D602 – Deployment

Performance Assessment #3 – Program Deployment

Bernard Connelly

Master of Science, Data Analytics, Western Governors University

Dr. Sherin Aly

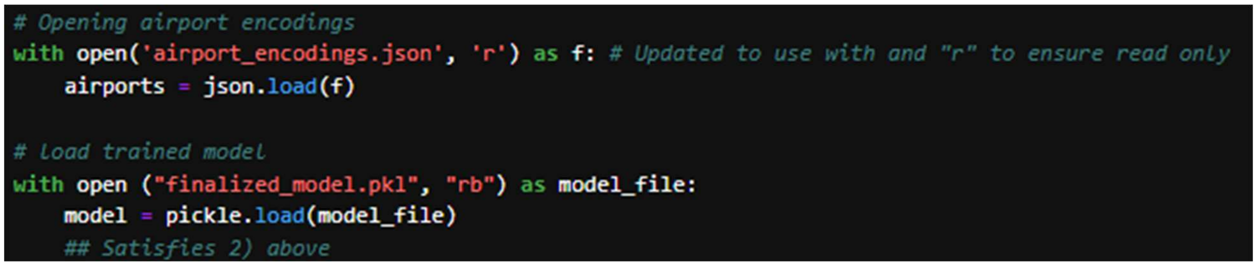
March 02, 2025

D602 Task 3 – Program Deployment

Creation of API & Tests

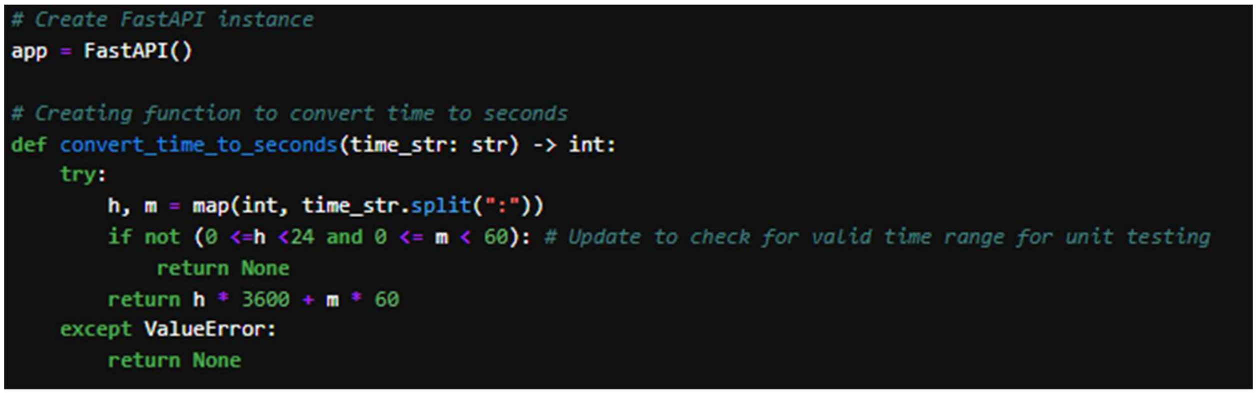
B. API via FastAPI

The creation of the API was a challenging task that used a variety of functions and methods with which I was unfamiliar. Breaking the task down into several components to meet the rubric made it more manageable, but the entire project was still an exercise in trial-and-error from start to finish. The individual coding prompts in the API narrowed the focus but added additional layers of complexity to the project. From a process perspective, I completed this task by first creating all of the code in an ipynb notebook in Jupyer Lab, then converting the code to a .py file and removing any details only relevant to a local instance. The first step was accessing and importing the previous project's JSON and pickle files.



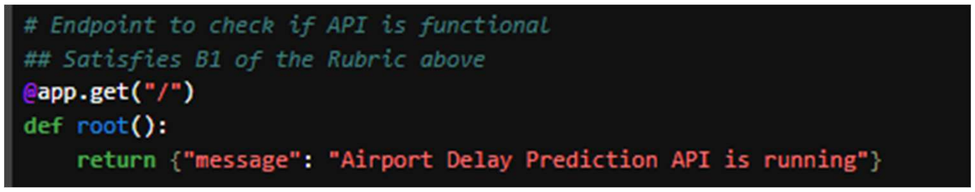
```python
# Opening airport encodings
with open('airport_encodings.json', 'r') as f: # Updated to use with and "r" to ensure read only
    airports = json.load(f)

# Load trained model
with open ("finalized_model.pkl", "rb") as model_file:
    model = pickle.load(model_file)
    ## Satisfies 2) above
```

The code for the JSON file was edited slightly to ensure it was taken as read-only based on best practices. Both loaded without issue. Afterward, the next step involved creating a function to convert the time to seconds:



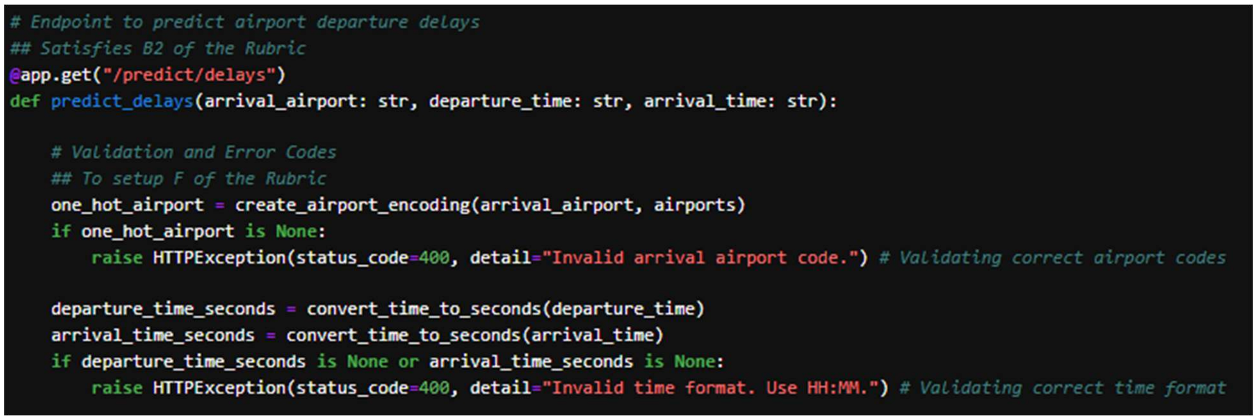
```python
# Create FastAPI instance
app = FastAPI()

# Creating function to convert time to seconds
def convert_time_to_seconds(time_str: str) -> int:
    try:
        h, m = map(int, time_str.split(":"))
        if not (0 <=h <24 and 0 <= m < 60): # Update to check for valid time range for unit testing
            return None
        return h * 3600 + m * 60
    except ValueError:
        return None
```

The above is the final iteration of the code. The initial attempt didn't require the if not statement to demonstrate a range, which was added later to debug the unit testing. After the above, I tackled creating the initial endpoint to illustrate that the API was up and running per part B1 of the Rubric:



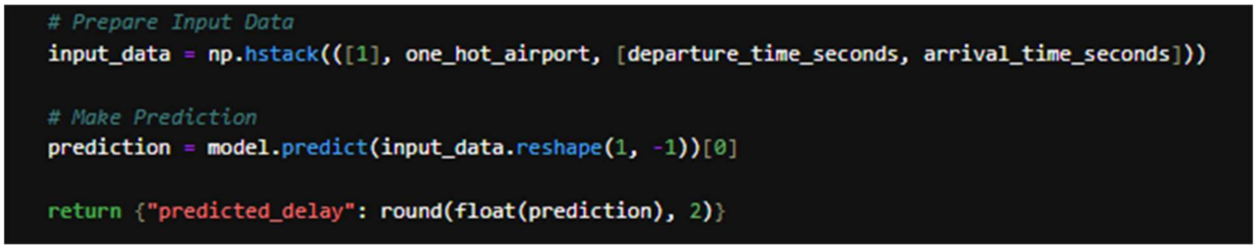
```
# Endpoint to check if API is functional
## Satisfies B1 of the Rubric above
@app.get("/")
def root():
    return {"message": "Airport Delay Prediction API is running"}
```

Afterwards, the endpoint for predictions was created using the function above. I simultaneously coded the error codes and validation steps for step F of the Rubric, as it made logical sense to complete both of these at the same time.



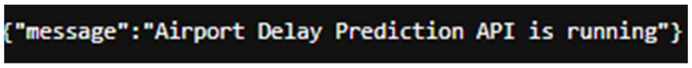
```
# Endpoint to predict airport departure delays
## Satisfies B2 of the Rubric
@app.get("/predict/delays")
def predict_delays(arrival_airport: str, departure_time: str, arrival_time: str):

    # Validation and Error Codes
    ## To setup F of the Rubric
    one_hot_airport = create_airport_encoding(arrival_airport, airports)
    if one_hot_airport is None:
        raise HTTPException(status_code=400, detail="Invalid arrival airport code.") # Validating correct airport codes

    departure_time_seconds = convert_time_to_seconds(departure_time)
    arrival_time_seconds = convert_time_to_seconds(arrival_time)
    if departure_time_seconds is None or arrival_time_seconds is None:
        raise HTTPException(status_code=400, detail="Invalid time format. Use HH:MM.") # Validating correct time format
```

With the validation and second endpoint complete, I completed the "to-do" list, which was housed in the original code, by ensuring the model was passed in a numpy array and making the prediction:



```
    # Prepare Input Data
    input_data = np.hstack(([1], one_hot_airport, [departure_time_seconds, arrival_time_seconds]))

    # Make Prediction
    prediction = model.predict(input_data.reshape(1, -1))[0]

    return {"predicted_delay": round(float(prediction), 2)}
```

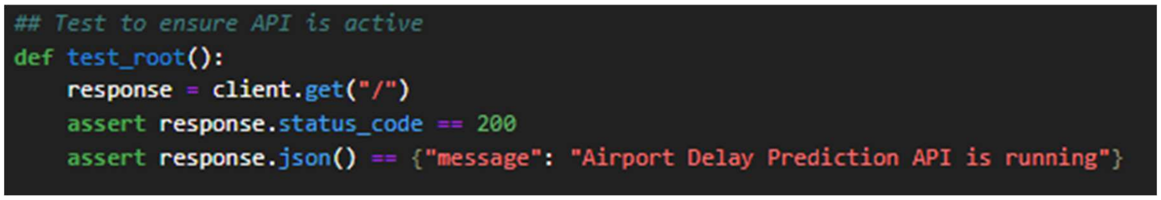
With the above complete, I could test the API in a local instance and confirm that the Airport Delay Prediction API is running. The message was returned successfully, completing this portion of the task.



```
{"message":"Airport Delay Prediction API is running"}
```

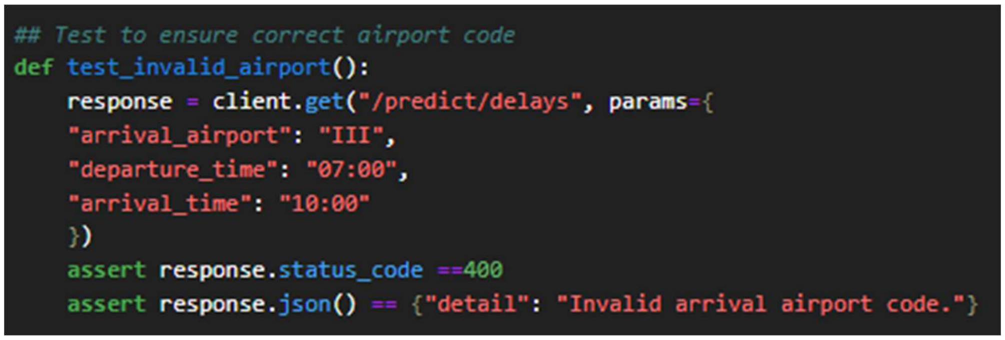
C. Unit Testing

Unit testing felt more familiar and straightforward than the creation of the API. To complete these steps, three separate tests were tackled – confirming the API was running, ensuring the correct format was utilized for the Airport Codes from the file, and confirming the correct date and time format were used for any API requests. Testing the root endpoint to ensure the API is active was tagged to the 200 HTTP Status code:



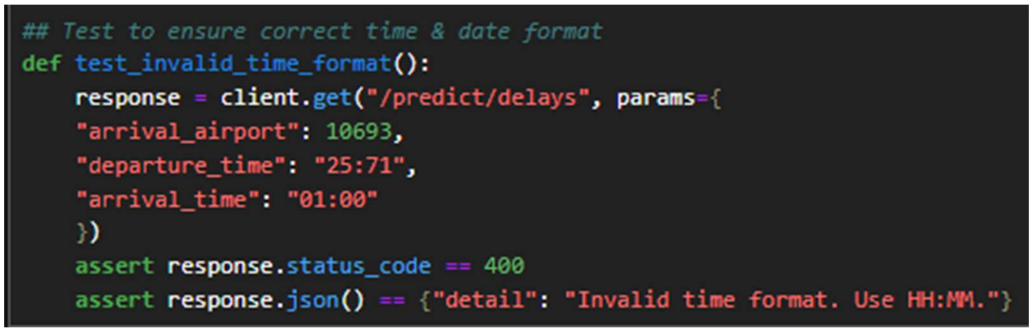
```
## Test to ensure API is active
def test_root():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Airport Delay Prediction API is running"}
```

Subsequently, the arrival airport code was validated with an incorrect value, "III," which would not map to any active airports, as these were all encoded with their 5-digit values, not the shorthand names.



```
## Test to ensure correct airport code
def test_invalid_airport():
    response = client.get("/predict/delays", params={
    "arrival_airport": "III",
    "departure_time": "07:00",
    "arrival_time": "10:00"
    })
    assert response.status_code ==400
    assert response.json() == {"detail": "Invalid arrival airport code."}
```

Finally, the date/time format passed into the initial API was flagged as HH: MM. To test this, an invalid number was selected for the departure time “25:71.”



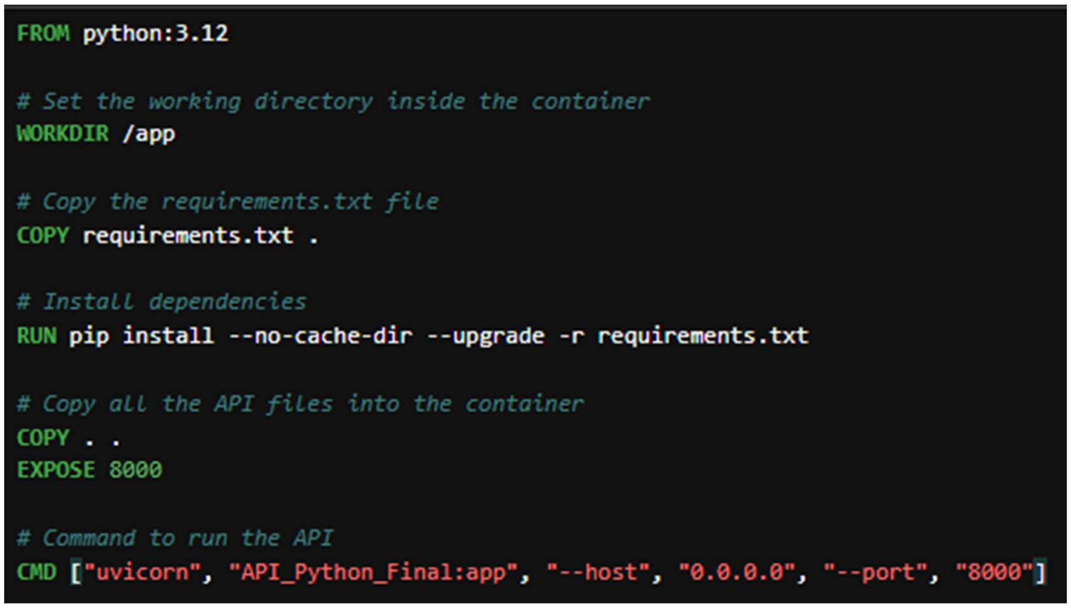
```
## Test to ensure correct time & date format
def test_invalid_time_format():
    response = client.get("/predict/delays", params={
    "arrival_airport": 10693,
    "departure_time": "25:71",
    "arrival_time": "01:00"
    })
    assert response.status_code == 400
    assert response.json() == {"detail": "Invalid time format. Use HH:MM."}
```

Like part B of this project, I validated these in a local instance to ensure the correct error codes would trigger. Once they cleared, the following steps were to get the relevant files into the GitLab repository and generate a Docker File to create a Docker Image to prepare for deployment to a container.

Deployment of API

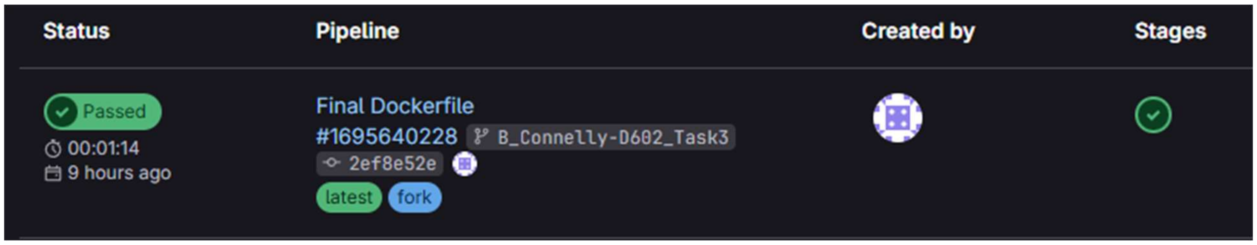
D. Docker File

The Dockerfile only required a handful of lines of code to be added to become functional. Most of these were generic components taught from the course resources. The file specified the base image for the container, pulling the Python 3.12 image from Docker Hub. Then, the working directory was set as /app, and the requirements listed in the requirements.txt file given were pulled into the relevant libraries to the container. The pip command was run to install all dependencies, all files were copied into the container, and finally, the port was exposed for HTTP requests. The final command to start the API included the uvicorn call for the FastAPI application, the specific Python file required, and the host and port.



```
FROM python:3.12

# Set the working directory inside the container
WORKDIR /app

# Copy the requirements.txt file
COPY requirements.txt .

# Install dependencies
RUN pip install --no-cache-dir --upgrade -r requirements.txt

# Copy all the API files into the container
COPY . .
EXPOSE 8000

# Command to run the API
CMD ["uvicorn", "API_Python_Final:app", "--host", "0.0.0.0", "--port", "8000"]
```

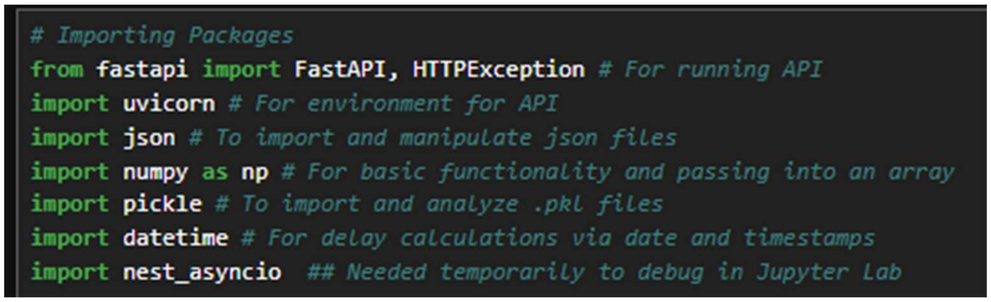
All the above details were pulled from the course documentation to configure correctly. The pipeline returned as functional after uploading this file along with the other relevant JSON, pickle, and .py files to the GitHub lab repository.



Status	Pipeline	Created by	Stages
✓ Passed 00:01:14 9 hours ago	Final Dockerfile #1695640228 B_Connelly-D602_Task3 2ef8e52e latest fork		✓

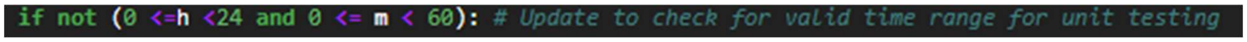
E. Challenges

Despite the straightforward details outlined in the above bullets, several difficulties had to be overcome before this project ran properly. The first piece was the code being set differently for the .ipynb local runs vs the final .py files. An additional package needed to be included, nest_asyncio, to ensure the API would run in the Jupyer Lab environment:



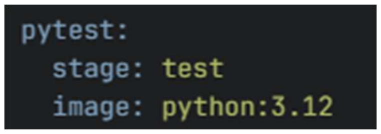
```python
# Importing Packages
from fastapi import FastAPI, HTTPException # For running API
import uvicorn # For environment for API
import json # To import and manipulate json files
import numpy as np # For basic functionality and passing into an array
import pickle # To import and analyze .pkl files
import datetime # For delay calculations via date and timestamps
import nest_asyncio  ## Needed temporarily to debug in Jupyter Lab
```

This item was removed from the final file, but the initial testing runs would not function without this step. When setting up the various Unit Tests, understanding the difference between the "message" and "detail" calls in each JSON response required several iterations and trials before they would respond adequately. Additionally, when setting up the date/time format test, a specific range had to be clarified in the original API file to ensure the test knew the relevant values to expect for the test itself:



```python
if not (0 <=h <24 and 0 <= m < 60): # Update to check for valid time range for unit testing
```

When uploading all files to the GitLab environment, there were multiple instances of the pipeline failing before I modified the GitLab-ci.yml file to call the 3.12 python image for pytest:



```yaml
pytest:
  stage: test
  image: python:3.12
```

Navigating to the live API component, I was unfamiliar with setting up and launching an API via one of the many available methods. I looked into Google Cloud services, Railway, and GitLab's built-in containers, but these all posed too difficult or unclear on how to set up. I ultimately downloaded and installed the Docker desktop application to navigate this problem.

The final issues resolves were cleared using the FAQs for the course resources, as there were issues with access tokens and Docker properly hooking into GitLab to have the image deployed to a container. Installing the Docker application also took multiple attempts, as the installation would fail repeatedly because of WSL errors. This was mitigated by checking

Docker's documentation and ultimately downloading and installing WSL to ensure the installation would clear.

Ultimately, there were many challenges posed by this project. Still, I found it very engaging and helpful overall to have a more full grasp of how the deployment of applications and code work from the back end in web-based services, and the various steps were beneficial in highlighting these details at a high level.

References

No other sources were used outside of the WGU course materials provided