Nick Cardin
Due 12/21/2021
Comp705

Project Writeup

The main idea for my project was to make an application where the user could find the cheapest flight no matter the airline, from point A to point B. In the end, I believe I managed to pull this off however there are always things I could add to make it look better. In terms of what I got done, one of the first things that I accomplished was making the Django project. When I started I had created an original blank Django project with the idea of using a different API that no longer existed by the second week. Once that API had been deleted I decided to kind of start over using the API I'm using now called Amadeus and I looked into using the cookie-cutter format for Django. This came with its own challenges however in the end I believe it made my project better. One thing to note about the cookie cutter I used is that it's slightly different from the one we used in the book for every cheese. The cookie-cutter format they give for that is explicitly for the everycheese project. After a couple of failed attempts using that I found a more general version that wasn't specifically geared towards the everycheese project but still provided me a base to work off of.

Once I had this base to work off of I started looking through the tasks I had made for myself and started developing the front end up to look nice. This was made easy due to learning about bootstrap and is what I used for the entire front end. I was able to grab date boxes to let the user pick a date as well as dropdown menus to pick which airport the user wanted to go off of. These dropdown boxes ended up giving me a bit of trouble at first as I learned how to use them more effectively. At first, I just had a hardcoded list of about five airports and then one option that was set by default but that you couldn't choose so as to have some sort of starting point. Speaking of starting points, I also wanted the first box to auto default to whatever the logged-in user’s default airport was based on what they had set in their profile. With that in mind, the authentication I used was just the default one for Django, also figured out how to use mailgun in order for my website to send emails even if only to a couple of users due to how mailguns sandbox works. From there I added a default airport field that used a dropdown to choose from the five airports I had hardcoded that was linked to the user model.

Eventually, these five airports weren't enough for the scope of my project since I knew my API could handle more airports than that, I just didn't know how to include more airports without adding every one myself which I had refused to do. What I ended up doing was scraping a Wikipedia page that contained the information that I wanted and using pandas to clean up the data to only include the name of the airport and its IATA code before turning it into a dictionary that I could use later. It's not the best way of doing it but to start I ran that scraping program once which returned a JSON formatted dictionary in a txt file which it would then get read from inside the user model to store all the airports I had. This ended up working for the user profile however that still really didn't allow me to get that information easily from the front end. So I went about getting a model set up specifically for the airports to be stored into. Getting the airports was easy enough for the user model since all I was doing was putting the list of airports into a choice box rather than having each airport as its own entry. As for the actual airport model I ended up having to learn about fixtures and how to load them in. After I had made the fixture it was easy enough to load them in so I could have it work on both Heroku and local since that list of airports won't really change for any reason. This allowed me to populate my database with the different airports which I was then able to iterate through on my homepage to create my list on the frontend for both the origin and destination lists.

Getting the information to be used with the third-party API however, was a different story. The API I ended up using didn't have a great way to use it with AJAX requests from what I was able to glean from the API documentation however what it did have was a python endpoint I could use to make my requests. So I used that to my advantage and moved my API calls into views where I then made a class for it. The first method was in charge of just calling the API using both the key and the secret that I got when I signed up for the API. After a bit of trial and error, I was able to figure out how to return the information the user had input on the first page into a dictionary that was then passed to my API call. From there I could just set the values needed for the API calls to the keys of the dictionary where the information could be found in their values. One problem I came into however was being able to just input no value for a return date if the user only wanted to find direct flight deals. I went through and tried setting the value to None which resulted in an error. I tried setting it to a blank string which also didn't work. What I ended up doing in the end was making two different calls, one containing the return date entry in the call and one without. From there it's just a simple if statement that if the return date value is not a blank string, then it would use the call with the return date and vice versa. After the call had been made, I went through and used pandas to extensively clean the data and only pull exactly what I wanted to display to the user and make the data look a little bit more user-friendly in terms of the names and values that the API returned. From here I returned the table as an HTML table which was easy enough due to pandas containing a to_html() function. From there I had an HTML table that would be returned and presented to the user however it didn't look that clean. What I ended up doing to fix this was to change my tables class on that page to be that of a bootstrap table that made the boxes a bit smaller and more streamlined while also allowing the user to hover over a table entry and have it highlight the entire row which helped it look a lot cleaner. Overall however in the end it still is just a table and if I were to go back and fix anything, it would probably be this and trying to figure out a way to make this information look a lot more user-friendly. Regardless, after this came implementing my own restful API. I had struggled to find what exactly I wanted to return to the user in the first place, however I ended up deciding on just returning the most recent query that a user had requested due to my data being different enough from just the third party API due to the extensive cleaning I had done to the data. I had the option of just sending back every single query from the past to the user from my restful API but I opted for just the most recent for space concerns. if I wanted to switch it however, it's just replacing one variable. Since I used cookie cutter to implement Django this made the process somewhat simple, otherwise just followed along with the quick setup guide that we went over in class to set up my restful API.

After this was completed my project was pretty much complete. The only other thing that I had a problem with was my understanding of how models worked and how they worked with the databases. I wasn't quite sure even after our classes explained them so they probably gave me the most trouble however after actually getting my hands dirty and troubleshooting… a lot of troubleshooting. Something finally clicked where I understood how models and how they relate to the databases. From there it was making sure I could replicate this in Heroku which always ended up giving me grief. However, as with the models/databases after working with it and some troubleshooting I was able to get Heroku working with all my features that I wanted. The main things I ran into in the beginning with Heroku was how the system variables worked but that turned out to not be much of a problem. The other big issue with Heroku that I found was that the preset variable it used for its connection to the PostgreSQL database was that it used a legacy URL that could not be changed in Heroku even if you deleted the database, made the variable and then made a different database as it would simply write over the custom made one I had set. I spent a couple of hours trying to troubleshoot that to no avail. What I ended up coming up with the next day, letting myself look at the problem with fresh eyes, was using the replace function in my settings that will replace postgre with postgresql meaning that as long as Heroku still uses the legacy naming scheme that doesn't work anymore, my code will change it to a working database URL no matter if the database changes or not.

Overall learning Heroku was a valuable experience in my eyes if not a tad bit frustrating at times however it ended up teaching me a lot from how to develop my frontend and figuring out how to return that data to the backend and how specifically Django worked with that.