Fall 2023- CSCE 438/838: IoT-Lab 6 - Automating Things

In this lab, we are going to develop tools to build a Machine Learning (ML) based automation system. You will use these to develop an end-to-end system. The instructions given below assume that you have a basic IoT Central application and a Python-based device implemented.

Machine Learning

Machine learning is a technique of data science that helps computers learn from existing data to forecast future behaviors, outcomes, and trends. In this lab, we are going to locally train an ML model and deploy it to the cloud.

Getting the data

We are going to use a machine learning model to make some predictions. So unlike in the previous lab, we can't use randomly generated data. We are going to use the Rain in Austrailia dataset. This dataset contains about 10 years of daily weather observations from many locations across Australia. It also has `RainTomorrow` column, which tells whether or not it rained the next day. Use the link to download the dataset from Kaggle.

Building a predictive model

We are going to train the model and save it as a pickle file in Python. It's a protocol to serialize Python objects and can be used to save the scikit-learn models.

1. Install Python libraries.

It is highly recommended that you use a virtual environment to manage dependencies.

```
pip install pandas
pip install numpy
pip install scikit-learn
```

Depending on your environment `pip3` may work instead of `pip` above.

2. Extract features and train the model

The Python code given below processes the data through a series of operations and trains a logistic regression model using the processed data. It needs to execute from the directory where the dataset has been placed.

```
# Load all the libraries
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report
from sklearn.linear_model import LogisticRegression
```

```
# Load the data
dataset = 'weatherAUS.csv'
rain = pd.read_csv(dataset)

# Reduce the cardinality of date by splitting it into year month and day
rain['Date'] = pd.to_datetime(rain['Date'])
rain['year'] = rain['Date'].dt.year
rain['month'] = rain['Date'].dt.month
rain['day'] = rain['Date'].dt.day
rain.drop('Date', axis = 1, inplace = True)

# Classify feature type
categorical_features = [
    column_name
    for column_name in rain.columns
    if rain[column_name].dtype == 'O'
]

numerical_features = [
    column_name
    for column_name in rain.columns
    if rain[column_name].dtype != 'O'
]

# Fill missing categorical values with the highest frequency value in the column
categorical_features_with_null = [
    feature
    for feature in categorical_features
    if rain[feature].isnull().sum()
]

for each_feature in categorical_features_with_null:
    mode_val = rain[each_feature].mode()[0]
    rain[each_feature].fillna(mode_val,inplace=True)

# Before treating the missing values in numerical values, treat the outliers
features_with_outliers = [
    'MinTemp',
    'MaxTemp',
    'Rainfall',
    'Evaporation',
    'WindGustSpeed',
    'WindSpeed9am',
    'WindSpeed3pm',
    'Humidity9am',
    'Pressure9am',
    'Pressure3pm',
    'Temp9am',
    'Temp3pm'
]
```

```python
for feature in features_with_outliers:
    q1 = rain[feature].quantile(0.25)
    q3 = rain[feature].quantile(0.75)
    IQR = q3 - q1
    lower_limit = q1 - (IQR * 1.5)
    upper_limit = q3 + (IQR * 1.5)
    rain.loc[rain[feature]<lower_limit,feature] = lower_limit
    rain.loc[rain[feature]>upper_limit,feature] = upper_limit

# Treat missing values in numerical features
numerical_features_with_null = [
    feature
    for feature in numerical_features
    if rain[feature].isnull().sum()
]

for feature in numerical_features_with_null:
    mean_value = rain[feature].mean()
    rain[feature].fillna(mean_value,inplace=True)



# Encoding categorical values as integers
direction_encoding = {
    'W': 0, 'WNW': 1, 'WSW': 2, 'NE': 3, 'NNW': 4,
    'N': 5, 'NNE': 6, 'SW': 7, 'ENE': 8, 'SSE': 9,
    'S': 10, 'NW': 11, 'SE': 12, 'ESE': 13, 'E': 14, 'SSW': 15
}

location_encoding = {
    'Albury': 0,
    'BadgerysCreek': 1,
    'Cobar': 2,
    'CoffsHarbour': 3,
    'Moree': 4,
    'Newcastle': 5,
    'NorahHead': 6,
    'NorfolkIsland': 7,
    'Penrith': 8,
    'Richmond': 9,
    'Sydney': 10,
    'SydneyAirport': 11,
    'WaggaWagga': 12,
    'Williamtown': 13,
    'Wollongong': 14,
    'Canberra': 15,
    'Tuggeranong': 16,
    'MountGinini': 17,
    'Ballarat': 18,
    'Bendigo': 19,
    'Sale': 20,
    'MelbourneAirport': 21,
    'Melbourne': 22,
    'Mildura': 23,
    'Nhil': 24,
    'Portland': 25,
    'Watsonia': 26,
    'Dartmoor': 27,
    'Brisbane': 28,
    'Cairns': 29,
    'GoldCoast': 30,
    'Townsville': 31,
    'Adelaide': 32,
    'MountGambier': 33,
    'Nuriootpa': 34,
    'Woomera': 35,
    'Albany': 36,
    'Witchcliffe': 37,
    'PearceRAAF': 38,
    'PerthAirport': 39,
    'Perth': 40,
    'SalmonGums': 41,
    'Walpole': 42,
    'Hobart': 43,
    'Launceston': 44,
    'AliceSprings': 45,
    'Darwin': 46,
    'Katherine': 47,
    'Uluru': 48
}
boolean_encoding = {'No': 0, 'Yes': 1}


rain['RainToday'].replace(boolean_encoding, inplace = True)
rain['RainTomorrow'].replace(boolean_encoding, inplace = True)
rain['WindGustDir'].replace(direction_encoding,inplace = True)
rain['WindDir9am'].replace(direction_encoding,inplace = True)
rain['WindDir3pm'].replace(direction_encoding,inplace = True)
rain['Location'].replace(location_encoding, inplace = True)

# See the distribution of the dataset
print(rain['RainTomorrow'].value_counts())

# Split features and target value as X and Y
X = rain.drop(['RainTomorrow'],axis=1)
y = rain['RainTomorrow']

# Split training and test split
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.2, random_state
= 0)

# Scale input using just the training set to prevent bias
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
# Train the model
classifier_logreg = LogisticRegression(solver='liblinear', random_state=0)
classifier_logreg.fit(X_train, y_train)

# Test the model accuracy
y_pred = classifier_logreg.predict(X_test)

print(f"Accuracy Score: {accuracy_score(y_test,y_pred)}")
print("Classifcation report", classification_report(y_test,y_pred))
```

3. Save the model

Once the model has been trained, it can be saved using pickle library. Then it can be loaded again when it needs to be used.

```
import pickle
with open("iot_model", "wb") as model_file:
    pickle.dump(classifier_logreg, model_file)
```

Use the following snippet of code to load the model again.

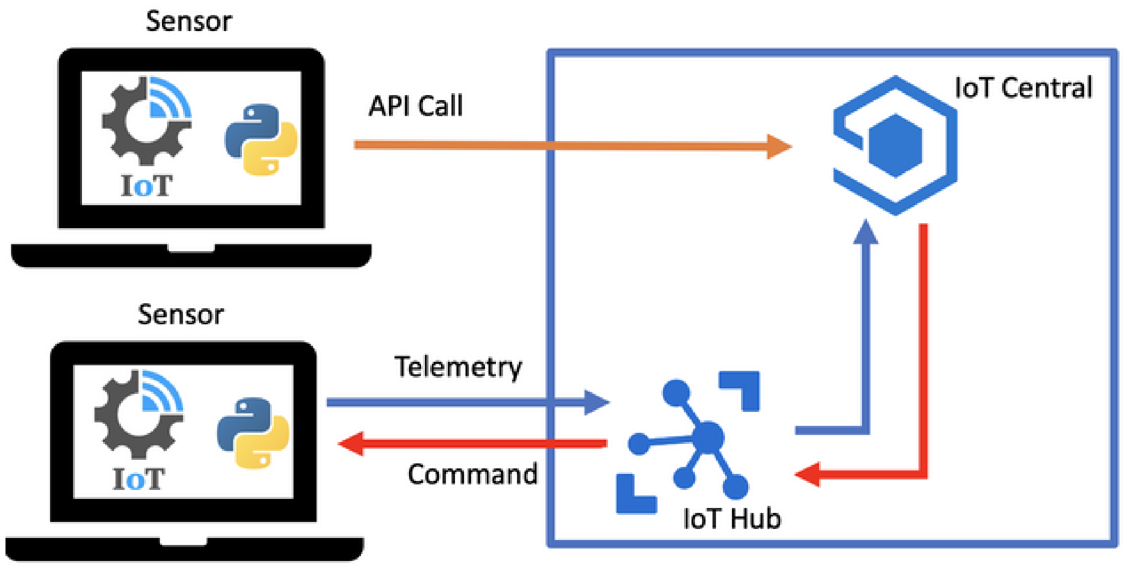
```
with open("iot_model", "rb") as model_file:
    model = pickle.load(model_file)
y_pred_new = classifier_logreg.predict(X_test)
```

```
print("Output of loaded model is same as original model ?", all(y_pred == y_pred_new))
```

Programmatically sending commands to IoT Central devices

In Lab 5, we sent IoT Central commands using the web application. IoT Central also allows using RESTful API to send commands. We are going to use this approach as it makes automating things much easier. In addition, we will use Azure portal's function as a service (FaaS) functionality. Therefore, the following sections combine functionalities of Azure IoT Center (https://apps.azureiotcentral.com), Azure Portal (https://portal.azure.com/), and local virtual IoT devices running python.

First, we will sent API commands through IoT Central. Here is the big picture:

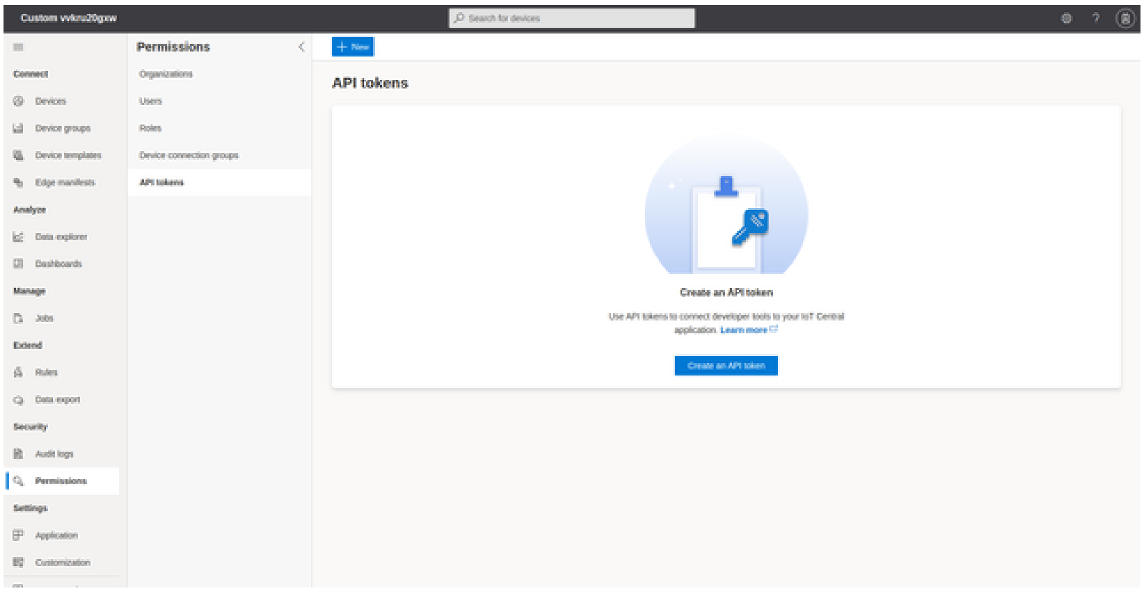


If you don't have it already, install the requests library.

```
pip install requests
```

To send requests to IoT Central, you will need to use the API key. Go to: https://apps.azureiotcentral.com/

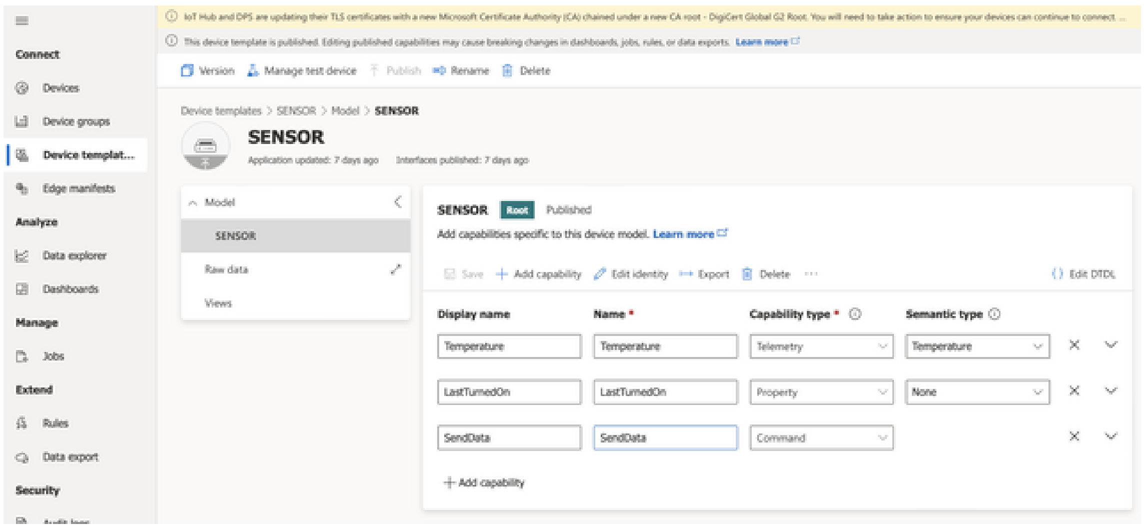
Click **My apps** on the left and click the IoT Central app you built in Lab 5 (or generate one based on Lab 5 instructions). You can get the API key from **Permissions -> API tokens**. Click **Create an API token**



Create a key using a Token name (e.g., 'IoTAzureML') and the **App Operator** role and click **Generate**. Copy the **Token** right away as you won't have access to it again. This API key allows you to access your app in the IoT Central. Any devices within this app can be accessed.

Then, we will use the virtual IoT device you created in Lab 5 with the `SENSOR` template. If necessary, create a new device using the instructions in Lab 5. Save the **Device ID**.

Make sure you use the **Device template** `SENSOR` from Lab 5 for your device, including `SendData` **Command** capability.



IoT Hub and DPS are updating their TLS certificates with a new Microsoft Certificate Authority (CA) chained under a new CA root - DigiCert Global G2 Root. You will need to take action to ensure your devices can continue to connect. ...

This device template is published. Editing published capabilities may cause breaking changes in dashboards, jobs, rules, or data exports. Learn more

SENSOR

Application updated: 7 days ago Interfaces published: 7 days ago

SENSOR Root Published

Add capabilities specific to this device model. Learn more

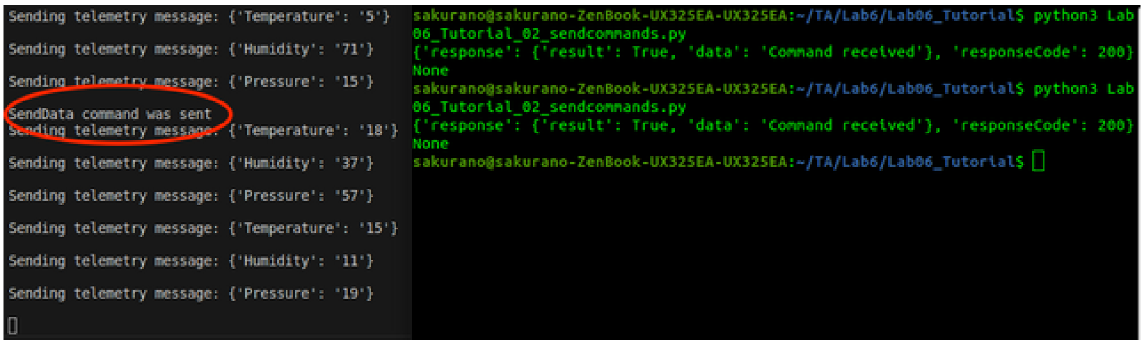
Display name	Name *	Capability type *	Semantic type
Temperature	Temperature	Telemetry	Temperature
LastTurnedOn	LastTurnedOn	Property	None
SendData	SendData	Command	

The following snippet assumes you have IoT Central and a device. You should already have saved the `device_id` and `api_key`. For the `iotc_sub_domain`, take a look at the address field of your browser and copy the portion before `.azureiotcentral.com ...` in the address field. This is the name of your IoT Central App.

```
import requests

iotc_sub_domain = "your_domain"
device_id = "your_device_id"
api_key = "your_api_key"

def _command_url():
    return
f"https://{iotc_sub_domain}.azureiotcentral.com/api/devices/{device_id}/commands/SendDat
api-version=2022-05-31"

def send_command():
    resp = requests.post(
        _command_url(),
        json={},
        headers={"Authorization": api_key}
    )
    print(resp.json())

print(send_command())
```

Please note that the capability Name MUST BE same as that is specified in `_def command_url():` Because this API Call will look for the Command name you specified. As a default, API call is looking for the Name `SendData`. If you name the capability name as `SendData` in your **SENSOR** template, then you do not need to modify the `def _command_url():`

After making changes, we will now send commands from one virtual IoT device to another one. In one terminal window, run your Lab 5 `Lab05_IoTCSample.py` code. This will start sending random temperature samples to IoT Central (and will listen for any commands). Open another terminal window and run the above (modified) code `Lab06_Tutorial_02_sendcommands.py`. It will send commands to the former virtual IoT node.



```
Sending telemetry message: {'Temperature': '5'}

Sending telemetry message: {'Humidity': '71'}

Sending telemetry message: {'Pressure': '15'}

SendData command was sent
Sending telemetry message: {'Temperature': '18'}

Sending telemetry message: {'Humidity': '37'}

Sending telemetry message: {'Pressure': '57'}

Sending telemetry message: {'Temperature': '15'}

Sending telemetry message: {'Humidity': '11'}

Sending telemetry message: {'Pressure': '19'}
```

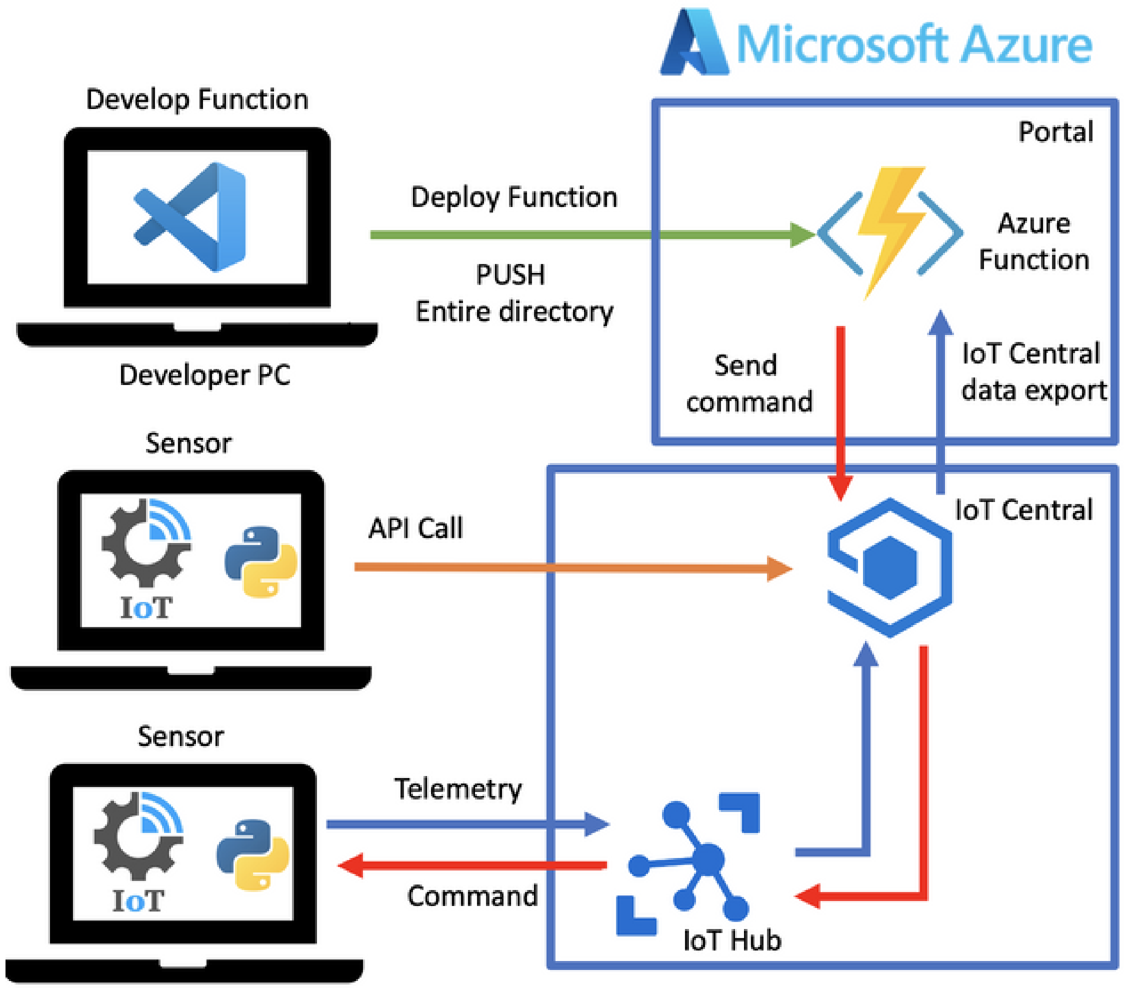
```
sakurano@sakurano-ZenBook-UX325EA-UX325EA:~/TA/Lab6/Lab06_Tutorial$ python3 Lab
06_Tutorial_02_sendcommands.py
{'response': {'result': True, 'data': 'Command received'}, 'responseCode': 200}
None
sakurano@sakurano-ZenBook-UX325EA-UX325EA:~/TA/Lab6/Lab06_Tutorial$ python3 Lab
06_Tutorial_02_sendcommands.py
{'response': {'result': True, 'data': 'Command received'}, 'responseCode': 200}
None
sakurano@sakurano-ZenBook-UX325EA-UX325EA:~/TA/Lab6/Lab06_Tutorial$
```

Now, you can send commands from one (virtual) IoT device to another through an API call to IoT Central! Next, we will deploy a function in the cloud instead of running it locally.

Deploying Azure functions

Azure Function is a function-as-a-service (FaaS) offering by Azure. It provides the runtime needed to run your application code without having to maintain the infrastructure. We will deploy a simple Python-based Azure function. We are going to use the Azure Function extension in Visual Studio Code to deploy the Azure function.

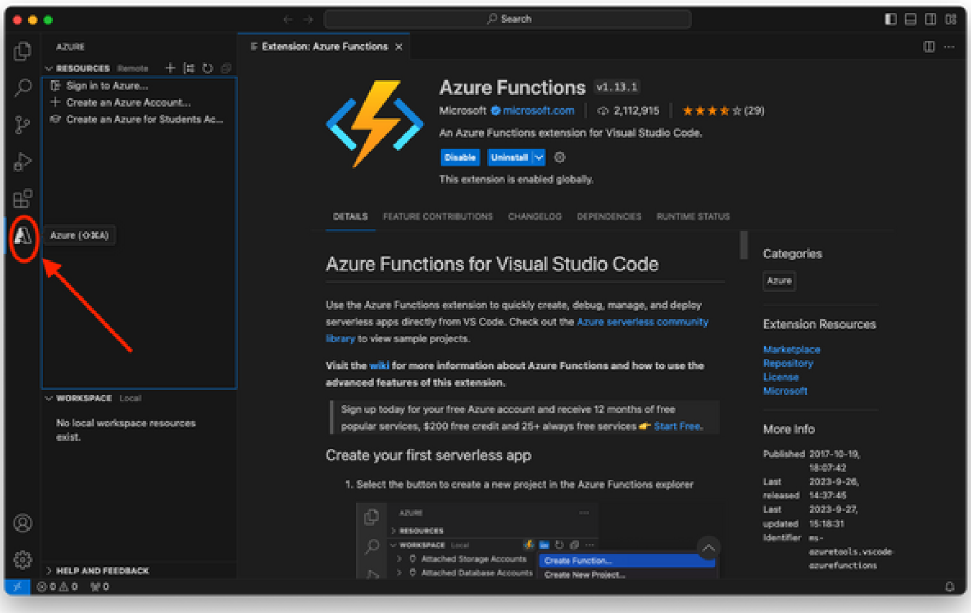
Here is the big picture.



1. If you haven't already, follow the instructions given in https://code.visualstudio.com/docs/setup/setup-overview to install VS Code.
2. Open VS Code and install **Azure Functions for Visual Studio Code**. To do this, press Ctrl + P (or Command+P in MacOS) to open VS Code quick open and paste the following command

```
ext install ms-azuretools.vscode-azurefunctions
```

3. After installation is complete, you will see the Azure icon on the Activity bar on the left.



No local workspace resources exist.

An Azure Functions extension for Visual Studio Code.

This extension is enabled globally.

Azure Functions for Visual Studio Code

Use the Azure Functions extension to quickly create, debug, manage, and deploy serverless apps directly from VS Code. Check out the Azure serverless community library to view sample projects.

Visit the wiki for more information about Azure Functions and how to use the advanced features of this extension.

> Sign up today for your free Azure account and receive 12 months of free popular services, $200 free credit and 25+ always free services 👉 Start Free.

Create your first serverless app

1. Select the button to create a new project in the Azure Functions explorer

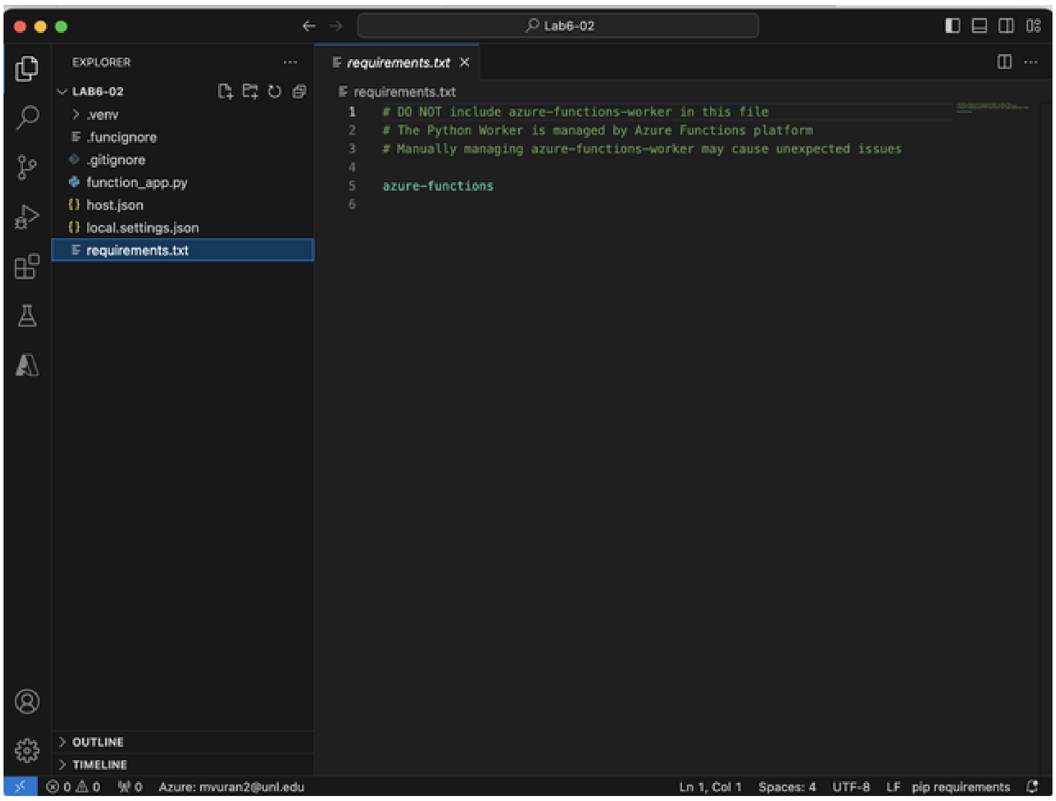
Published	2017-10-19, 18:07:42
Last released	2023-9-26, 14:37:45
Last updated	2023-9-27, 15:18:31
Identifier	ms-azuretools.vscode-azurefunctions

Click the Azure icon and familiarize yourself with the interface on the left. You will see two portions: **RESOURCES** on top, and **WORKSPACE** at the bottom. We will go back and forth between these fields. Note that additional buttons emerge when you hover your mouse icon on **RESOURCES** and **WORKSPACE**. Press **Sign in to Azure** in the RESOURCES tab and complete the authentication process.

4. Now, hover over WORKSPACE and click the Azure Functions icon (lightning-looking icon) and then click **Create a new project**. Then select the following options

- Language: Python
- Interpreter: Latest version installed in your system
- Template for the project: HTTP Trigger
- HTTP Trigger name: HTTPTrigger
- Authorization Level: Function
- How to open: Current Window

5. You should have a folder structure like the one given below (you may want to click Explorer, the top icon on the left tab).

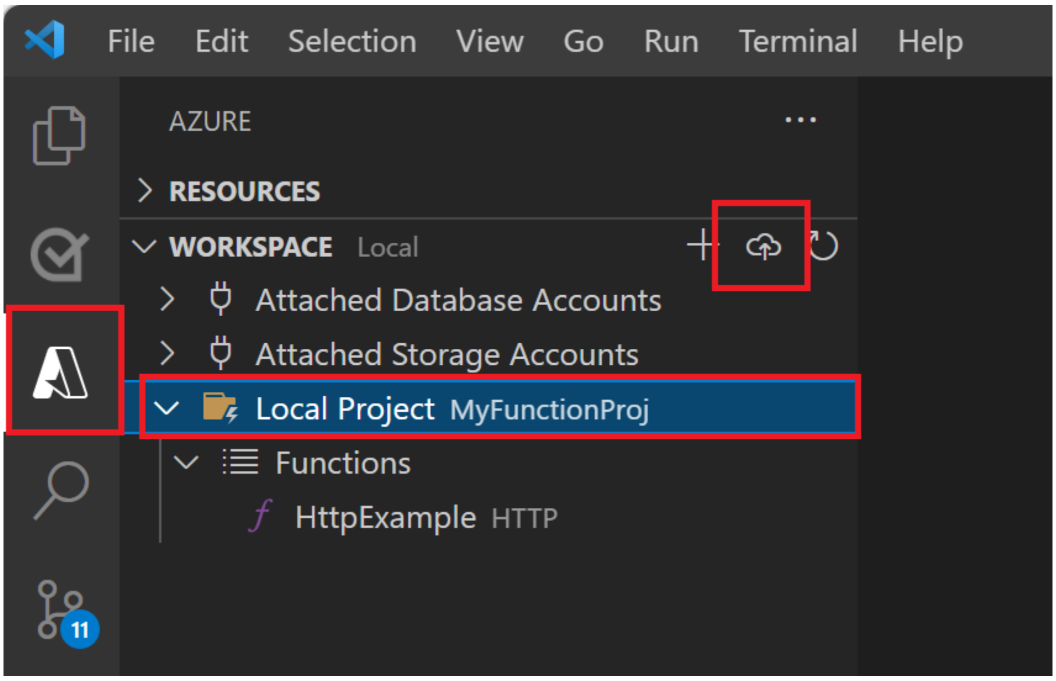


```
# DO NOT include azure-functions-worker in this file
# The Python Worker is managed by Azure Functions platform
# Manually managing azure-functions-worker may cause unexpected issues

azure-functions
```

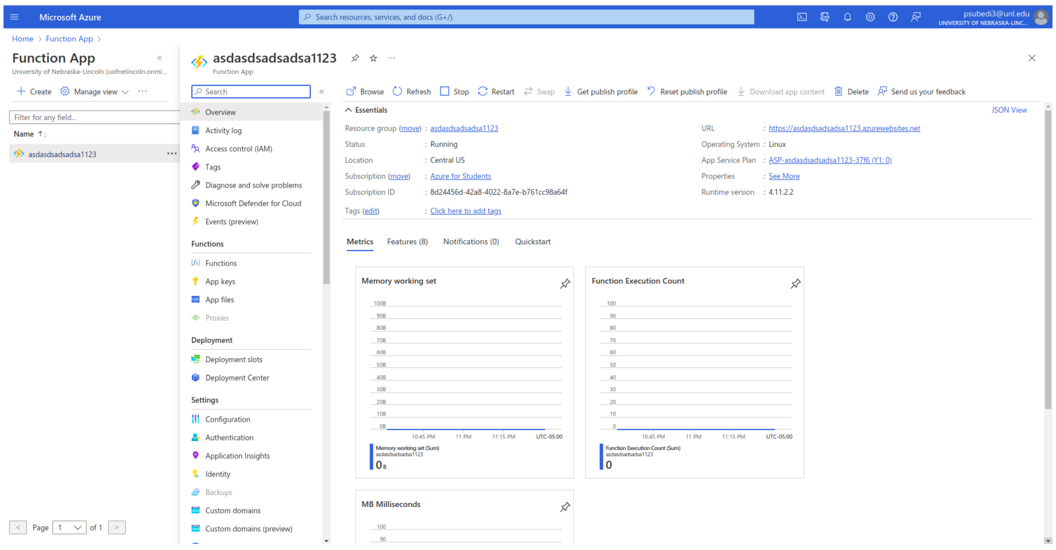
6. We will focus on the automatically generated `function_app.py`. Add a line to log the req_body `logging.info(req_body)` in the `function_app.py` file. The code should be like this

```
import azure.functions as func
import logging

app = func.FunctionApp(http_auth_level=func.AuthLevel.FUNCTION)

@app.route(route="HTTPTrigger")
def HTTPTrigger(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
            logging.info(req_body)
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    if name:
        return func.HttpResponse(f"Hello, {name}. This HTTP triggered function
executed successfully.")
    else:
        return func.HttpResponse(
             "This HTTP triggered function executed successfully. Pass a name in the 
query string or in the request body for a personalized response.",
             status_code=200
        )
```

7. Now go back to Azure tab and in the RESOURCES section, click Azure Functions logo and select **Create Function App in Azure...** and set the following parameters.
    - Name: up to you.
    - Stack: Python 3.9
    - Location: Central US It will take some time for the function to be created.
8. Now go to the WORKSPACE tab and then click **Deploy to function App** -> Function you created.



9. You can visit [Azure portal]() -> Function App and select the function app you just created.



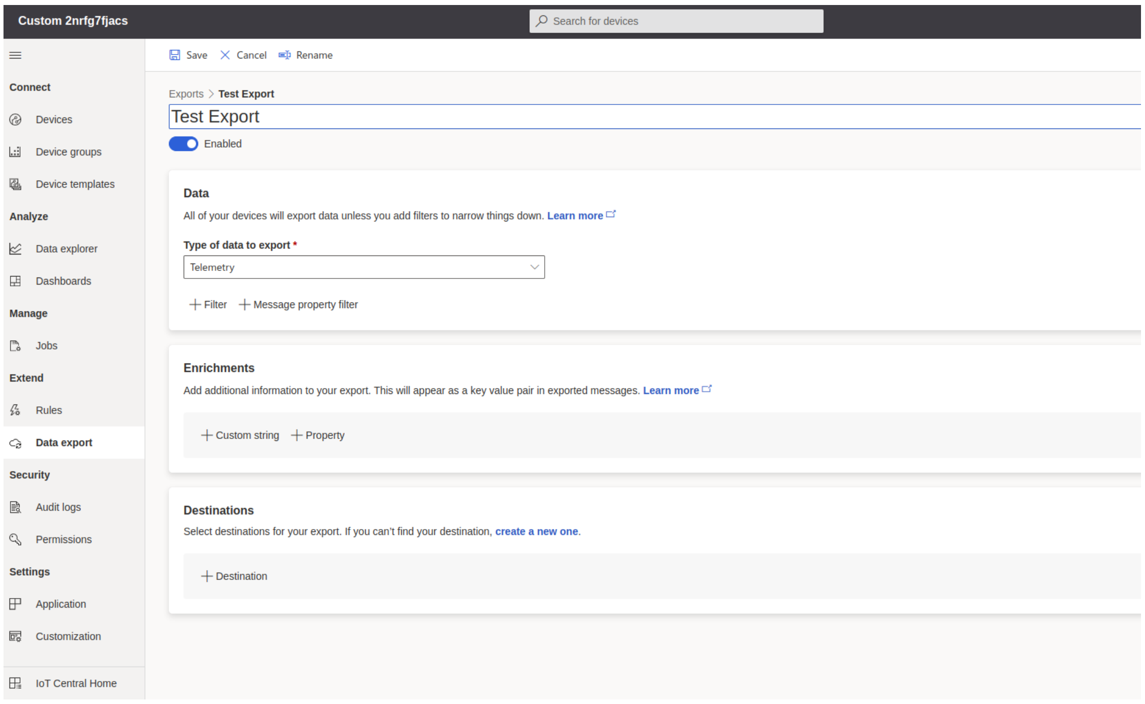
Resource group (move) : asdasdsadsadsa1123
Status : Running
Location : Central US
Subscription (move) : Azure for Students
Subscription ID : 8d24456d-42a8-4022-8a7e-b761cc98a64f
Tags (edit) : Click here to add tags
URL : https://asdasdsadsadsa1123.azurewebsites.net
Operating System : Linux
App Service Plan : ASP-asdasdsadsadsa1123-37f6 (Y1: 0)
Properties : See More
Runtime version : 4.11.2.2

10. Get the URL with API key from Functions -> HttpTrigger- >Code+Test-> Get function URL.
11. You can monitor the logs of function execution in Functions-> HttpTrigger-> Monitor -> Logs
12. If you need dependencies installed for your function to work, you can list them in the requirements.txt file.

Exporting IoT Central to Azure Function

Now, we will link IoT Central to the Azure function we created.

Go to the **Data export** section for your IoT Central application and click **+ New export** at the top.



Test Export

Enabled

Data

All of your devices will export data unless you add filters to narrow things down. Learn more

Type of data to export *: Telemetry

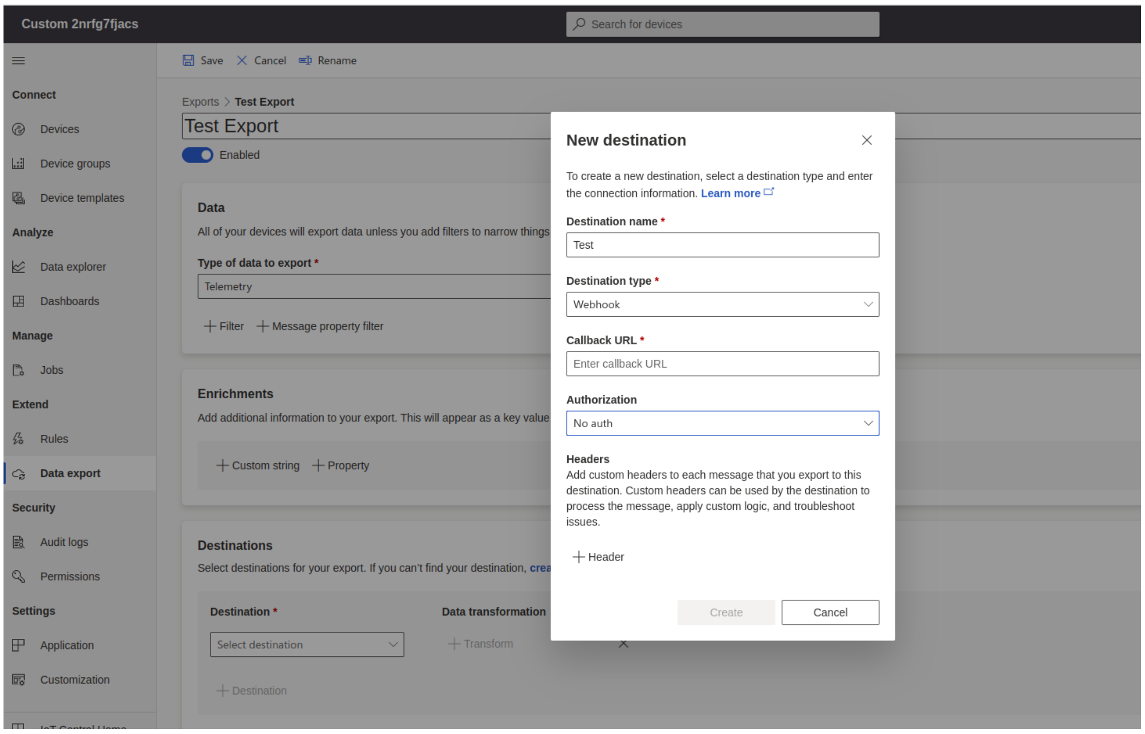
Enrichments

Add additional information to your export. This will appear as a key value pair in exported messages. Learn more

Destinations

Select destinations for your export. If you can't find your destination, create a new one.

In the destination section, **create a new one**. Select Webhook as the destination type and use the previously copied function URL in the callback URL field.



New destination

To create a new destination, select a destination type and enter the connection information. Learn more

Destination name *
Test

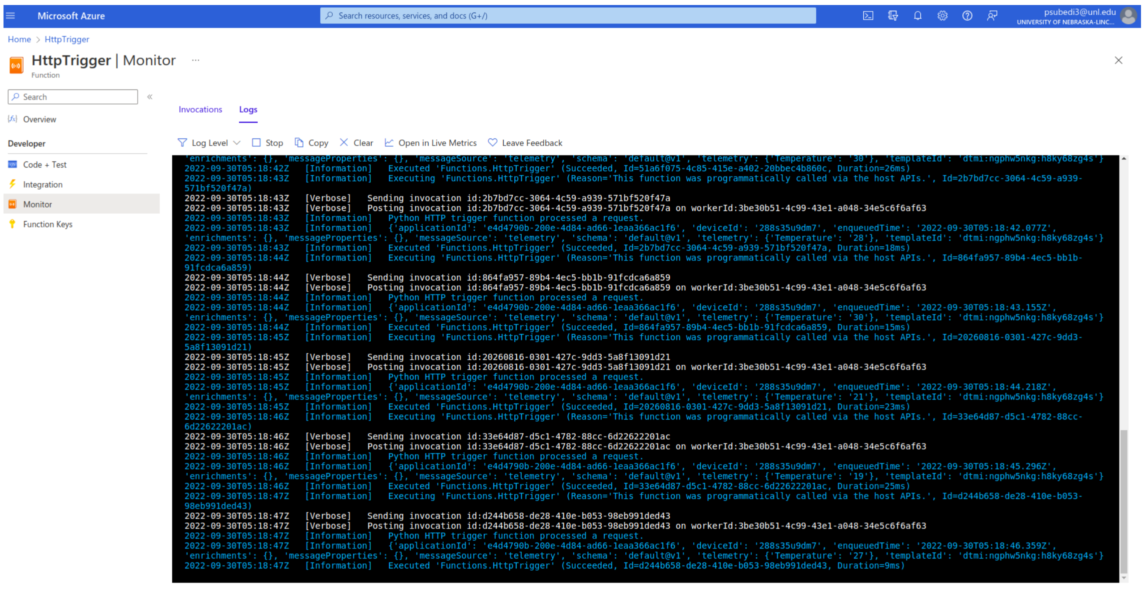
Destination type *
Webhook

Callback URL *
Enter callback URL

Authorization
No auth

Headers
Add custom headers to each message that you export to this destination. Custom headers can be used by the destination to process the message, apply custom logic, and troubleshoot issues.

. Run your Python-based virtual IoT device from Lab 5. See the logs for the function. You should see the relevant logs in the console. (It might take a while to show up after loading the screen).



```
'enrichments': {}, 'messageProperties': {}, 'messageSource': 'telemetry', 'schema': 'default@v1', 'telemetry': {'Temperature': '30'}, 'templateId': 'dtmi:ngphw5nkg:h8ky68zg4s'}
2022-09-30T05:18:42Z   [Information]   Executed 'Functions.HttpTrigger' (Succeeded, Id=51a6f075-4c85-415e-a402-20bbec4b860c, Duration=26ms)
2022-09-30T05:18:43Z   [Information]   Executing 'Functions.HttpTrigger' (Reason='This function was programmatically called via the host APIs.', Id=2b7bd7cc-3064-4c59-a939-
571bf520f47a)
2022-09-30T05:18:43Z   [Verbose]   Sending invocation id:2b7bd7cc-3064-4c59-a939-571bf520f47a
2022-09-30T05:18:43Z   [Verbose]   Posting invocation id:2b7bd7cc-3064-4c59-a939-571bf520f47a on workerId:3be30b51-4c99-43e1-a048-34e5c6f6af63
2022-09-30T05:18:43Z   [Information]   Python HTTP trigger function processed a request.
2022-09-30T05:18:43Z   [Information]   {'applicationId': 'e4d4790b-200e-4d84-ad66-1eaa366ac1f6', 'deviceId': '288s35u9dm7', 'enqueuedTime': '2022-09-30T05:18:42.077Z',
'enrichments': {}, 'messageProperties': {}, 'messageSource': 'telemetry', 'schema': 'default@v1', 'telemetry': {'Temperature': '28'}, 'templateId': 'dtmi:ngphw5nkg:h8ky68zg4s'}
2022-09-30T05:18:43Z   [Information]   Executed 'Functions.HttpTrigger' (Succeeded, Id=2b7bd7cc-3064-4c59-a939-571bf520f47a, Duration=18ms)
2022-09-30T05:18:44Z   [Information]   Executing 'Functions.HttpTrigger' (Reason='This function was programmatically called via the host APIs.', Id=864fa957-89b4-4ec5-bb1b-
91fcdca6a859)
2022-09-30T05:18:44Z   [Verbose]   Sending invocation id:864fa957-89b4-4ec5-bb1b-91fcdca6a859
2022-09-30T05:18:44Z   [Verbose]   Posting invocation id:864fa957-89b4-4ec5-bb1b-91fcdca6a859 on workerId:3be30b51-4c99-43e1-a048-34e5c6f6af63
2022-09-30T05:18:44Z   [Information]   Python HTTP trigger function processed a request.
2022-09-30T05:18:44Z   [Information]   {'applicationId': 'e4d4790b-200e-4d84-ad66-1eaa366ac1f6', 'deviceId': '288s35u9dm7', 'enqueuedTime': '2022-09-30T05:18:43.155Z',
'enrichments': {}, 'messageProperties': {}, 'messageSource': 'telemetry', 'schema': 'default@v1', 'telemetry': {'Temperature': '30'}, 'templateId': 'dtmi:ngphw5nkg:h8ky68zg4s'}
2022-09-30T05:18:44Z   [Information]   Executed 'Functions.HttpTrigger' (Succeeded, Id=864fa957-89b4-4ec5-bb1b-91fcdca6a859, Duration=15ms)
2022-09-30T05:18:45Z   [Information]   Executing 'Functions.HttpTrigger' (Reason='This function was programmatically called via the host APIs.', Id=20260816-0301-427c-9dd3-
5a8f13091d21)
2022-09-30T05:18:45Z   [Verbose]   Sending invocation id:20260816-0301-427c-9dd3-5a8f13091d21
2022-09-30T05:18:45Z   [Verbose]   Posting invocation id:20260816-0301-427c-9dd3-5a8f13091d21 on workerId:3be30b51-4c99-43e1-a048-34e5c6f6af63
2022-09-30T05:18:45Z   [Information]   Python HTTP trigger function processed a request.
2022-09-30T05:18:45Z   [Information]   {'applicationId': 'e4d4790b-200e-4d84-ad66-1eaa366ac1f6', 'deviceId': '288s35u9dm7', 'enqueuedTime': '2022-09-30T05:18:44.218Z',
'enrichments': {}, 'messageProperties': {}, 'messageSource': 'telemetry', 'schema': 'default@v1', 'telemetry': {'Temperature': '21'}, 'templateId': 'dtmi:ngphw5nkg:h8ky68zg4s'}
2022-09-30T05:18:45Z   [Information]   Executed 'Functions.HttpTrigger' (Succeeded, Id=20260816-0301-427c-9dd3-5a8f13091d21, Duration=23ms)
2022-09-30T05:18:46Z   [Information]   Executing 'Functions.HttpTrigger' (Reason='This function was programmatically called via the host APIs.', Id=33e64d87-d5c1-4782-88cc-
6d22622201ac)
2022-09-30T05:18:46Z   [Verbose]   Sending invocation id:33e64d87-d5c1-4782-88cc-6d22622201ac
2022-09-30T05:18:46Z   [Verbose]   Posting invocation id:33e64d87-d5c1-4782-88cc-6d22622201ac on workerId:3be30b51-4c99-43e1-a048-34e5c6f6af63
2022-09-30T05:18:46Z   [Information]   Python HTTP trigger function processed a request.
2022-09-30T05:18:46Z   [Information]   {'applicationId': 'e4d4790b-200e-4d84-ad66-1eaa366ac1f6', 'deviceId': '288s35u9dm7', 'enqueuedTime': '2022-09-30T05:18:45.296Z',
'enrichments': {}, 'messageProperties': {}, 'messageSource': 'telemetry', 'schema': 'default@v1', 'telemetry': {'Temperature': '19'}, 'templateId': 'dtmi:ngphw5nkg:h8ky68zg4s'}
2022-09-30T05:18:46Z   [Information]   Executed 'Functions.HttpTrigger' (Succeeded, Id=33e64d87-d5c1-4782-88cc-6d22622201ac, Duration=25ms)
2022-09-30T05:18:47Z   [Information]   Executing 'Functions.HttpTrigger' (Reason='This function was programmatically called via the host APIs.', Id=d244b658-de28-410e-b053-
98eb991ded43)
2022-09-30T05:18:47Z   [Verbose]   Sending invocation id:d244b658-de28-410e-b053-98eb991ded43
2022-09-30T05:18:47Z   [Verbose]   Posting invocation id:d244b658-de28-410e-b053-98eb991ded43 on workerId:3be30b51-4c99-43e1-a048-34e5c6f6af63
2022-09-30T05:18:47Z   [Information]   Python HTTP trigger function processed a request.
2022-09-30T05:18:47Z   [Information]   {'applicationId': 'e4d4790b-200e-4d84-ad66-1eaa366ac1f6', 'deviceId': '288s35u9dm7', 'enqueuedTime': '2022-09-30T05:18:46.359Z',
'enrichments': {}, 'messageProperties': {}, 'messageSource': 'telemetry', 'schema': 'default@v1', 'telemetry': {'Temperature': '27'}, 'templateId': 'dtmi:ngphw5nkg:h8ky68zg4s'}
2022-09-30T05:18:47Z   [Information]   Executed 'Functions.HttpTrigger' (Succeeded, Id=d244b658-de28-410e-b053-98eb991ded43, Duration=9ms)
```

Stop the app: Make sure you stop your Azure Function App to prevent exceeding your quota. Go to your Function App and click **Stop**.

Lab Assignment

The task for this lab is to combine all 3 things you learned here to create an end-to-end system.

Requirements

1. Randomly split out 1% of the data from the Rain in Austrailia dataset before doing any processing. Use the rest of the dataset for training and testing and save the final model.
2. Create an IoT central application with a device template for devices that send data like the provided dataset. It should also have a COMMAND defined.
3. Create an IoT Central device (Python based) that takes the 1% of the dataset not used for the model creation and sends data randomly to IoT Central.
4. Create another IoT Central device that only listens for a COMMAND.
5. Create an Azure function that runs on data exported from IoT Central. It should load the previously trained model and if the model predicts that it will rain tomorrow, send a command to the listening device.

Results

1. Screenshot and Python code that fulfills each device requirement in this lab
2. Screenshots from Azure for the completion of the requirements.

Report format

- Development Process
  - Record your development process.
  - **Acknowledge any resources that you found and helped you with your development (open-source projects/forum threads/books).**
  - Record the software/hardware bugs/pitfalls you had and your troubleshooting procedure.
- Results
  - Required results from the section above.

Submission Instructions:

1. Submit your lab on Canvas on or before the deadline (Oct 6, 8:29 am)
2. Your submission should include one single pdf explaining everything that was asked in the tasks and screenshots if any
3. Your submission should also include all the code that you have worked on with proper documentation
4. Failing to follow the instructions will make you lose points

References

https://www.analyticsvidhya.com/blog/2021/06/predictive-modelling-rain-prediction-in-australia-with-python/ https://learn.microsoft.com/en-us/rest/api/iotcentral/
https://learn.microsoft.com/en-us/azure/azure-functions/
https://learn.microsoft.com/en-us/azure/azure-functions/create-first-function-vs-code-python