RESTful API Project

API Specification Document

Course: INFO U762 – 01F

Semester: Spring 2024

Instructor: Dr. Grover Walters

Date: April 17, 2024

David C. Miller

**Weather API**

* My Weather API is using the National Weather Service API located at [API Web Service (weather.gov)](https://www.weather.gov/documentation/services-web-api#/default/station_observation_latest).
* The base API URL is: [https://api.weather.gov/stations/{stationId}/observations/latest](https://api.weather.gov/stations/%7bstationId%7d/observations/latest)
* The station ID that I am using is: KCQT, with is Los Angeles.
* The complete API URL is: <https://api.weather.gov/stations/KCQT/observations/latest?require_qc=false>
* The data return is GEOJSON as a Dictionary data type.
* Python Code Sections:

API Pull:

api\_url = "https://api.weather.gov/stations/KCQT/observations/latest?require\_qc=false"

data = fetch\_data\_from\_api(api\_url)

Converting data from GeoJSON to GeoDataFrame to a List to be Queried:

gdf = geopandas.GeoDataFrame.from\_features([data])

gdf = gdf.\_\_geo\_interface\_\_

values\_list = list(gdf.values())

Extracting Current Conditions String Value from List:

tree\_obj = objectpath.Tree(values\_list)

CurrentCondition = tuple(tree\_obj.execute('$..textDescription'))

CurrentConditionValue = CurrentCondition[0]

print(CurrentConditionValue)

Extracting Current Temperature and converting to Float Value:

tree\_obj = objectpath.Tree(values\_list)

temp\_obj = tuple(tree\_obj.execute('$..temperature'))

temp\_value = str(temp\_obj)

tempC = temp\_value[39:]

tempC = tempC[:4]

tempC = float(tempC)

print(tempC)

Converting the Temperature from Celsius to Fahrenheit and rounding to a single decimal point:

tempF = round((tempC \* (9/5)) + 32, 1)

print(tempF)

The final values that will be passed on to the custom API will be tempF and CurrentConditionValue.

**Currency API**

* My currency API is using the ExchangeRate-API Free Currency API – [ExchangeRate-API - Open Access, No Key Required](https://www.exchangerate-api.com/docs/free)
* The API link provides a complete list of foreign currencies all calculated against the US Dollar.
* The API URL is : <https://open.er-api.com/v6/latest/USD>
* The data returns as JSON as a Dictionary data type.
* Value was returned by nested dictionary query.
* Python Code Sections:

API Pull:

api\_url = "https://open.er-api.com/v6/latest/USD"

data = fetch\_data\_from\_api(api\_url)

Extract the Euro conversation rate for one US Dollar. Setting a hypothetical amount of 1000 Euro’s to be converted and rounding result to 2 decimal places:

rates\_data = {}

EUR = 1000

USD = 0

For k in data.keys():

if 'rates' in k:

rates\_data[k] = data[k]

EUREXCH = rates\_data['rates']['EUR']

USD = round(EUR \* EUREXCH, 2)

print(USD)

The final value that will be passed on to the custom API will be USD.

**Brewery API**

* My brewery API is using the Open Brewery DB API – [Open Brewery DB | Documentation](https://www.openbrewerydb.org/documentation/#search-breweries)
* The API link provides a list of over 8250 breweries from around the world.
* The API URL is : <https://api.openbrewerydb.org/v1/breweries/search?query=los%20angeles>
* The data returns as JSON as a List data type.
* Python Code Sections:

API Pull:

api\_url = "https://api.openbrewerydb.org/v1/breweries/search?query=los%20angeles"

data = fetch\_data\_from\_api(api\_url)

Code to select BrewPubs, which have beer and food, and eliminate BrewPubs that do not have website or phone numbers listed:

df = pd.DataFrame(data)

df.to\_csv("brewery.csv", index = False)

df\_1 = df[df['website\_url'].notna()]

df\_2 = df\_1[df\_1['phone'].notna()]

df\_3 = df\_2[df\_2['brewery\_type'] == 'brewpub']

Code to randomly select a single entry from the results returned:

df\_4 = df\_3.sample(n=1)

Extracted the relevant values to pass on:

BrewName = df\_4.iat[0,1]

BrewPhone = df\_4.iat[0,12]

BrewURL = df\_4.iat[0,13]

The final values that will be passed on to the custom API will be BrewName, BrewPhone, and BrewURL.

**API Driver Application**

* My Driver Application will collect data from the 3 APIs above and populate the following variables:
  + Temp
  + Weather Description
  + Currency
  + Brewery Name
  + Brewery URL
  + Brewery Phone
* My Driver Application will use API Update to push these values to the Custom API.
* My Driver Application will use API Get to retrieve these values from the Custom API.
* Python Code Sections:

API Update:

BASE = " http://127.0.0.1:5000/"

APP\_VERSION = "v1/"

data = [{'wtemp': tempF, 'wdescription': CurrentConditionValue, 'currency': USD, 'brewname': BrewName, 'brewurl': BrewURL, 'brewphone': BrewPhone}]

for i in range(len(data)):

response = requests.patch(BASE + APP\_VERSION + "data/" + str(i), data[i])

print(response.json())

input()

response = requests.patch(BASE + APP\_VERSION + "data/2")

print(response.json())

API Get:

BASE = " http://127.0.0.1:5000/"

APP\_VERSION = "v1/"

**Custom API**

* My custom API will accept API Update requests from the Driver Application to update the following pieces of data in the DB:
  + Temp
  + Weather Description
  + Currency
  + Brewery Name
  + Brewery URL
  + Brewery Phone
* My custom API will accept API Pull requests to see the Travel related data from Los Angeles.
* Python Code:

#Import Dependencies

from flask import Flask

from flask\_restful import Api, Resource, reqparse, abort, fields, marshal\_with

from flask\_sqlalchemy import SQLAlchemy

#define application and database variables

app = Flask(\_\_name\_\_)

api = Api(app)

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///database.db'

db = SQLAlchemy(app)

app\_version = "v1/"

#create the data definition

class DataModel(db.Model):

id = db.Column(db.Integer, primary\_key=True)

wtemp = db.Column(db.Float, nullable=False)

wdescription = db.Column(db.String, nullable=False)

currency = db.Column(db.Float, nullable=False)

brewname = db.Column(db.String, nullable=False)

brewurl = db.Column(db.String, nullable=False)

brewphone = db.Column(db.String, nullable=False)

#outputs to log/screen to verify data visually

#Commented Out. I used the DB Browser from sqlitebrowser.org to confirm that DB looks good.

# def \_\_repr\_\_(self):

# return f"DataModel(wtemp = {wtemp}, wdescription = {wdescription}, currency = {currency}, brewname = {brewname}, )"

#run this statement the first thme to create the database structure

#db.create\_all()

#handle the incoming data request with a parser

#arguments for a put request

data\_put\_args = reqparse.RequestParser()

data\_put\_args.add\_argument("wtemp", type=float, help="Temp in Celsius", required=True)

data\_put\_args.add\_argument("wdescription", type=str, help="Weather Description", required=True)

data\_put\_args.add\_argument("currency", type=float, help="Currency in Euros", required=True)

data\_put\_args.add\_argument("brewname", type=str, help="BrewPub Name", required=True)

data\_put\_args.add\_argument("brewurl", type=str, help="BrewPub URL", required=True)

data\_put\_args.add\_argument("brewphone", type=str, help="BrewPub Phone Number", required=True)

#arguments for an update request

data\_update\_args = reqparse.RequestParser()

data\_update\_args.add\_argument("wtemp", type=float, help="Temp in Celsius", required=True)

data\_update\_args.add\_argument("wdescription", type=str, help="Weather Description", required=True)

data\_update\_args.add\_argument("currency", type=float, help="Currency in Euros", required=True)

data\_update\_args.add\_argument("brewname", type=str, help="BrewPub Name", required=True)

data\_update\_args.add\_argument("brewurl", type=str, help="BrewPub URL", required=True)

data\_update\_args.add\_argument("brewphone", type=str, help="BrewPub Phone Number", required=True)

#Map the types to columns extracted from the database object

resource\_fields = {

'id': fields.Integer,

'wtemp': fields.Float,

'wdescription': fields.String,

'currency': fields.Float,

'brewname': fields.String,

'brewurl': fields.String,

'brewphone': fields.String

}

#Set up the Resource Functions for CRUD

class Data(Resource):

#GET (READ in CRUD)

#@marshal\_with serializes output from the DB as a dictionary (json object) so we can work with it in python

@marshal\_with(resource\_fields)

def get(self, data\_id):

result = DataModel.query.filter\_by(id=data\_id).first()

if not result:

abort(404, message="Could not find data with that id")

return result

#POST (CREATE in CRUD)

@marshal\_with(resource\_fields)

def put(self, data\_id):

args = data\_put\_args.parse\_args()

result = DataModel.query.filter\_by(id=data\_id).first()

if result:

abort(409, message="Data id taken...")

data = DataModel(id=data\_id, wtemp=args['wtemp'], wdescription=args['wdescription'], currency=args['currency'], brewname=args['brewname'], brewurl=args['brewurl'], brewphone=args['brewphone'])

db.session.add(data)

db.session.commit()

return data, 201

#PUT (UPDATE in CRUD)

@marshal\_with(resource\_fields)

def patch(self, data\_id):

args = data\_update\_args.parse\_args()

result = DataModel.query.filter\_by(id=data\_id).first()

if not result:

abort(404, message="Data doesn't exist, cannot update")

if args['wtemp']:

result.wtemp = args['wtemp']

if args['wdescription']:

result.wdescription = args['wdescription']

if args['currency']:

result.currency = args['currency']

if args['brewname']:

result.brewname = args['brewname']

if args['brewurl']:

result.brewurl = args['brewurl']

if args['brewphone']:

result.brewphone = args['brewphone']

db.session.commit()

return result, 200

#DELETE (DELETE in CRUD)

def delete(self, data\_id):

abort\_if\_data\_id\_doesnt\_exist(data\_id)

del Data[data\_id]

return '', 204

#Register the Resource called video to the API (remember to change versions when making changes for submission)

api.add\_resource(Data, "/" + app\_version + "data/<int:data\_id>")

#Run the API body

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)