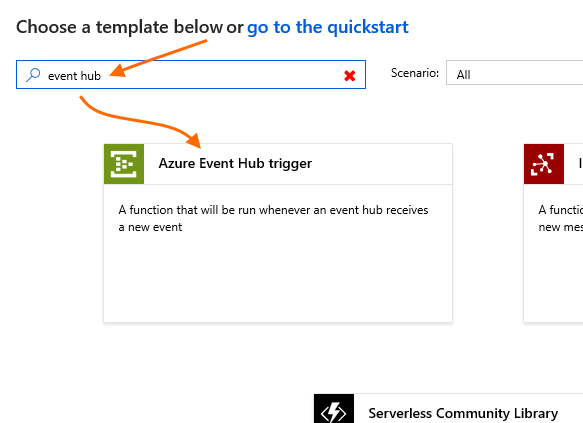
Next, you will be asked to pick your Development Environment. Click “In-portal”, then click “Continue”.



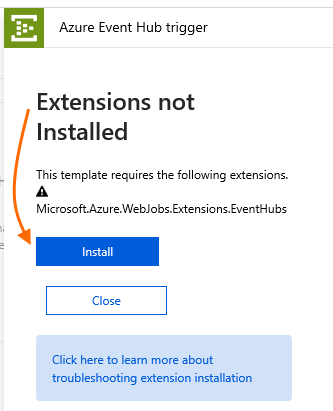
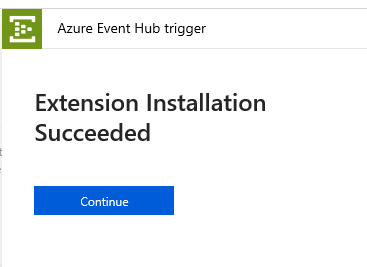
Next, click “More templates…” and “Finish and view templates”.



On the Function template view, find the “Azure Event Hub trigger” template. You can scroll to find it, or you can type “event hub” in the search box to narrow the selections. Click it to continue.



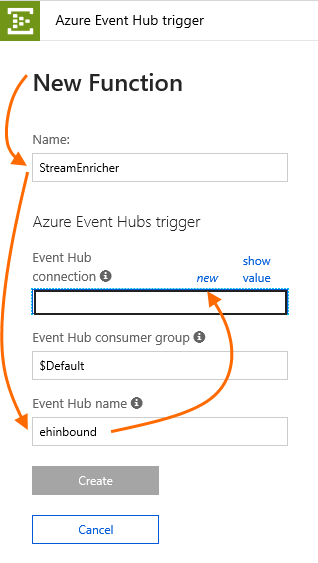
You may now be prompted to install missing extensions. If so, click “Install” and wait for installation to complete, then click “Continue”.

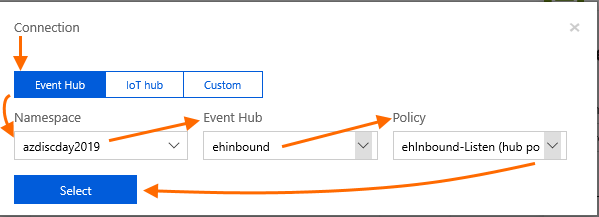
Next, on the “New Function” blade, provide a Function name. Fill in the first Event Hub name you created in task 1.

Leave “Event Hub consumer group” at its default value (“$Default”). This is a more advanced option for when more than one group of resources “listens” to an Event Hub; it is not needed for this lab.

Then, configure a connection to the first Event Hub: that is where inbound messages from taxi devices will come, and you will now configure this Function to listen for those messages on that Event Hub. Next to “Event Hub connection”, click “new”.

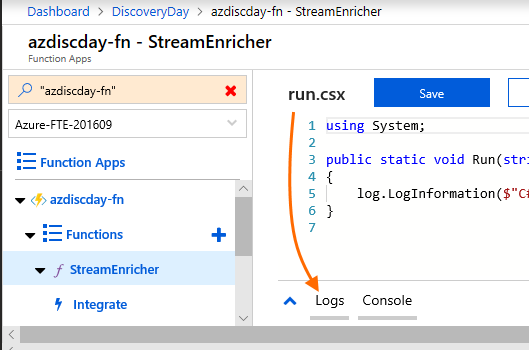


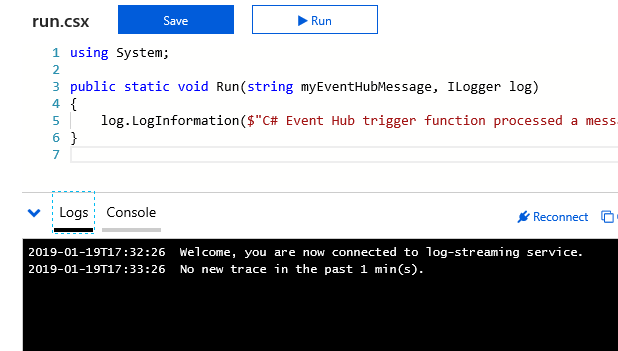
On the Event Hub connection view, select the Event Hub namespace, the first Event Hub and the Listen policy you created on the Event Hub. You created all of these in task 1. Then click “Select”.



You will now be returned to the “New Function” view, which now shows the Event Hub connection that you just created. Click “Create”, which is now enabled, to create the Function.

You will now see the Function’s code window. Note the Logs tab at the bottom; when in this code view, you should always click this to open it, as it will be very useful for debugging and monitoring your Function’s activity.

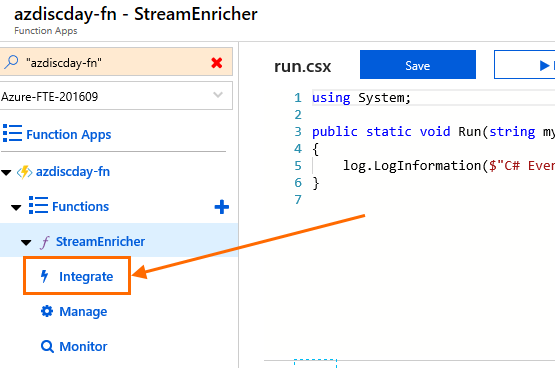




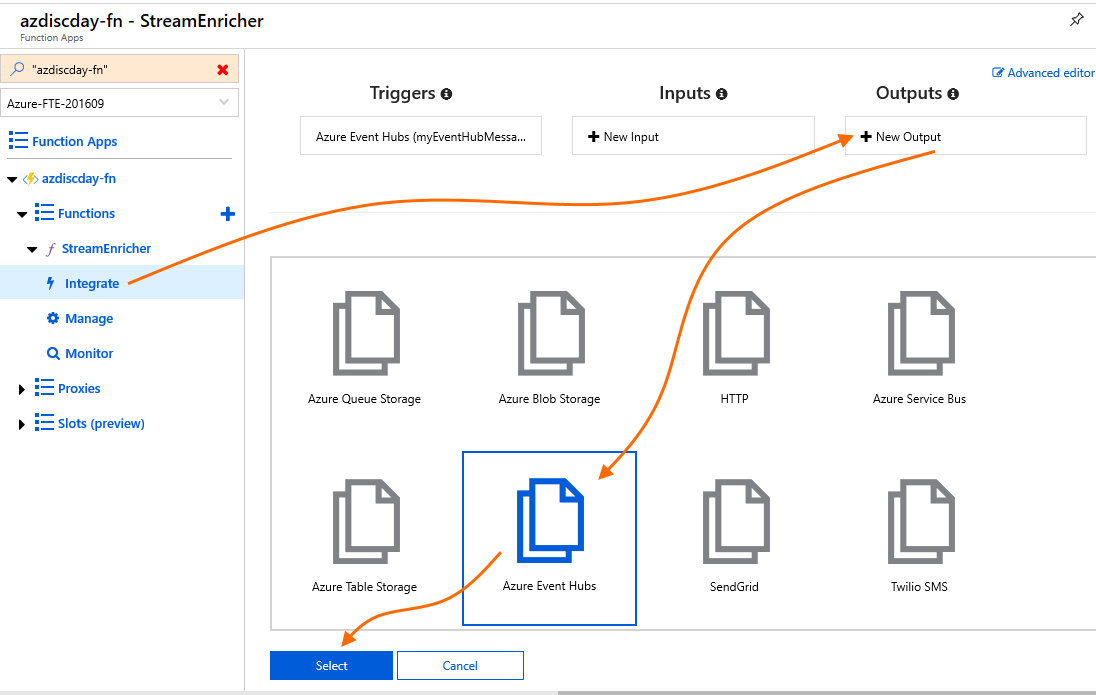
#### Add an Output Binding

Now, you need to add an output binding. This is where your Function will write enriched stream messages after processing them.

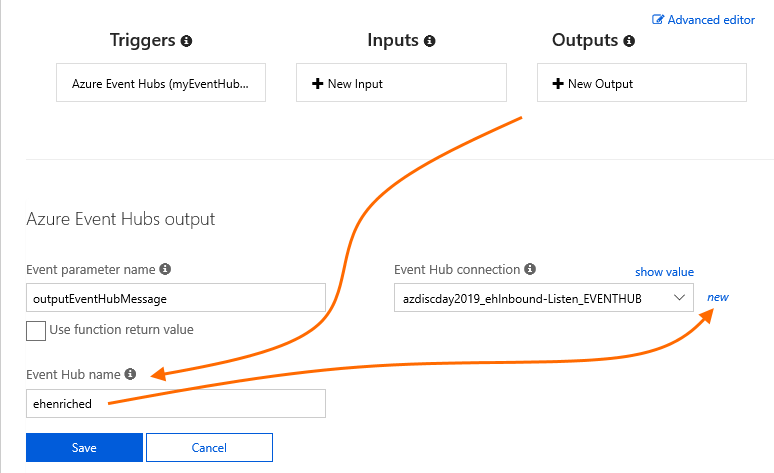
In the left menu, find your Function and, below it, click “Integrate”.



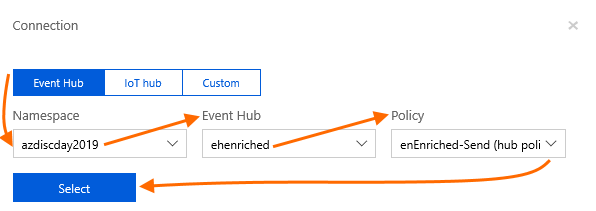
On the Integrate view, under “Outputs” click “+ New Output”. Then click “Azure Event Hubs” from the choices. Then click “Select”.



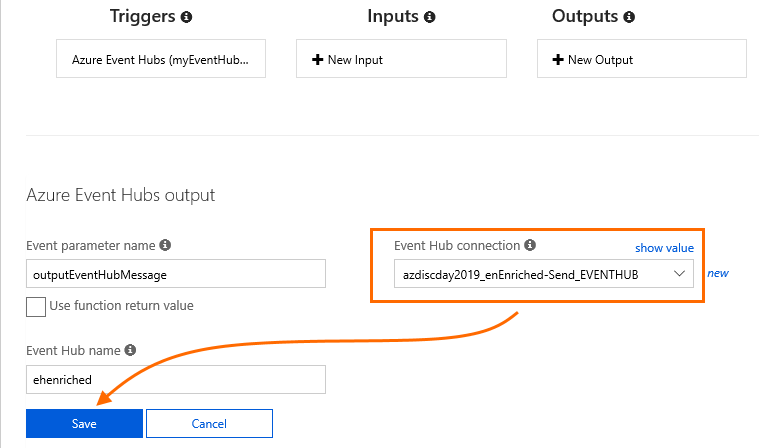
On the output view, provide the second Event Hub name you created in task 1. This is the Event Hub to which the Function sends the enriched message after finishing processing. Then, under “Event Hub connection”, click “new” (ignore the pre-filled connection – that is for the inbound Event Hub which we used previously for the Function trigger).



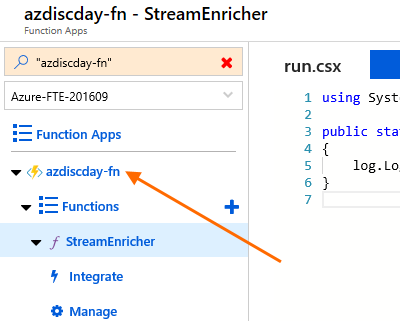
For this Event Hub Connection, provide the same Event Hubs Namespace as previously. This time, specify the second Event Hub you created in task 1, and the Send policy you created on that Event Hub. Then click “Select”.



You are now back at the output creation view, which now shows the Event Hub connection you just created. After verifying that it looks correct, click “Save”.



Next, in the left navigation bar, click on the Function App node (lightning-bolt icon with the name you used to create the Function App in the Azure portal). This will bring you back to the Overview. You can now move on to the next step.



To complete this task, you must make your Function code aware of this new output binding. Return to the code view and expand the “Logs” tab.

Add a parameter to the Run() method signature. The name of the parameter must be the “Event parameter name” you specified on the new output binding view – in this case, the default “outputEventHubMessage” is used (see previous screenshots).

Add the parameter as follows:

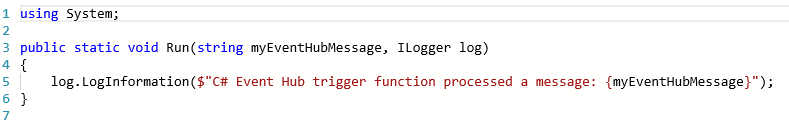
out string outputEventHubMessage.

Next, in the body of the function, add an initial assignment for the output parameter’s value (this is required for output parameters in C# functions). You will add more useful code later in this task.

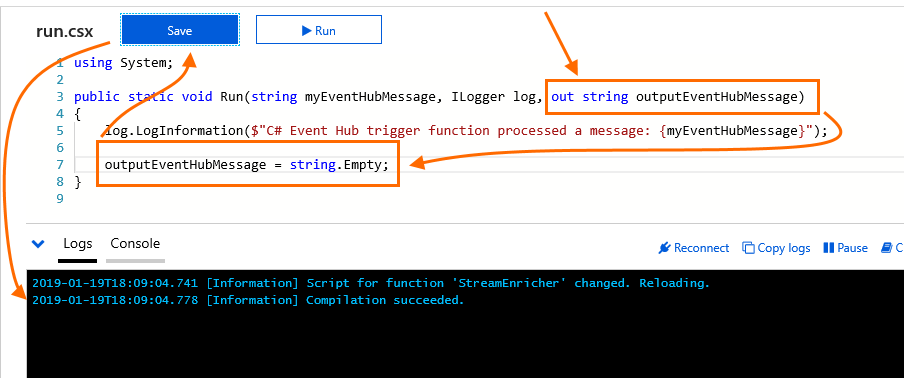
outputEventHubMessage = string.Empty;

Then click “Save” at the top of the code window. If the code is correct, you will see a “Compilation succeeded” message in the Logs tab.

Before:



After adding the out parameter and the new line of code and clicking “Save”:



You have now successfully added an output binding, and made your Function code aware of it. You can now proceed to the next step.

#### Import Packages from Nuget

The Function’s code will need to use two external libraries for its functionality. Your Function will need to download these libraries from nuget.org.

Your Function will use these Packages:

1. Newtonsoft.Json. This is a widely-used JSON processing library.
2. pelazem.azure.cognitive.textanalytics. This is a library that wraps several calls to the Azure Text Analytics Cognitive Service into one result set.  
   The Text Analytics Cognitive Service requires a separate API call for each type of text analytics; this library abstracts away this and other complexities and returns one combined text analytics result, with which your code will enrich the inbound message.  
   Note that this library is a convenience. You could also interact directly with the Text Analytics Cognitive Service via HTTP REST calls; like all the Azure Cognitive Services, the service we will use exposes a REST API.

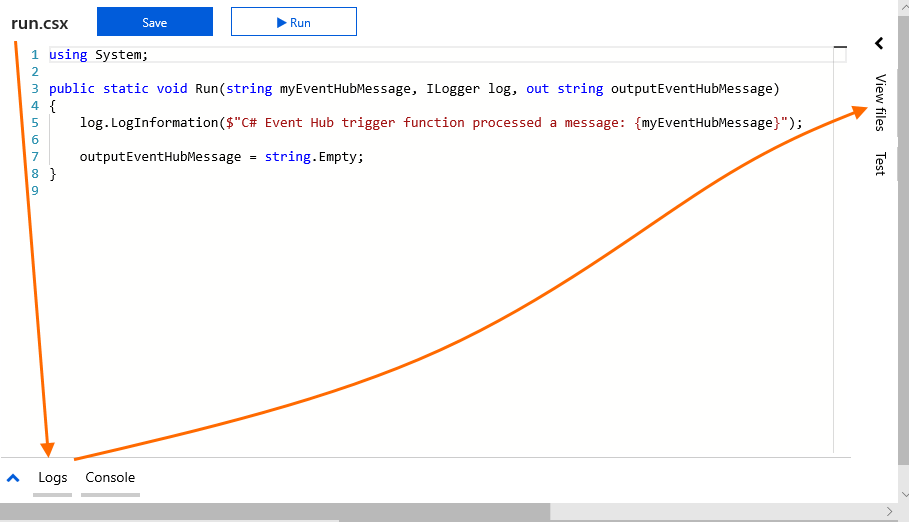
In this lab, you are authoring a Function in the portal. Portal-authored Functions use a specific file to initiate nuget package download and install. This file is called function.proj. This file is provided with this lab document.

Please download a copy to your machine from this URL:

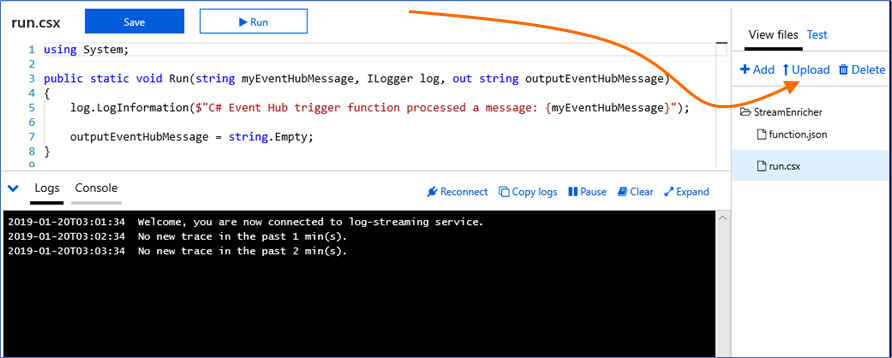
<https://raw.githubusercontent.com/plzm/azure-discoveryday2019-mdw/master/labs/lab3/StreamEnricherFunction/function.proj>

Note: you can also go to the github repository for this workshop at <https://github.com/plzm/azure-discoveryday2019-mdw> and navigate all labs and documents there.

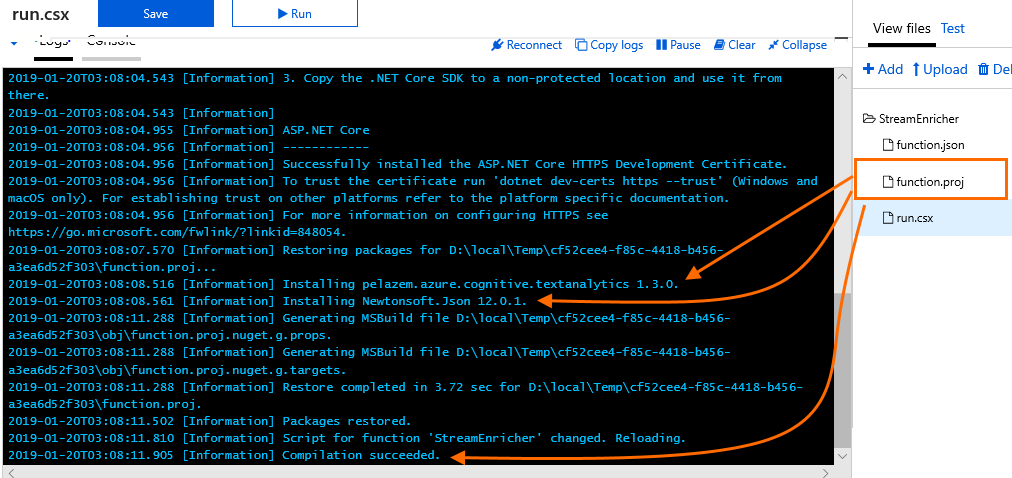
In your Function’s code view, first expand the “Logs” tab at the bottom. Then expand the “View files” tab at the right.



You will now see both the Logs console at the bottom, and the “View files” list of Function files at the right. Click “Upload” above the list of files.



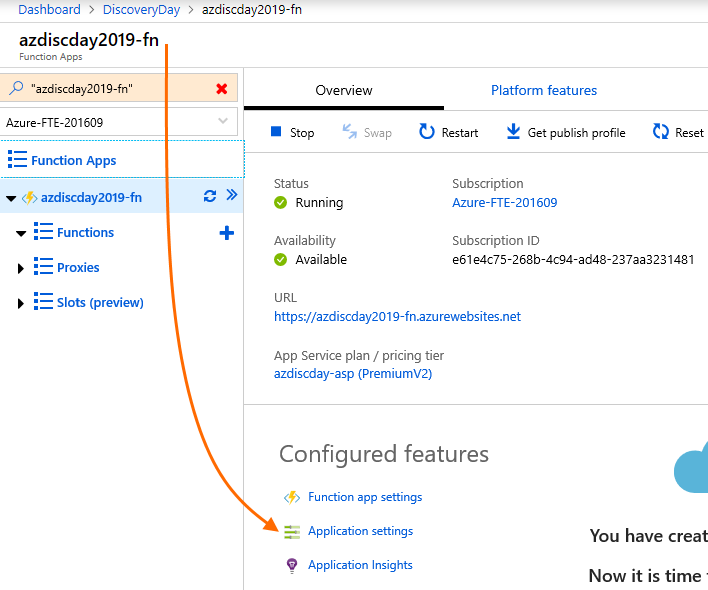
Find the function.proj file you just downloaded, and select it in the upload dialog. After it uploads, you will see several messages in the Console tab, concluding with installation of the two packages and successful compilation of your Function.



When your Function has installed these two packages and compiled successfully, this step is complete. Please move on to the next step.

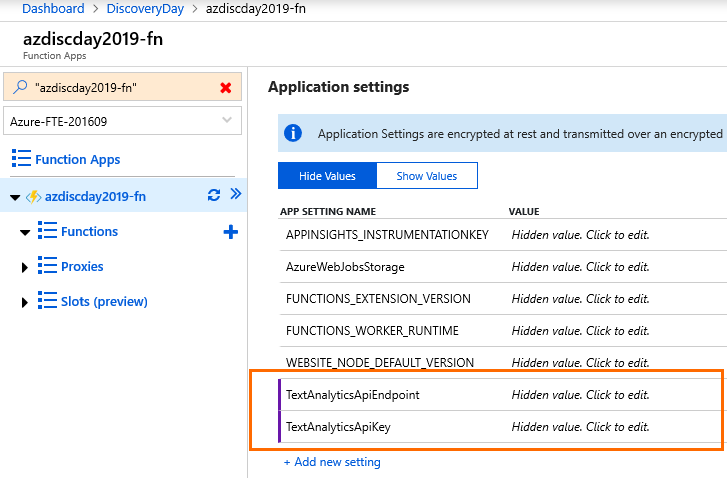
#### Add Application Settings

In the left navigation, click the node with the lightning bolt and your Function App’s name to go to the Overview. In the Function App Overview, click “Application Settings”.

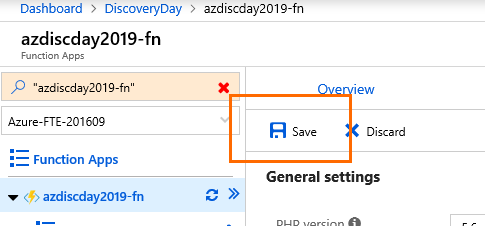


Scroll down to the “Application settings” section. Add two new application settings. For each, click “+ Add new setting”, then add the setting as follows. Note that as soon as you enter the value, the portal view will hide the value and show “Hidden value. Click to edit.” This is to protect sensitive data.

|  |  |
| --- | --- |
| Name | Value |
| TextAnalyticsApiEndpoint | Enter the API Endpoint value you retrieved after deploying the Text Analytics Cognitive Service in task 2. |
| TextAnalyticsApiKey | Enter the API Key value you retrieved after deploying the Text Analytics Cognitive Service in task 2. |



Next, scroll back up and click “Save”.



After the Application Settings are saved, move to the next step.

#### Add and Test Function Code

The last step in this task is to add our Function’s code, then to test it with a sample message.

Your Function code needs to accomplish the following tasks:

1. Receive a new taxi device message from the inbound Event Hub
2. Deserialize the message to structured JSON
3. Extract the customer’s comments text from the taxi message JSON
4. Instantiate the text analytics library, pass in the customer’s comments text for analysis, and receive text analytics results back
5. Add selected parts of the text analytics results to the received (inbound) taxi device message – this is where message enrichment occurs
6. Write the enriched message to the output, which points to the second Event Hub (which you will work with further in lab 4)

A complete, working version of the Function code is provided for you in this workshop’s github repo. The full URL to the code file is:

<https://raw.githubusercontent.com/plzm/azure-discoveryday2019-mdw/master/labs/lab3/StreamEnricherFunction/run.csx>

Please download a copy of this file, then upload it into your Azure Function using the same process as you did with function.proj in a previous step. You should overwrite the existing run.csx file.

After the file is uploaded, confirm in the “Logs” tab (you did expand it, didn’t you?) that the script for your Function changed and that compilation succeeded.

Let’s examine a few key pieces of this code. You are encouraged to experiment and change the code, clicking “Save” after changes – look for compilation messages in the “Logs” tab when you do this.

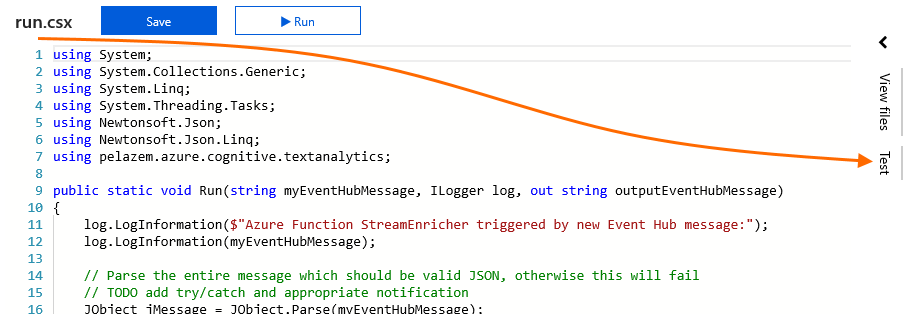
Note that the code contains many console output statements (lines beginning with log.LogInformation). Many of these are commented out with a leading //. These lines – when not commented out – will write to the Logs tab, providing very helpful real-time information about what your Function is doing.

Consider uncommenting (or commenting) some of these lines as you examine, test, and change the code.

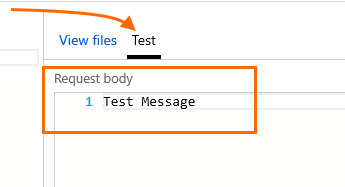
Can you match what the code is doing to the list of six tasks above?

Note the lines that use Environment.GetEnvironmentVariable(“…”). These lines go to the Application Settings and retrieve the values you pasted there in an earlier step. It’s a good idea to uncomment the immediately following log.LogInformation() calls that echo those values to the “Logs” tab, to ensure you pasted the right keys and values into Application Settings.

Now, let’s test the Function code. Still in your Function code window, expand the “Test” tab on the right side.



You will see a “Request body” text box, pre-populated with sample text “Test Message”.

Select and delete that test message.

A sample message is provided for you in the github repo for this workshop. Please navigate to this URL and copy/paste the sample taxi device message into the Test “Request body” text box where you just deleted “Test Message”.