COMP3011 Web Services and Web Data
Coursework 2

Bogdan-Alexandru Ciurea
sc20bac - 201438333

Introduction

The purpose of this document is to describe the high-level design of the application developed for the second coursework for (COMP3011) Web Services and Web Data, that being the creation of a flight aggregation software in Python.

This software was individually developed but is connected to other services deployed to *PythonAnywhere* by the team's (FLOC: Flight Listing Operational Consortium) members, therefore, communication, synchronisation, and standardisation between team members were a priority during the development of the services. These services can be of 5 different types: Airline companies, Car rental companies, Local guides, Accommodation services, and Payment providers.

This document will tackle the features that the software will have as well as the overall architecture of the software, thus providing a comprehensive insight into the code, before reading it.

Available commands

Although this implementation does not follow the implementation in the first coursework, it manages to utilise all the endpoints that are provided by the rest of the members. This being said, the resulted available command of the this implementation of the flight aggregator are going to be the following:

command	description
help	print this help message
show_bookings	show specific bookings
get_all_flights	list all available flights
get_flights	list flights with given parameters
book_flight	book a flight
cancel_booking	cancel a flight booking
search_car	search a cars
book_car	book a car
search_accommodation	search accommodations
book_accommodation	book accommodation
search_tours	search for tours
book_tour	book a tour
search_attractions	search for attractions
book_attraction	book for attraction
exit	exit the program

Table 1: List of possible commands

Code

This chapter will briefly explain the development process of the application and the architectural decisions of the code as well as design decisions that have been made along the way.

Environment

The environment that was used in the development process was on an ARM architecture running Python 3.10.1 and a Python virtual environment was used to ensure that the code can be as portable as possible.

Furthermore, a list of libraries that have been used in the development of this application to provide better CLI display options but also for features such as encryption or environment variables (we will discuss why they are necessary for a future chapter). The used libraries are `rich (12.6.0)`, `python-dotenv (0.20.0)`, `requests (2.26.0)`, `asyncio (3.4.3)` and `cryptography (40.0.2)`; this list can also be found in the `requests.txt` file inside the submission and can be used to install all of the required dependencies.

Before ruining the software it would be best practice to ensure that all the required files are present in the environment to ensure that the software will be working as expected, therefore the root folder should contain 10 files, those being two folders (`api` and `command_parsers` folders), one `.env` file, three python scripts (`card_manger.py`, `main.py`) and `card_manager.py`, `cards.txt`, `bookings.txt`, `requirements.txt`. The last file that is present is a Postman export that was used during development for testing the colleagues' endpoints and is named `postman_export.zip`.

Code architecture

The architecture of the code was important during the development process because, as so many services and endpoints had to be linked, it was we had to ensure that we can debug, change or maintain the code as easily as possible, therefore the decided approach was to use as many classes as possible, each one taking care of a specific service.

The architecture of this software can be split into 3 main parts:

1. API managers: Located in the `api` folder, these are the classes that will have a function for all of the endpoints of a service and will manage the request parsing or creation to return some `python data classes` that will contain the correlated information. In diagram 1, they are located on the 2^{nd} row (from top to bottom and including the APIs).

2. Command parsers: Located in the `command_parsers` folder, these are the classes that will read the information from the CLI and will print any errors to the screen. They are located in the 3^{rd} row plus the `card_manager` and `booking_manager.py` classes.

3. Main: The last class (located in `main.py`) will be responsible for connecting the command parses but also managing the main loop.

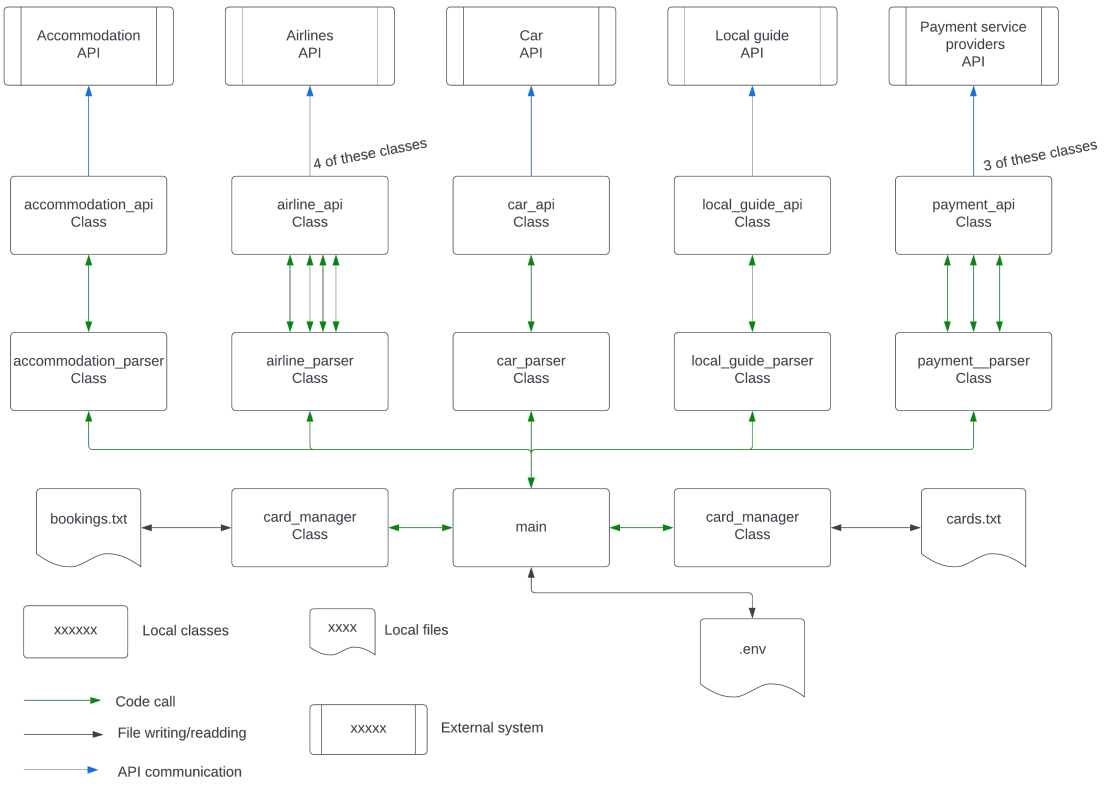

Figure 1: Architecture Diagram

Connection with the other services

To communicate with the other services constant communication and debugging were required between the team members. After this process, the software was successfully developed and could communicate with all services. One problem during the development and testing part was that Python was very sensitive if a JSON request did not have a specific key and it made the program crash, therefore, the code as, in all places that require request parsing, a try-except statement that will ensure that the code will not break but return a string that can be printed to the screen (this message can occur any time so if the console is showing an error but does not allow the user to input data, it is better to wait for it to finish loading).

Another way that testing was done was through Postman. Each endpoint was mapped in this software and they have been uploaded with the deliverable software to provide a backup for understanding the colleagues' work.

Results

Coming back to the used libraries that we mentioned in the previous chapter, because of the usage of the `rich` library, the resulted CLI output is beautifully arranged and colourised, but is also allows us to give the user default input (like `2` when selecting the airline) but also input that, if it not in an array of accepted strings (can easily be seen in the `luggage` input), it will not allow the user to continue. These arrays or checks are done for every input but it wouldn't be a good idea to display 40 different seats for example.

Two other features are the saved cards and the saved bookings. The first feature is done by the `cards_manager` and will store multiple cards in a local file; obviously, the card's details will be encrypted (using the `cryptography` library). The second one will be similar in implementation to the cards manager and will take care of storing and reading multiple bookings. One important aspect of the booking manager is that it is suppose to replace a functionality that is not supported by any of the services, therefore the software, although not the best option, will store them in a local file.

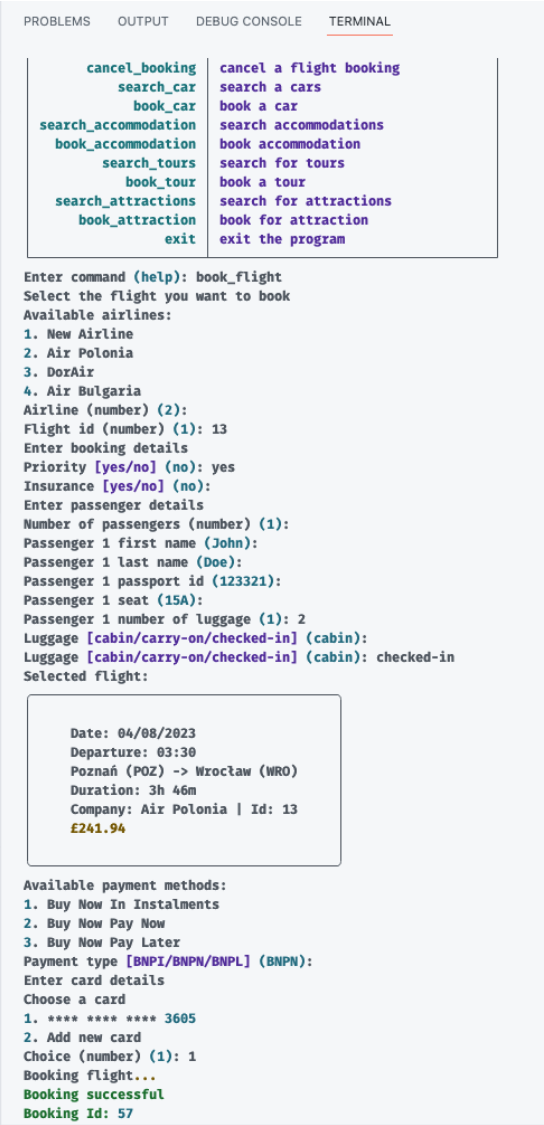

Figure 2: Resulted CLI output

In addition to this, we will have to keep in mind that the software will have some weak points, especially when waiting for the request's answers to come back from the other services. This can be instantly seen when launching the application as multiple responses will be awaited in order to make the application more immersive.

Changes from initial document

In the original document, the implementation asked for the aggregator to provide users with the option of selecting additional services at the destination's location. In this implementation, that feature was dropped because the databases were not large enough, therefore not guaranteeing the fact that we will have a service at that location.