Coursework 2 Report

Shamita Datta

201309120

sc19sd@leeds.ac.uk

COMP3011 Web Services and Web Data

1 Description of Flight API

For this project, I was required to make a Flight API that could be called by the flight aggregator to see what flights are available at the date, departure airport and destination airport supplied by the user, and then create the opportunity for a user to then create a reservation based on these details. These functions where implemented using `Python` and `Django`, and uploaded to `PythonAnywhere` so that the payment and flight aggregator services could also be incorporated. The link to the `PythonAnywhere` file is as follows: `http://shamitadatta2.pythonanywhere.com`, with all of the code available at `https://github.com/shamdatts/sc19sd_flights`.

2 Functionality

For each flight, there are five different URLs that can be called by the aggregator. Each URL allows the user to either manipulate their reservation or to search for flight details.

2.1 Query Flights: *"flights/query=<str:date>&<str:departureAirport>&<str:destinationAirport>/"*

This URL relies upon the user supplying a date, departure airport (written as the airport code e.g. LBA for Leeds Bradford Airport), and destination airport (which should also be written by its airport code). Once these three details are supplied, the function searches the database for flights from the origin to the destination on that date, and if a flight is found, it returns a 200 status code and a JSON response consisting of the flight and seat details. The flight details consist of the following fields:

- **flightId**: the primary key of that specific flight (for my flights, all of the IDs are prefixed with "SD").
- **planeModel**: this is the design of the plane, all of my flights use Boeing 747s.
- **numberOfRows** and **seatsPerRow**: as all of my planes are Boeing 747s, I kept these two values the same to keep the planes consistent. Each plane had 35 rows with 6 seats per row.

- **departureTime** and **arrivalTime**: the departureTime was randomly selected between 05:00 and 23:00, with a random minute of either being on the hour (00), quarter past (15), half past (30) or quarter to (45). The duration of the flight was randomly selected between 1 and 4 hours, and then added to the departureTime to create the arrivalTime.
- **departureAirport** and **destinationAirport**: these two fields were provided by the user in the URL query.

In order to have flights that the flight aggregator could query, I created some initial flights and stored them in my database. The flights were between the following airports:

- LBA (Leeds Bradford Airport)
- BHD (George Best Belfast City Airport)
- IPC (Isla de Pascua Easter Island Airport)
- CCU (Netaji Subhash Chandra Bose Kolkata International Airport)
- DXB (Dubai International Airport)
- SYD (Sydney Kingsford Smith Airport)

For the 30 days following 12th May 2023, we created a flight from each airport to each other airport once a day, at a random quarterly departureTime between 05:00 and 23:00. The arrivalTime was a random duration of 1-4 hours after this. This meant that on each day, there were 30 flights between all of these locations. Each flight has 210 seats, as each plane was a Boeing 747 with 35 rows and 6 seats per row. Each seat had a seatNumber and an allocated fixed seat price; the seat prices were the same throughout the whole plane, but could differ between location.

Once these records had been made and stored in the database, the query flights function took the date, departureAirport and destinationAirport provided by the URL and searched the database for any available flights. It returned a JSON response to the user and the appropriate code based on the results of the search. I used serializers in my functions to format the response as a JSON string, and then used a defined function to create the response that is returned to the user. If no flights are found, the page returns a 404 status error saying there are no flights found on this day. Additionally, as this URL requires users to *GET* data from the database, any other method (e.g. PUT, DELETE, etc.) will return a 405 status error.

2.2 Create a reservation: *"res/book/"*

This URL allows a user to create a flight reservation. The URL uses a POST method so receives the body of the request to create the reservation. The body must include the passengerId, the seatNumber they have selected, the flightId of their selected flight, and whether they have confirmed their payment or have agreed to bring hold luggage. If either the passengerId, seatNumber or flightId are not supplied, the page returns a 400 status error stating that they are missing a required field. If the seatNumber that they have supplied has already been taken by another reservation, the page returns a 409 error stating that that seat has already been taken. If the hold luggage or payment confirmed values are not as expected, the page returns a 405 status error stating that the fields have been inputted incorrectly. If none of these errors occur and

the seatId, passengerId, hold luggage, flightId and paymentConfirmed values all are valid, a reservation object is created and saved to the database. The details from this reservation are then returned to the user with a 200 status code, including the following fields:

- **reservationId**: each reservation for my flights begin with "SD" followed by their reservationId.
- **seatId**
- **holdLuggage**: either set to True or False
- **paymentConfirmed**: either set to True or False
- **passenger**:
  - **firstName**
  - **lastName**
  - **dateOfBirth**
  - **passportNumber**
  - **address**
- **flight**: definitions as described in Section 2.1
  - **flightId**
  - **planeModel**
  - **numberOfRows**
  - **seatsPerRow**
  - **departureTime**
  - **arrivalTime**
  - **departureAirport**
  - **destinationAirport**
- **seat**
  - **seatNumber**: selected by the user
  - **seatPrice**

If one of the objects (flight, seat or passenger) does not exist, the page returns a 400 status error and the error message. If for some reason the reservation can"t be created, the page returns a 404 status error. If any other HTTP method is used other than POST, the page returns a 405 status error.

2.3 Get a reservation: *"res/query=<int:reservationId>/"*

This function simply returns the details from the reservation supplied by the URL to the user. It gets the reservation from the database by reservationId and returns it in the format described in Section 2.2. It returns a 200 status code and the reservation details if successful; if the reservation cannot be found, a 404 status error is returned. If the method supplied is not a GET method, it returns a 405 status code.

2.4 Update a reservation: *"res/update/query=<int:reservationId>/"*

This URL uses the PUT method to update values of the reservation depending on the data supplied. It first checks if the reservation exists in the database by reservationId and returns a 404 status code if it cannot be found. It then loads the body of the request and reassigns all the details according to this. It saves the reservation to the database and returns a 200 status code and JSON Response with the reservation data formatted as mentioned in Section 2.2 to the user. If the seat or passenger do not exist a 404 status code is returned to the page; if any fields are missing a 400 status code is returned, and if there is invalid data in the body a 405 status code is returned.

2.5 Delete a reservation: *"res/delete/query=<int:reservationId>/"*

In a similar way to Section 2.4, this function deletes reservations from the database by reservationId. If it has been successfully deleted, a HTTP 200 status code is returned; if the reservation cannot be found a 404 status code is returned. If the HTTP method is not DELETE< a 405 status code is returned.

2.6 Confirm payment of a reservation: *"res/confirm/query=<int:reservationId>/"*

Also following the same format as Section 2.4 and 2.5, this function changes the status of the paymentConfirmed field in the reservation model. This can be used after the payment has been confirmed by the flight aggregator.

3 Implementation

In order to implement my API, I followed the API documentation that we created in Coursework 1 (`https://app.swaggerhub.com/apis/LUKEMCDOWELL2014/FlightsAPI/1.0.0#/`) and followed these details for each function. Each of my functions were created in the `Django` `views.py` file, and these functions were called by their specified URL.

3.1 Implementing the view functions

All of my functions followed a similar structure. I used many try/except statements to return the correct and valid JSON response and HTTP status code as described in our API documentation mentioned above, and used `Django's` serializers to put the valid responses from the functions into JSON strings. These strings were then passed into the helper functions that I created to help format the JSON response that would be sent to the user. As the only responses were about the flights

or the reservations, I created two helper functions (one for each) so that the responses exactly imitated our documentation outlined in Coursework 1.

3.2 Testing the functions

As specified in Section 2.1, I created many flights in my database and so testing the query flights function was simple. In order to create the flights and seats in the database, I created my own `manage.py` commands that I could run to manipulate the model; this allowed me to bulk create and update the flights instead of having to create them individually.

To test the other functions, as I could not have any initial reservations already in my system, I created test cases for successful and unsuccessful runs of the create, update, get, delete and confirm reservation functions. These test cases helped me certify that the functions worked before sending them to the flight aggregator and payment teams.

3.3 The Models

My flight API had four models associated with it, as originally defined in Coursework 1, however upon implementation the final Seat model included the additional SeatTaken field, which is False if the seat is available and True once someone has selected it in their reservation. The models are described below; colours represent foreign keys, and the bold field in each column is the primary key. The data types can be found in the Coursework 1 report or in the code supplied alongside this report.

FlightDetails	Passenger	Seat	Reservation
flightId	**passengerId**	**seatId**	**reservationId**
planeModel	firstName	flightId	seatId
numberOfRows	lastName	seatNumber	seatNumber
seatsPerRow	dateOfBirth	seatPrice	**passengerId**
departureTime	passportNumber	seatTaken	holdLuggage
arrivalTime	address		paymentConfirmed
destinationAirport			
departureAirport			

3.4 Coordinating with the Payment and Flight Aggregator Services

In order to make sure my API worked with the other parts of our web service, I remained in constant communication with the other 12 members of my team. We would message about issues in the code and fix them, and we all worked to make sure that the functionality followed the specification we had described in Coursework 1. It was quite difficult navigating so many different aspects in one, for example formatting responses between the aggregator and our flights were quite different as we did not properly define the requirements in Coursework 1. Additionally, I think we all could have managed our time better, as the creation of each individual API was fine but integrating them all together proved to be very tasking. Overall I am happy with our results and proud of our collaboration and further developed team skills as a result of this project.