**RESTful API Project**

Jake Stavrakas

Department of Informatics, USC Upstate

INFO U 798-01W

Dr. Walters

March 13th, 2024

In this interoperability project, I am creating a RESTful API. REST API stands for representational state transfer application protocol interface (redhat).

My idea is to make my own card game by combining three cards into one called DigiYuMon. This is an amalgamation of three names Digimon, Yugioh, and Pokémon which are Japanese trading card games. Essentially, I want to pull from the three card APIs at random to create this new combination of a video game. This would be like a rock paper scissors game but with an additional twist to keep fans of all three card games entertained. If you pull a Yugioh card it will beat a Digimon card but a Yugioh card will lose to a Pokémon card. A Pokémon card will beat a Yugioh card, but a Pokémon card will lose to a Digimon card. Finally, a Digimon card will beat a Pokémon card but it will lose to a Yugioh card. Now if you pull two cards of the same type you play the card game the way the creators intended. Minus of course the additional cards such as energy in Pokémon or spell cards in Yugioh. You will just assume you can do anything you can with the card you pulled as if it had all the energy it needed but still adhere to rules such as HP and attack power.

I want to begin with the game I know the most which is Pokémon and this game will differ from the original slightly. In the original game, you need a 60-card deck with three different types of cards a Pokémon card, an energy card, and a trainer card (pdf). The Pokémon cards evolve but you need the base Pokémon to place down first, a Charmander would evolve into a Charmeleon but if you pull a Charmeleon you cannot place it down without the Charmander first. It is like building a house you need the foundation before you build the second floor. The energy card is attached to a Pokémon and allows it to use different attacks. Some attacks require three energy cards and it could be a match of different types of energy cards(). In DigiYuMon we will only be using the Pokémon card and assuming that the energy of that Pokémon is at max. We will also forego the need to have a primary stage Pokémon, meaning you can play a Charmeleon without having a Charmander first. This will make it very interesting as you could pull a weak starter Pokémon or a coveted legendary Pokémon to battle, which will make it a game of chance.

Now onto the Yugioh rules which I am less familiar with

Creating my API I had taken inspiration from the movie API we had made in class but with some tweaks. I created a Python file titled main.py and began to import the dependencies we used in class specifically the flask, and then flask\_restful including Api, Resource, reqparse, abort, fields, marshal\_with, and SQLAlchemy. I then defined the application variables to start with version 1 of this API. Now let me explain what these things mean, Flask is a web framework written in Python that lets you develop web applications easily([source](https://pythonbasics.org/what-is-flask-python/)). The items in flask\_restful allow me to create the API, which allows that API to return HTTP error codes such as 404 and marshaling which converts an object in memory into a form that can be sent as a message ([marshaling](https://hyperskill.org/learn/step/18204)).

Next I created a model titled MonsterModel using the (db.Model) function from SQLAlchemy. This allowed me to essentially create a data frame with the Monster ID, the monster name, the attack, and the hp or health points of the monster. I then created the database structure with db.create\_all which creates the tables I defined above and the database.(<https://flask-sqlalchemy.palletsprojects.com/en/2.x/quickstart/>)

Then I had to add the arguments for a put request, a put request is where I define what is being parsed into my API. In this I had the following code

monster\_put\_args = reqparse.RequestParser()

monster\_put\_args.add\_argument(

    "name", type=str, help="Name of the monster is required", required=True

The monster\_put\_args = reqparse.ReqestParser() enables adding and parsing multiple arguments in the context of a single request. So I can add multiple arguments like the name one underneath which I made a string type and required it with the required = True, a bit of Boolean logic at the end. I adjusted these for each as this was the name one it required a string of letters, for the hp argument I required the type to be int, or integer. After creating all the arguments for the put request I had to do the same for an update request using the code.

To map these to the columns I used the resource\_fields = {“id”: fields.String, etc}. after mapping these I have to set up the resource functions for CRUD an acronym for Create, Read, Upload, and Delete. To do this we created four functions one for each letter in CRUD. The Get is Read, Post is Create, Put or patch is Update, and Delete is Delete. We also included error codes with messaging such as 409 “Monster ID Taken” or 404 “Monster doesn’t exist, cannot update”.

Finally, I need to register the resource called Monster to the API with the code below Api.add\_resource(Monster, “/” + app\_version + “monster/<int:monster\_id>”) then we run the API body with and move onto the get and push parts of the api, that is it for the main.

The next step is to integrate the data from the three APIs into mine. The Digimon one comes in a data frame; I had to change some parameters to get it to work with the code. To do this I mimicked the code from our homework and tweaked it to import data from the public API into my own. I got the code to post to my API from the Digimon API, resulting in all the Digimon cards on the website being posted into my API. This totaled around 2000 Digimon and in my first version of this API, I kept it as is and ran the Pokemon and Yugioh monsters after those 2000 in the Monsters to make sure I could pull from all three. I decided after seeing this many Digimon in my project that I would need to limit the number of monsters in each to make the game more even. I decided on 33 monsters from each game and I made this decision after the Digimon put phase.

To set the Pokemon to be put after this I chose to make variables for starting\_id = 2776 and the num\_monsters = 33. I then made another variable titled monsters\_id that combined these two so that the new monsters added in would start at 2776 and only put in 33 monsters to keep the number of monsters from each small. I used the following code monster\_ids = [starting\_id + i for i in range(num\_monsters)] (<https://www.freecodecamp.org/news/python-for-loop-for-i-in-range-example/> ). If data: is the code we used in class and it checks to see if data exists and isn’t null or none if it has data it will return the Boolean value TRUE, and then the code will allow the data pulled into my API.

Sources:

<https://stackoverflow.com/questions/16476924/how-can-i-iterate-over-rows-in-a-pandas-dataframe>

Rename the column headers so it will fit into my own API’s classifications.

<https://www.geeksforgeeks.org/python-generate-random-string-of-given-length/>

End Point information: <https://medium.com/@asvinjangid.kumar/creating-your-own-api-in-python-a-beginners-guide-59f4dd18d301>

<https://stackoverflow.com/questions/61977076/how-to-fetch-data-from-api-using-python>

<https://rapidapi.com/blog/how-to-use-an-api-with-python/>

Publish to GIT: <https://code.visualstudio.com/docs/sourcecontrol/intro-to-git#:~:text=Use%20the%20Publish%20to%20GitHub,code%20to%20the%20remote%20repository>.

Potential Driver code

from your\_flask\_application import app

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

# Setting up any necessary configurations or environment variables

# You might also want to set up logging, error handling, etc.

# Running the Flask application

app.run(debug=True)

There are 37 YuGiOh cards so we can make this work!

So 33 cards from each game and I set the random to two numbers so its 1- 99.