Faculty of Engineering & Applied Science

Software Quality

Project Deliverable 1 -- Design & Testing

Course Instructor: Dr. Mohamed El-Darieby
Date Submitted: March 15th 2024

PROJECT GROUP NUMBER #: 2

Group Member	Student ID	Contribution
Jason Manarroo	100825106	System Design, Class Diagram, Detailed Design, Implementation Setup *(M.V.Cs, Prototype Test)*
Trent Jordan	100831853	System Design, Requirements, User Acceptance Testing, Sequence Diagram
Jordan Hagedorn	100828122	Requirements, Classes Skeleton Implementation, Testing
William Chamberlain	100846922	Unit Tests, Integration Tests
John Howe	100785128	High-Level Spring Diagram

Ticket Booking Application

For this project our group will use the principles of TDD to develop an application for booking flights from an airline company. To suit the course we will be using the java language within the spring framework to control our web application. Junit will be how our tests are conducted and maven will be our build. The version control will be contained in our git repository found here: https://github.com/jasonmzx/Software-Quality-Final-Project

As we progress through the project a number of tools will be explored for pipelining the implementation and release of the code we are producing.

Requirement Gathering:

Our team has been given and created some functional requirements to help us better understand what kind of software our clients want our group to build. These requirements are as follows in the list below:

1. The application has a web interface.
2. The application allows users to book direct or multi-stop flights.
3. The application allows users to book a one-way or round trip.
4. The application contains only a list of weekly direct flights.
5. The application reports the total flight time.
6. A 24-hour format is used in the application.
7. The ticket may use a 12-hour or 24-hour format per user preferences.
8. The application does not produce any cyclic trips from the same airport.

As part of the design of the architecture we arrived at some further functional requirements that would be needed to implement the system:

9. A search algorithm must be implemented to ensure that each customer will obtain the most efficient flight from start to finish.
10. A ticket display page with all information for the flights in a booking
11. Flights file import into Java

TDD methodology:

The TDD methodology allows for our team to produce high quality software that functions without bugs from product start to product finish. By using the TDD methodology our team will be designing tests for our software before, during, and after the implementation. By designing tests our team can ensure the software we produce is of high quality and fit for use. During this phase of the project, our team will design our system (in this document there are UML diagrams of the planned outcome). After this our team finished designing the test cases we would like our software to be able to pass. By doing this our team can have a clear understanding of what each function should be doing as the outputs are defined in the tests. Overall, our team uses TDD to make certain that our code is high quality and well made from beginning to end.

User Acceptance Tests:

User acceptance testing is the most subjective and difficult to quantify unlike the rest of the tests presented here. In order to remove subjectivity the user acceptance tests must ensure that the functional requirements are objectively met. Each functional requirement must then have its own test for user acceptance as given here:

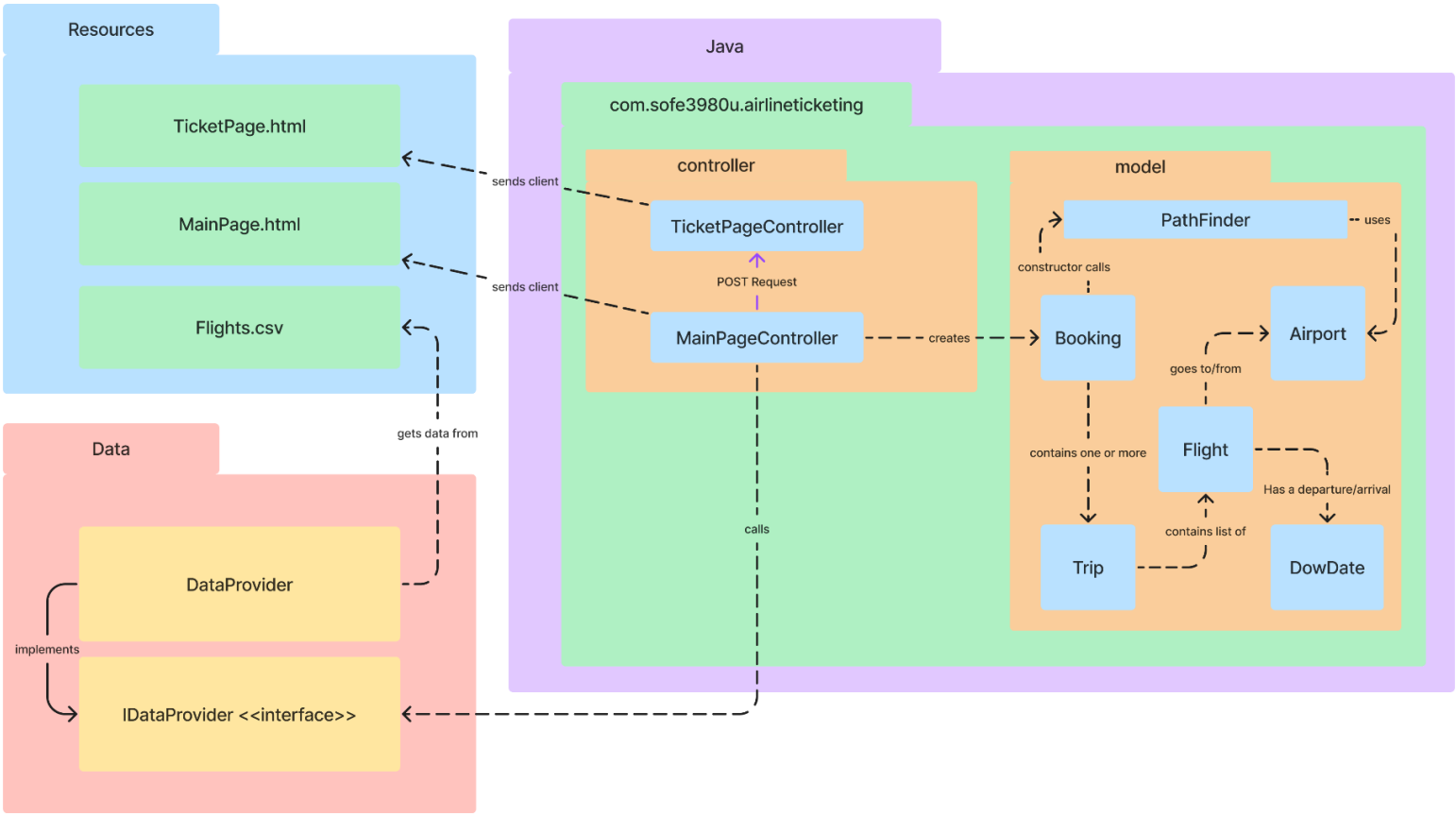
1. A user can access the web interface from any web browser
2. A user can book either direct or indirect flights successfully
3. A user can book one way or round trips successfully
4. Users are able to book flights within 1 week but not outside of that
5. When a flight is booked the total flight time is visible to users
6. 24 hour notation is used in the arrival and departure time when selecting a flight
7. Users are able to select 12 or 24 hour notation when receiving the ticket information
8. Users are unable to make a cycle when booking flights
9. Users are able to find indirect flights to their destination if no direct ones are available
10. Users are able to access all the relevant information when they make their bookings
11. Files are able to be uploaded to the application through an API or backend

High Level Design Considerations:

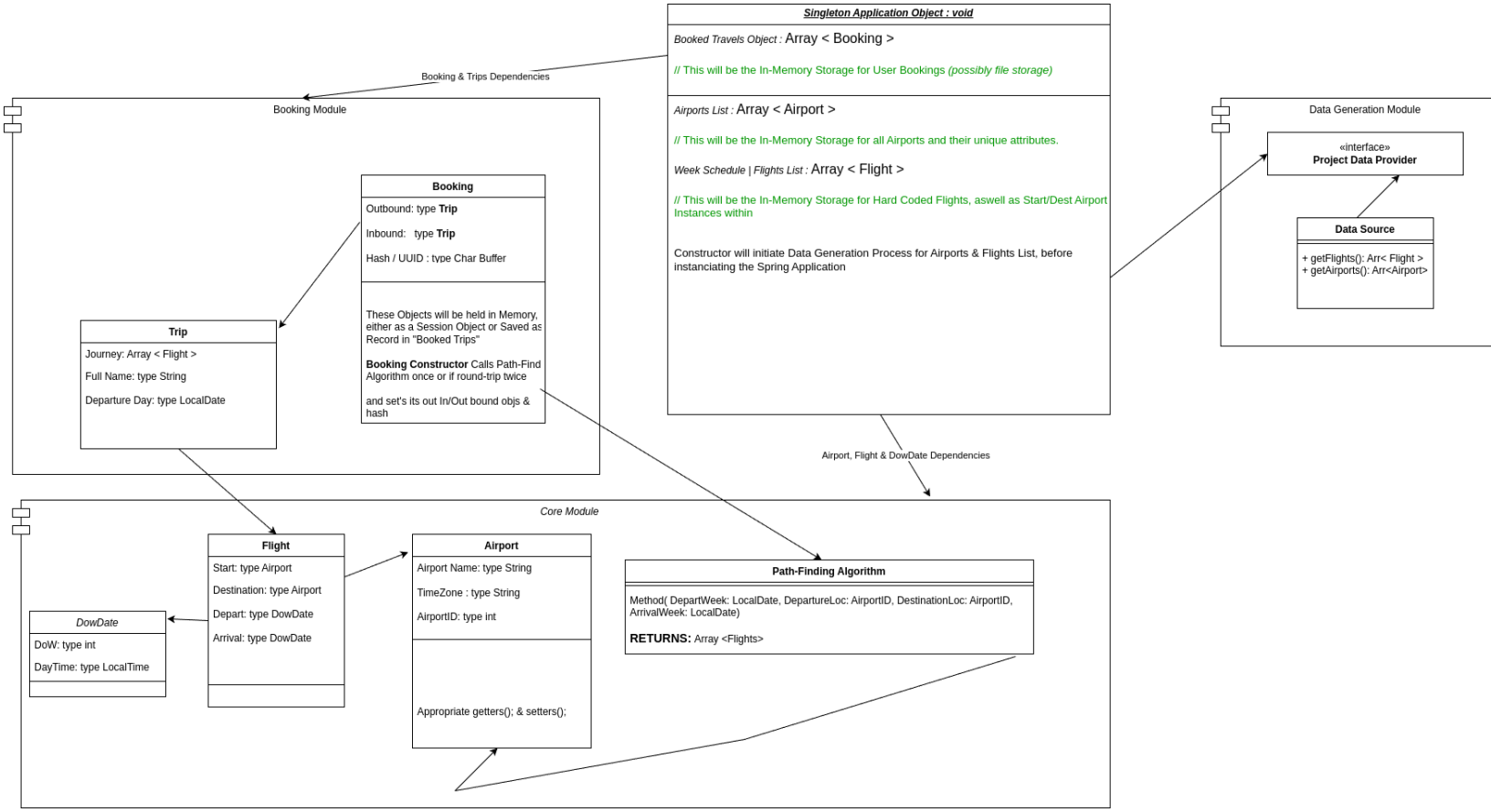
We want to keep dependencies and additional libraries to a minimum. Currently we're only using Spring Boot & JUnit as external dependencies, we've imported these via Maven, as our entire team is familiar with Java and the Spring framework using Maven due to the code assignments in this course. As this is a flight booking application with emphasis on Date/Time booking and time stamped ticket bookings and an inherent time constrained & involved project, we'll be using the built in Java Date/time libraries.

Diagrams:

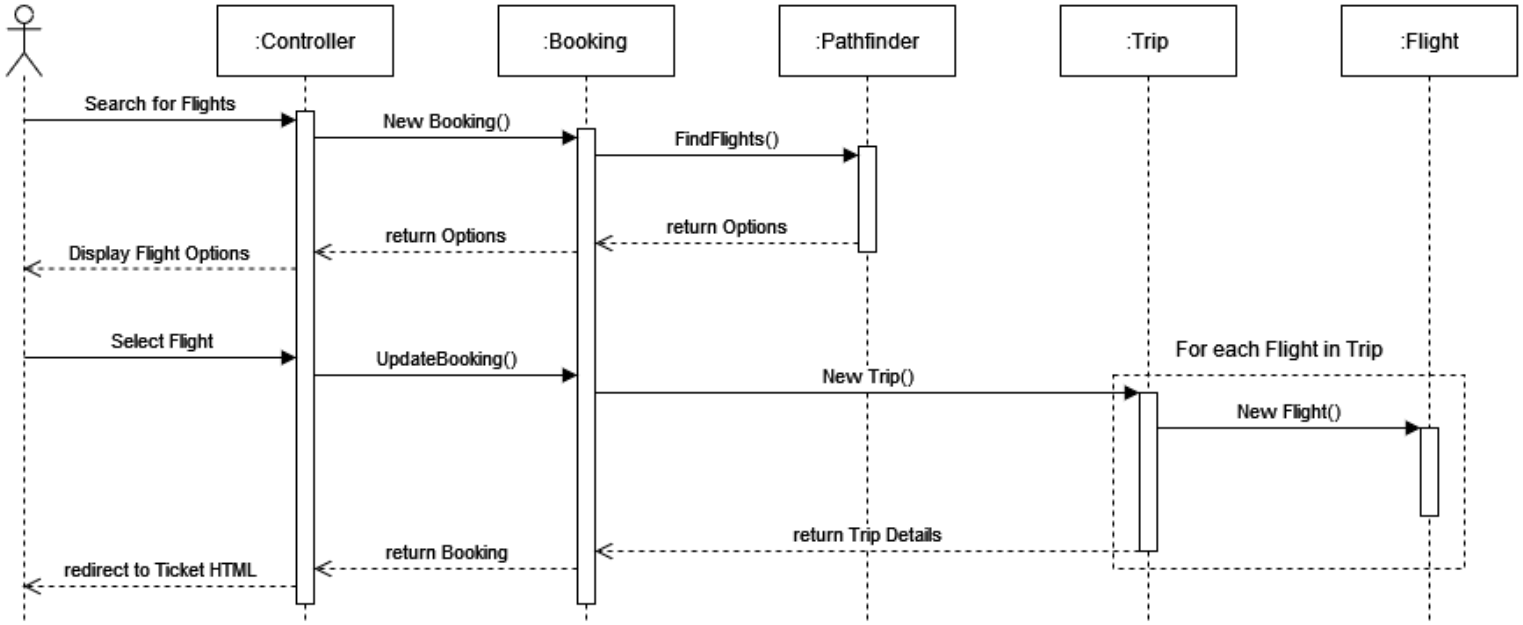
High-Level Spring Diagram:



Application Class Diagram, including Modules as well as a Singleton Object Instantiation



Sequence Diagram for Booking ticket



Detailed Design:

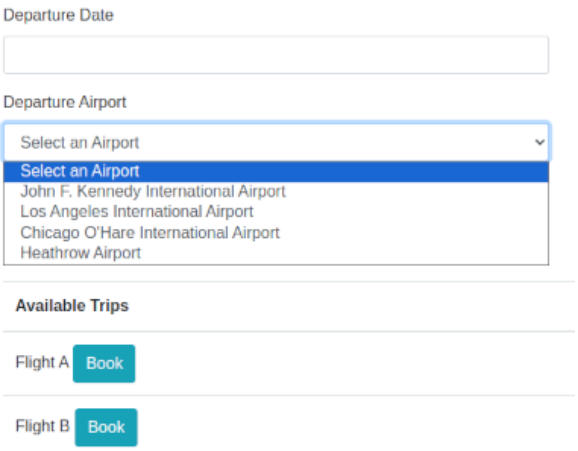
This project uses Java, Spring Boot framework (REST & MVC controllers), as well as the generic Java Date/time libraries to build this application. A thing to note is that we haven't implemented a user system, therefore all ticket orders will be differentiated as a UUID or some kind of unique identifier. Another limitation we chose for this project is an In-Memory Database represented as a Singleton Object, similar to State saved in a video-game as a mutable singleton object, or hierarchy of objects.

SOLID principles & Design pattern consideration given Java Spring's inherent design:
Here were utilizing the Singleton design pattern as it provides a good way to model a connection to a data source, here where using an Interface to segregate low level DB store logic with high-level getters and setters which the application relies on. *(as per the Interface Segregation principle)* This is very convenient for when we want to switch to a JDBC Database Pooling object which also acts like a Singleton!

Another consideration is that Spring Boot actually doesn't use objects or the traditional library approach of defining an API for the developer to use, rather they take advantage of the Java Method Decorators to Inject Dependencies directly onto a method, allowing for the implementation of said methods to be more-modular. I don't think it will be an issue working with the Singleton and the synchronized java features for thread safety!

Spring Boot also offers the Model View Controller design pattern (MVC) for their HTML Template Web Application system, where the controllers methods are dependency injected *(with @Controller for ex.)* and it returns the HTML templates with various attributes *(This attributes pulled from my Model, specifically my Singleton MemoryStore!)*

Quick Proof of concept Prototype:




```
public class MemoryStore {

    private static MemoryStore instance = null; // SINGLETON INSTANCE
    private List<Airport> airportsList;

    private MemoryStore() { //Private Constructor, as we'll only be building
        airportsList = new ArrayList<>();

        // Populate the list with some airports
        airportsList.add(new Airport("John F. Kennedy International Airport"
        airportsList.add(new Airport("Los Angeles International Airport", "A
        airportsList.add(new Airport("Chicago O'Hare International Airport",
        airportsList.add(new Airport("Heathrow Airport", "Europe/London", 4)
    }

    public static synchronized MemoryStore getInstance() {
        if (instance == null) {
            instance = new MemoryStore();
        }
        return instance;
    }
```

Singleton Prototyping works just fine!

Adhoc testing of mock-up frontend with placeholder flights, however getting our Airport List data into a dropdown menu on the View!

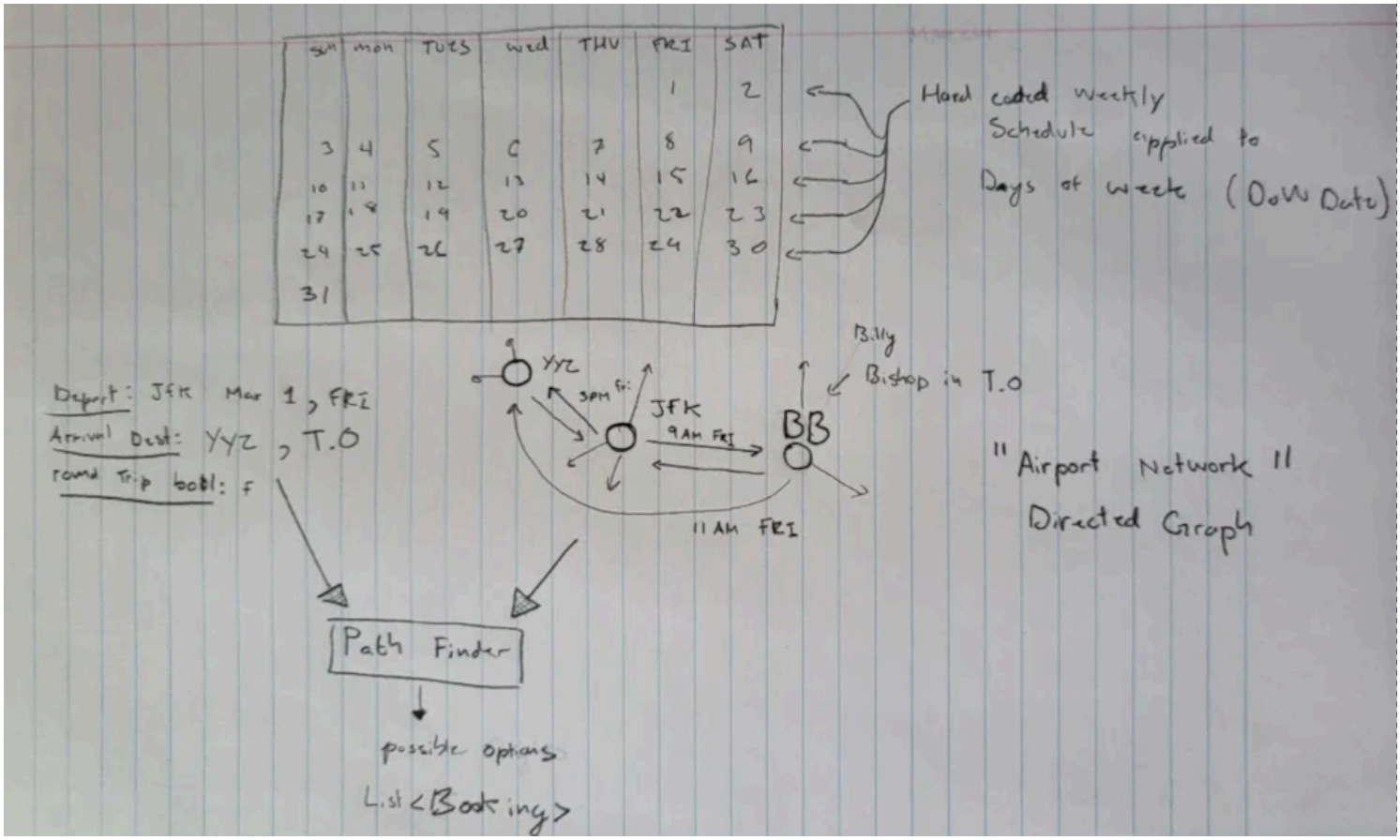
The **synchronized** keyword in Java ensures that only one thread can access a method or block at a time, making it crucial for thread-safe singleton creation by preventing multiple instances in a multithreaded environment. [docs.oracle.com]

This is just a best practice, I doubt our project will adapt threads

Okay so now we've got the entire stack of the application prepared and the MVC setup with Spring, but the actual functionality of the application isn't complete!

Below is a basic illustration of a potential view the user will see (The Datepicker) and how our weekly schedule will be mapped onto this schedule, matching Days of the Week, hence our DoWDate class.

Another thing to note is below, a showcase of the Model's Pipeline, getting a query for a specific trip and finding relevant trips the user might be able to book!

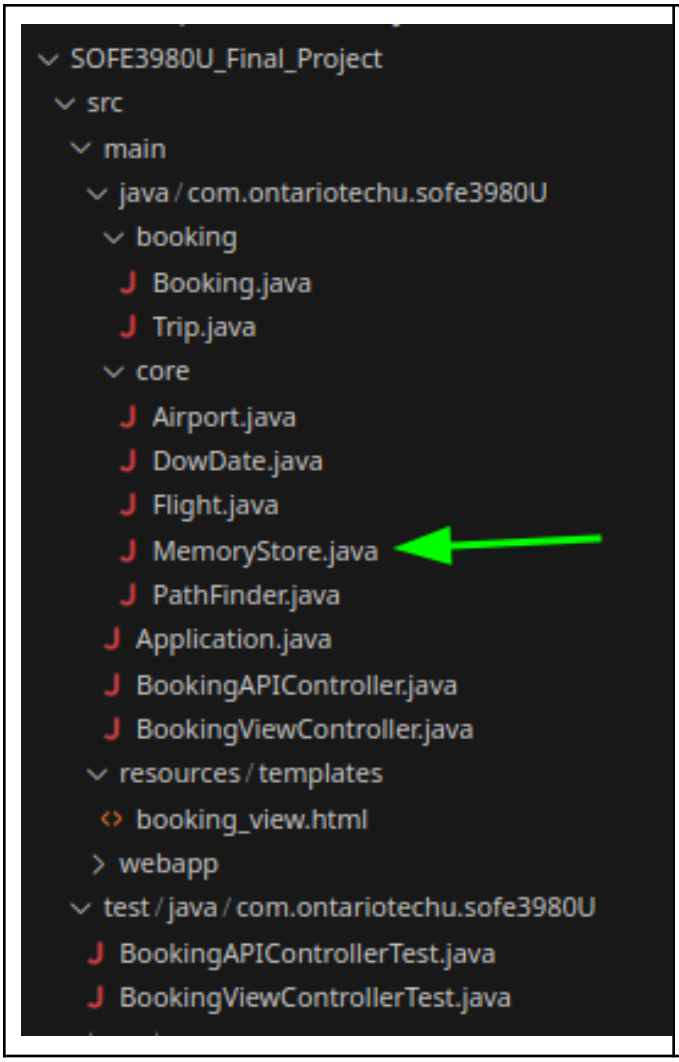


Here I've got the Month of March 2024 laid out on a 7 day calendar, this kind of visual will also be seen on the User view, as an HTML datepicker, as well as the Departure and Arrival Dropdowns discussed above.

This illustration also demonstrates an informal POST request depiction to our web service, essentially trying to find potential bookings for this chosen Trip. In the centre of the page notice the Airport Network which is a directed graph as well as having multiple links at varying times *(Ex: YYZ to JFK, and JFK to YYZ flight back and forth every 1.5 hours, from 8am to 8pm)* Obviously for simplicity we can model these as many non-recurring edges!

For now the Path Finding Algorithm will be kept simple, something like a sub-optimal yet reliable search algorithm like **Breadth-First-Search** and in the future we can drop in another search algorithm, maybe something like A* algorithm with heuristics for Cost, Time Zone differences, etc… as well as distance and monetary cost considerations *(Cost + Heuristic for A*)*

This will also use the *Interface Segregation Interface*, to allow pop & replace solutions for switching algorithms that maintain the interface's getters and setters.



As you can see here, in the core package, you can see a MemoryStore.java, this will hold information including but not limited ta:

- Possible Airports of Departure & Arrival

```
private static MemoryStore instance = null; // SINGLETON
private List<Airport> airportsList;
```

  MemoryStore is a synchronized instantiation, and here we can see the List of Airport objects, which also take time zones into account via **java.time**

- Airport Directed graph, on a 1 week schedule. **Example entry in this set:**
  - *Abu Dhabi, UAE to Doha, Qatar flies every Friday at 3:30 PM (15:30)*

- In Memory Record of Ticket Bookings composed of 1 or 2 Trip objects, with a unique identifier and a Memory Limit for practical purposes *(E.X: shut down booking feature after Booking Ledger is larger than 128 MB in memory or at X records long)*

- MemoryStore will inherit from an Interface which can be thought of as an agnostic provider of data, which can either be tied to a Database like JDBC's connection to SQL-like dbs, or just something simple like the in-memory one we'll be doing for this.

Future Steps & Design Considerations not yet addressed:

So we've set the stage for the basic implementation of everything, including an initial implementation of the classes making up the model, as well as the data source as a Singleton. The next steps are to:

- Make sure the **MemoryStore** & **PathFinder** classes are implementations of Interfaces for modularity and Dependency Inversion *(entities depend on abstractions rather than on other entities directly)*

- An Airport Network Directed Graph will need to be implemented into the MemoryStore for easy look-up during path finding. This may include different newer components or data-structures to efficiently search & find paths. Returning booking options

- General unique identification system for Booked Options, for easy re-visiting of the Booked Ticket view, given a special url possible, *example: localhost:8080/booking/3af68* will generate the booked ticket view with a booked instance matching this hex ID?

Unit Test Table:

Booking, Trip, Airport, DowDate, and Flight classes all have limited usefulness as far as testing since these are all data only classes. They provide no functional methods beyond getter and setter methods and are used for structuring and storing data. MemoryStore, BookingAPIController, BookingViewController, and particularly PathFinder are the classes that contain functional logic in this design, thus being useful test subjects. The Controllers are only really useful as integration or system tests however, as they contain no actual logic and are merely gateways to the core services.

MemoryStore

MemoryStore is the class responsible for retrieving and generating the flight and airport data.

Test Case	Method	Inputs	Expected Output	Rational
1	getInstance	none	MemoryStore Object	MemoryStore does not have a constructor as it is a singleton and should return an instance of itself.
2	getInstance 2x	none	Identical MemoryStore	As a singleton, the same MemoryStore object should be returned each time.
3	getFlights	none	List<Flight>	MemoryStore is to retrieve a list of flights from database/file.
4	getAirports	none	List<Airport>	MemoryStore is to retrieve a list of airports from database/file.

Pathfinder

The Pathfinder class is the heaviest class in the application and warrants the most testing to ensure correct results. It is responsible for interpreting the DAG formed by airport and flight objects to route a trip. The following tests are generated pairwise.

Test Number	**datasource**	**Airport a**	**Airport b**	**Depart**	**Arrive**	**Result**
1	valid	B	B	2	2	invalid
2	valid	null	null	null	null	invalid
3	null	B	null	1	1	invalid
4	null	null	A	1	2	invalid
5	null	A	A	2	null	invalid
6	null	A	B	null	1	invalid
7	valid	null	A	null	1	invalid
8	valid	A	B	1	2	valid, path AB, after time 1, before time 2
9	valid	B	A	2	1	invalid
10	null	A	null	2	1	invalid
11	null	A	A	null	2	invalid
12	null	B	A	1	null	invalid
13	null	null	B	1	1	invalid