Project Deliverable 2: Group Report

Faculty of Engineering and Applied Science

SOFE 3980U: Software Quality | CRN: 73385 | Section: 001

Due: April 2nd, 2024

Group 31

Rolf-Jaden Sibal (100845721)

Ontario Tech University

Oshawa, Ontario

rolfjaden.sibal@ontariotechu.net

Dmitri Rios Nadeau (100783206)

Ontario Tech University

Oshawa, Ontario

ericdmitri.riosnadeau@ontariotechu.net

Nathan Perez (100754066)

Ontario Tech University

Oshawa, Ontario

nathan.perez@ontariotechu.net

Noah Toma (100825559)

Ontario Tech University

Oshawa, Ontario

noah.toma@ontariotechu.net

Logan Butler (100828103)

Ontario Tech University

Oshawa, Ontario

logan.butler@ontariotechu.net

Table of Contents

Guidelines

Link to GitHub: https://github.com/D-aces/SQ_Project_Group31.git
Link to the video: Vid.mp4

The following are some of the application requirements:

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

Section 1: Adjustments

For the Trip class, some slight modifications were made. For instance, we now only have three main methods, except the three methods are `findFlightPath()` and `getFlightPath()` and `setFlightPath()` instead of `getFlightPaths()` and `getFlightTime()` respectively. To test the Trip class, a special case for testing is required. Since the class must call the database to get flight information, we have to use Mockito when testing the class in order to complete the testing for this class. Thus, we can make a "mock object" for each of these objects using Mockito. Using a library such as ASM, the fake object is a dynamically produced object that assumes the role of an instance of a class or implements an interface. Typically, when we run a JUnit test, we inform JUnit about our class so that it can examine it for annotations via reflection. We have added all of our methods that are marked with `@Test` to a list of test methods. It instantiates the class, executes any methods annotated with `@Before`, executes the test method, and then executes any methods annotated with `@After` for each test method. Upon detecting the `@RunWith` annotation, JUnit transfers all of its processing responsibilities to the distinct runner class and stops doing its own work. In this instance, the MockitoJUnitRunner uses reflection to investigate the class and generate mock objects for everything marked with `@Mock` before eventually invoking the standard JUnit code to perform the tests. Therefore, by the time we reach our setUp() method, we can be certain that the ds variable is not null, even if there is no code in the class that sets it.

Additionally, we added a new module class “database.java” to the system design. Using JDBC connections to a MySQL database, the database class manages CRUD operations on bookings and flights as a data access object (DAO) for a travel booking system. In addition to storing several booking types (MR, MO, DR, and DO) in the database, it provides ways to get all bookings for a user, search for particular flights, and query connecting airports. Establishing database connections, running SQL queries to get or alter flight and booking data, and managing SQL exceptions are some of the key functions.

In our system, we've adopted the Model-View-Controller (MVC) architecture, a design pattern that separates the application into three interconnected components. This separation enhances the manageability and scalability of our application by allowing independent development, testing, and maintenance of each component. Specifically, the 'Model' comprises our booking classes, which represent the data structure and business logic of our application. The 'View' is realized through our template HTML files, responsible for presenting the model data to the user. Lastly, the 'Controller,' our booking controller, acts as the intermediary between the Model and the View. It processes user inputs, manipulates data using the Model, and returns the output to be displayed by the View. This structured approach not only facilitates a clear division of responsibilities but also improves the application's adaptability to changes in business requirements or user interfaces.

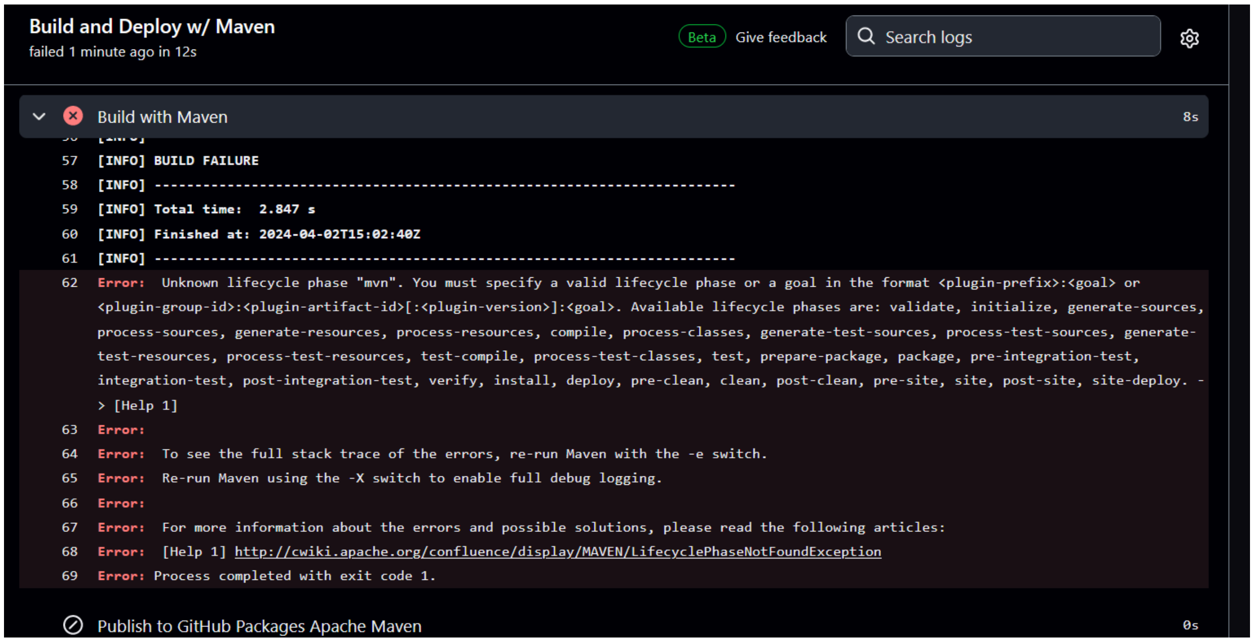
Some classes were removed, like the Ticket class. It was no longer a necessary component of the backend implementation, as it was ascertained that ticket generation could be considered a frontend issue. Therefore, it was no longer necessary to have a Ticket class for the purposes of direct model interaction.

Section 2: CI/CD Pipeline Setup

→ We must set up a new Java Project using Maven for the pipeline to work.

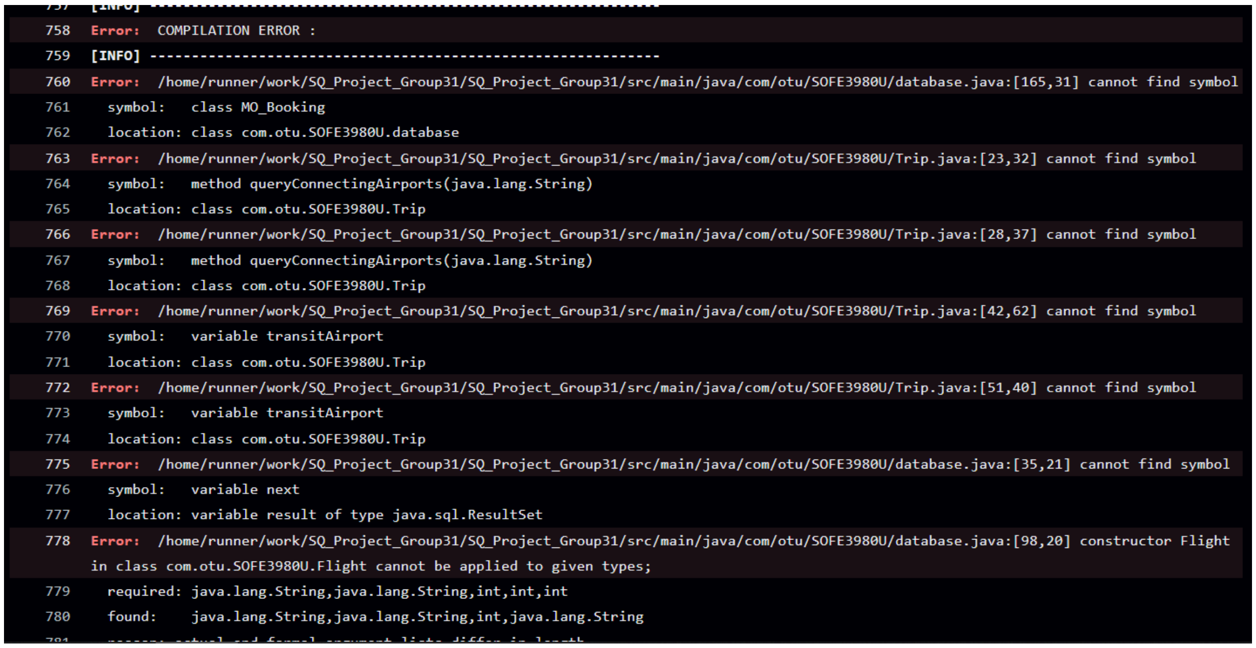
Using GitHub Actions, we can deploy our Java project using Maven and release it at our discretion.

For the CI/CD pipeline, our first attempt led to a build failure:



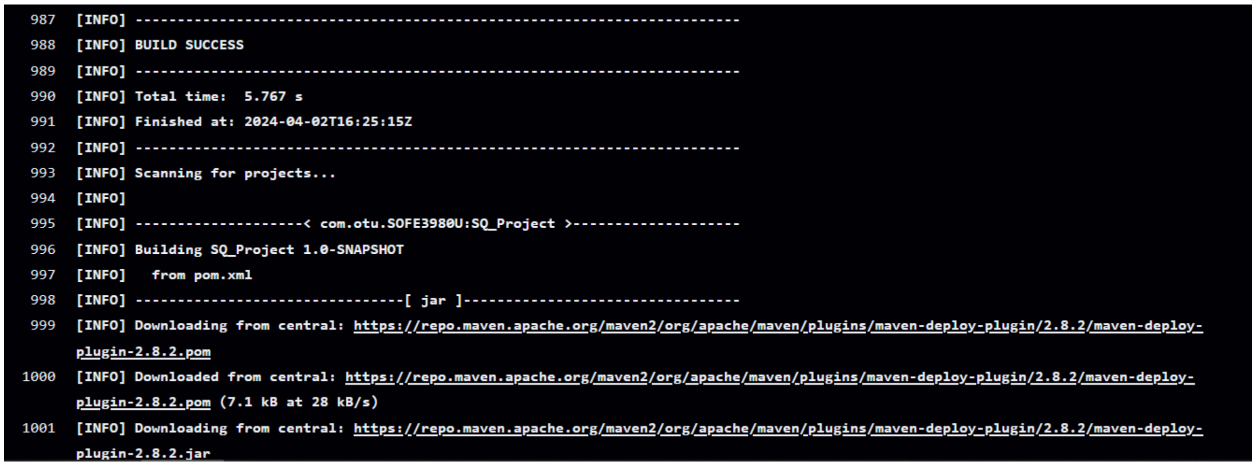
```
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.847 s
[INFO] Finished at: 2024-04-02T15:02:40Z
[INFO] ------------------------------------------------------------------------
Error:  Unknown lifecycle phase "mvn". You must specify a valid lifecycle phase or a goal in the format <plugin-prefix>:<goal> or <plugin-group-id>:<plugin-artifact-id>[:<plugin-version>]:<goal>. Available lifecycle phases are: validate, initialize, generate-sources, process-sources, generate-resources, process-resources, compile, process-classes, generate-test-sources, process-test-sources, generate-test-resources, process-test-resources, test-compile, process-test-classes, test, prepare-package, package, pre-integration-test, integration-test, post-integration-test, verify, install, deploy, pre-clean, clean, post-clean, pre-site, site, post-site, site-deploy. -> [Help 1]
Error:  
Error:  To see the full stack trace of the errors, re-run Maven with the -e switch.
Error:  Re-run Maven using the -X switch to enable full debug logging.
Error:  
Error:  For more information about the errors and possible solutions, please read the following articles:
Error:  [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/LifecyclePhaseNotFoundException
Error: Process completed with exit code 1.
```

After adjusting our YAML file, we were able to successfully build the project, but still have compilation errors:



```
Error:  COMPILATION ERROR :
[INFO] -------------------------------------------------------------
Error:  /home/runner/work/SQ_Project_Group31/SQ_Project_Group31/src/main/java/com/otu/SOFE3980U/database.java:[165,31] cannot find symbol
  symbol:   class MO_Booking
  location: class com.otu.SOFE3980U.database
Error:  /home/runner/work/SQ_Project_Group31/SQ_Project_Group31/src/main/java/com/otu/SOFE3980U/Trip.java:[23,32] cannot find symbol
  symbol:   method queryConnectingAirports(java.lang.String)
  location: class com.otu.SOFE3980U.Trip
Error:  /home/runner/work/SQ_Project_Group31/SQ_Project_Group31/src/main/java/com/otu/SOFE3980U/Trip.java:[28,37] cannot find symbol
  symbol:   method queryConnectingAirports(java.lang.String)
  location: class com.otu.SOFE3980U.Trip
Error:  /home/runner/work/SQ_Project_Group31/SQ_Project_Group31/src/main/java/com/otu/SOFE3980U/Trip.java:[42,62] cannot find symbol
  symbol:   variable transitAirport
  location: class com.otu.SOFE3980U.Trip
Error:  /home/runner/work/SQ_Project_Group31/SQ_Project_Group31/src/main/java/com/otu/SOFE3980U/Trip.java:[51,40] cannot find symbol
  symbol:   variable transitAirport
  location: class com.otu.SOFE3980U.Trip
Error:  /home/runner/work/SQ_Project_Group31/SQ_Project_Group31/src/main/java/com/otu/SOFE3980U/database.java:[35,21] cannot find symbol
  symbol:   variable next
  location: variable result of type java.sql.ResultSet
Error:  /home/runner/work/SQ_Project_Group31/SQ_Project_Group31/src/main/java/com/otu/SOFE3980U/database.java:[98,20] constructor Flight
in class com.otu.SOFE3980U.Flight cannot be applied to given types;
  required: java.lang.String,java.lang.String,int,int,int
  found:    java.lang.String,java.lang.String,int,java.lang.String
```

After fixing the compilation errors,



```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  5.767 s
[INFO] Finished at: 2024-04-02T16:25:15Z
[INFO] ------------------------------------------------------------------------
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< com.otu.SOFE3980U:SQ_Project >--------------------
[INFO] Building SQ_Project 1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
[INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.8.2/maven-deploy-plugin-2.8.2.pom
[INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.8.2/maven-deploy-plugin-2.8.2.pom (7.1 kB at 28 kB/s)
[INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.8.2/maven-deploy-plugin-2.8.2.jar
```

We were able to successfully deploy our project via GitHub Actions.

Picture of the tests running:

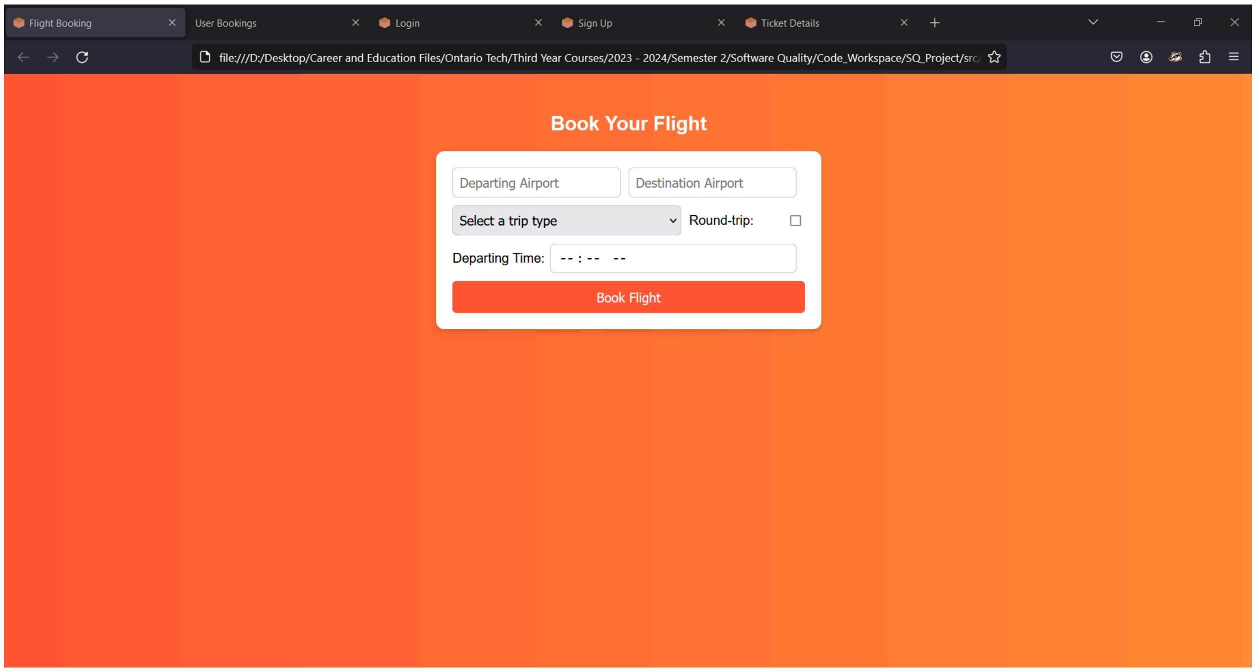
Section 3: Method / Module Implementation

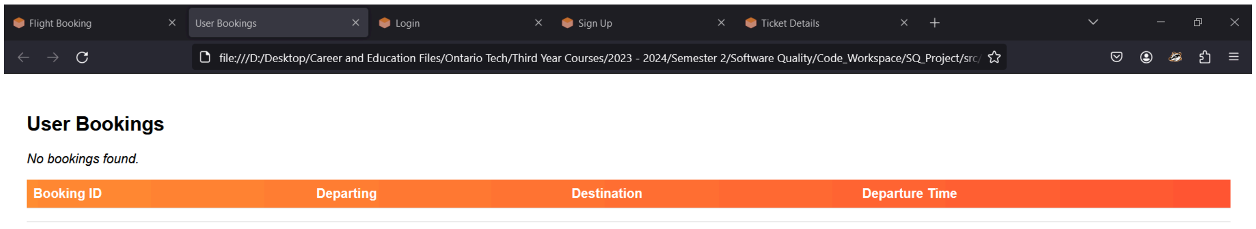
Added a repo with a new maven project: https://github.com/D-aces/SQ_Project_Group31

Class	*Method*	*Argument(s)*	*Return*	*Responsibility*
Airport	airport	Name, Connecting airports	N/A	Constructor used to instantiate an Airport object.
Airport	getConnectingAirports	None	String[]	For the given airport object this method will return an array of airport names that can be accessed via a direct flight.
Airport	getName	None	String	For the given airport object this method will return the airport's name.
Flight	flight	Arrival Time, Departing Airport, Destination Airport, Departing time, Flight Duration, ID	N/A	Constructor used to instantiate a flight object.
Flight	getArrivalTime	Arrival time	Integer	Method to calculate the arrival time for a flight.
Flight	getDepartingAirport	Departing airport	String	Returns the airport name the flight will depart from.
Flight	getDestinationAirport	Destination airport	String	Returns the airport name the flight will arrive at.
Flight	getDepartingTime	Departing time	Integer	Returns the time the flight will depart.
Flight	getFlightDuration	Flight Duration	Integer	For the given flight object this method will be used to get the time the flight arrives at the next airport.

Flight	getID	ID	Integer	Returns the ID of the flight will arrive at.
Trip	findFlightPath	Departing airport, Final destination	N/A	For the given trip object a string array with the possible flight paths from the departing airport to the final destination airport will be returned.
Trip	getFlightPath	trip	String	For the given trip object, returns the given flight path
Trip	setFlightPath	List <flights>[]	N/A	Sets the current flight path for the trip.
Booking	getTripType	None	Bool	For the given booking object the trip type (e.g. multi-stop or direct) if the trip is direct the method returns true, and false if the trip is multi-stop
Booking	getOneWay	None	Bool	For the given booking object the method returns true if the trip is one-way or false if the trip is round-trip
Booking	addFlight	None	Void	For the given booking object the method adds a new trip (if round-trip this is added to an array of trips).
Booking	getFlightTime	None	Integer	For the given booking object the total flight time for the booking is returned, (e.g. if a booking class of round-trip is used this may include the sum of both outgoing and returning trip bookings).

Subsection 3.1 HTML Templates and Controllers:

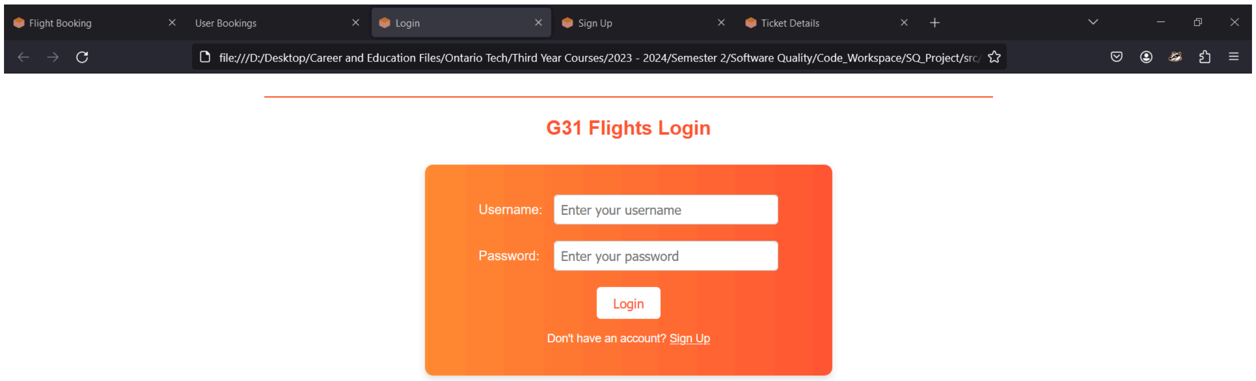


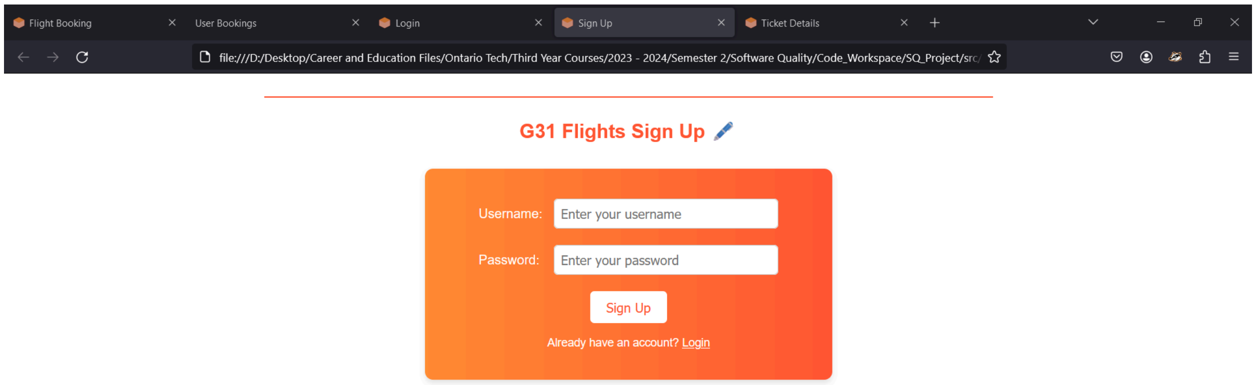

Flight Booking
User Bookings
Login
Sign Up
Ticket Details
file:///D:/Desktop/Career and Education Files/Ontario Tech/Third Year Courses/2023 - 2024/Semester 2/Software Quality/Code_Workspace/SQ_Project/src/

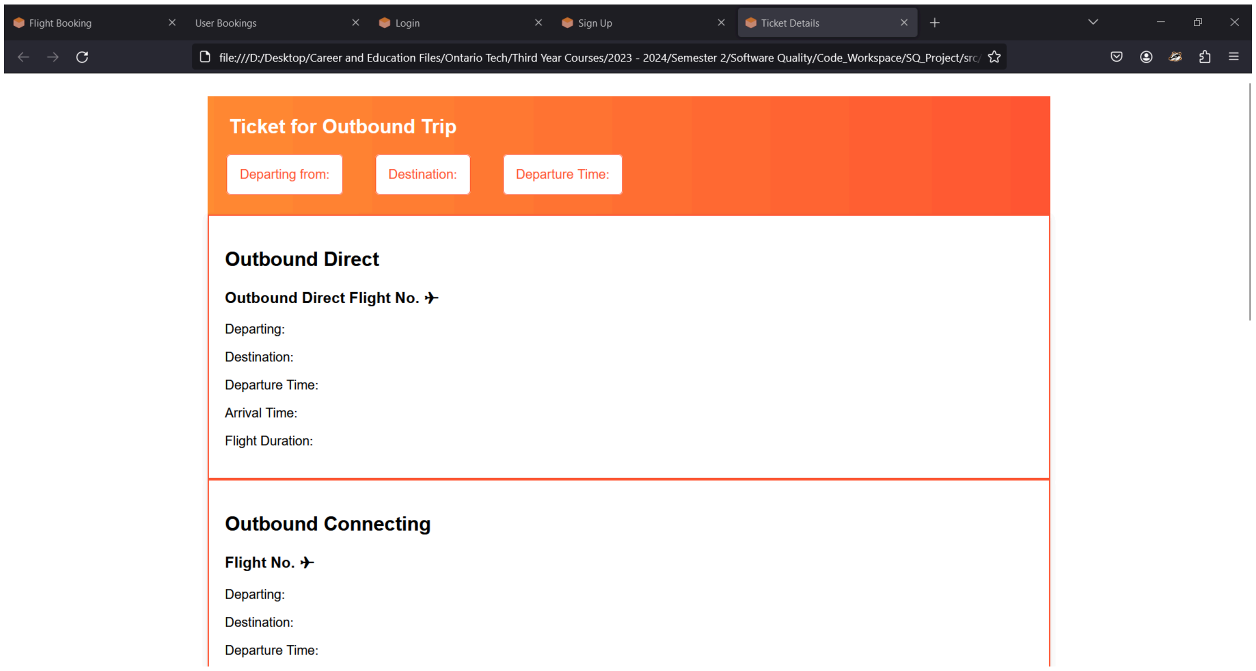
User Bookings

No bookings found.

Booking ID	Departing	Destination	Departure Time


Flight Booking
User Bookings
Login
Sign Up
Ticket Details
file:///D:/Desktop/Career and Education Files/Ontario Tech/Third Year Courses/2023 - 2024/Semester 2/Software Quality/Code_Workspace/SQ_Project/src/
G31 Flights Login
Username: Enter your username
Password: Enter your password
Login
Don't have an account? Sign Up


Flight Booking
User Bookings
Login
Sign Up
Ticket Details
file:///D:/Desktop/Career and Education Files/Ontario Tech/Third Year Courses/2023 - 2024/Semester 2/Software Quality/Code_Workspace/SQ_Project/src/
G31 Flights Sign Up
Username: Enter your username
Password: Enter your password
Sign Up
Already have an account? Login


Flight Booking
User Bookings
Login
Sign Up
Ticket Details
file:///D:/Desktop/Career and Education Files/Ontario Tech/Third Year Courses/2023 - 2024/Semester 2/Software Quality/Code_Workspace/SQ_Project/src/

Ticket for Outbound Trip

Departing from: Destination: Departure Time:

Outbound Direct

Outbound Direct Flight No. ✈

Departing:

Destination:

Departure Time:

Arrival Time:

Flight Duration:

Outbound Connecting

Flight No. ✈

Departing:

Destination:

Departure Time:

Section 4: Test Results

The master test plan involved the development and test methodology of a flight booking web app, making use of Java, Maven, JUnit, Spring Boot, and Thymeleaf. It served as a blueprint for our testing strategy and approach by outlining the framework within which all testing activities would be conducted in the project lifecycle.

The master plan defined the objectives, scope, and constraints of our testing efforts The reliability, functionality, usability, and performance of our flight booking web app across various scenarios and user interactions was explained.

Our master test plan included a detailed description of the techniques and tools that we would employ. We leveraged unit testing, integration testing, system testing, and acceptance testing to validate different aspects of our application JUnit served as our primary framework for unit testing, while integration testing involved testing the interaction between different components using Spring Boot. System testing focused on testing the end-to-end functionality of the application, and acceptance testing ensured that the app met the specified requirements and user expectations.

Our master test plan also outlined the test environment setup, including hardware and software required for testing. We used Maven to manage dependencies and build our project, ensuring consistency and reproducibility across different environments. Additionally, we leveraged Thymeleaf for front-end development, ensuring seamless integration with our Spring Boot backend.

The master test plan also included details on test data management, test case design, and execution procedures. We generated realistic test data to simulate various user scenarios and edge cases, ensuring comprehensive test coverage. Test cases were designed to validate both functional and non-functional requirements, including security, performance, and scalability.

Section 5: Application Deployment

The following project pipeline was successfully deployed with GitHub Actions; the continuous integration and deployment (CI/CD) workflow was defined and documented in a.yml file. Certain GitHub activities, such pushes and pull requests to specified branches or labeling a release, set off this process setup. It describes a series of tasks, such as the build process, in which Maven bundles the application, runs tests, and compiles the source code, and the deployment processes, which could include deploying straight to a production environment or uploading artifacts to a repository.

Encrypted secrets are used to securely handle environment variables and sensitive data, such as API keys or credentials, within the GitHub Actions process. This guarantees that such important data remain hidden and are only available to those components of the workflow that require them. Dependency caching is incorporated into the process, which greatly increases build time efficiency by reusing the same resources during various workflow runs.

The deployment process is designed in such a way that the workflow notifies the development team whether the build and deployment operations succeed or fail. The project benefits from an automated continuous integration/continuous development pipeline by utilizing GitHub Actions. This ensures that the main branch is always stable and deployable and that any integration problems are identified early and fixed quickly.

We could not deploy the maven project with Jenins on GCP as requested due to unforeseen and unknown errors thrown by GCP. Though many attempts were made by every group member, GCP would not allow us to set up a Jenkins system with GCP Kubernetes.

Section 6: Group Contributions

Noah Toma: Booking classes/interface, CI/CD help, miscellaneous
Rolf-Jaden Sibal: CI/CD implementation, test cases, maven-publish.yml
Logan Butler: Database, back-end implementation
Nathan Perez: DB pop, Trip class, CI/CD help, pom.xml help, maven-publish.yml help
Dmitri Rios Nadeau: Controller, HTML templates and pom.xml
Everyone: Project Deliverable documents (you're reading one of them now!)