09/12/2020

Booking App

Systems Analysis & Design

Lecturer: Anila Mjeda

GitHub:
https://github.com/Eoghan1232/CS4125_Project1_TeamBased

Submitted by Group 3:
Eoghan Russell 17202124
Pawel Ostach 17214211
Damian Skrzypek 17217679
Darragh Kelly 17235545

Table of Contents

Business Scenario

This project is a transport ticket booking mobile application. The app will be capable of letting its users being able to book and pay for transport tickets. It has the capabilities of letting the user once they have booked tickets to view these tickets and be able to cancel them.

This system works as follows:

The project will be based on a mobile application for the practicality of this app being used on public transport. The app will be an easy to use GUI user interface, providing a user experience that will be simple and easy to navigate for anyone.

The user will download the app and install it on their phone/smart device. They will be able to register and login into the application. A users account will contain details about them, including their name and email address. Once logged in they will be able to search the nearby routes and will be able to select the route that they desire. Once selected the user will have the option to book a ticket on this route, if they have a discount code, they will be able to use here and get money off the ticket. After booking the ticket they will be able to view this ticket and will be able to show it to the ticket inspector if they get asked.

The admin and user will be able to change their account information. If in case the admin or user forgets their password, they will have the option to change password.

Administrators of the app will have access to the same app as users, but we'll have more abilities then that of the user. If needed, Admins will have the ability to add or remove routes and be able to modify existing routes.

Software Lifecycle

We decided on an agile approach for our project as we were most familiar with this software development cycle. However, it's still important to discuss how we came to this decision of choosing agile over the waterfall software cycle.

Waterfall Model

"The Waterfall SDLC model is a sequential software development process in which progress is regarded as flowing increasingly downwards (similar to a waterfall) through a list of phases that must be executed in order to successfully build a computer software". (Bassil 2012)

The major issue we had when discussing the waterfall model was the structure and sequence of steps that must be taken. We can't move onto the next step without the previous step being complete as well as not being able to move back a step if required. It seemed to be a poor choice to make when we all used agile methodologies on placement, and we didn't feel we would succeed in adapting a new software lifecycle in a semester. We also assumed that there would be hiccups and changes need throughout the project and we would not get everything perfect in one iteration.

Agile

The term agile stands for "moving quickly" (Balaji and Murugaiyan 2012). We liked the idea of having deliverables through the semester that we could discuss in our weekly scrum meetings. One of the main advantages of using an agile approach is the "ability to respond to the changing requirements of the project" (Balaji and Murugaiyan 2012). This is a bonus over waterfall as we could make changes at any point throughout the project and incorporate them.

Taking an iterative approach to our work, we could achieve our goals each week in terms of deliverables. Each build is an increment of the last in terms of features, the final build holds all the features required from the start.

With an agile approach, there was no confusion when one part was going to be completed and the next started, having weekly scrum meetings and goals meant there is no guesswork between each team member as there is continuous communication and input from everywhere.

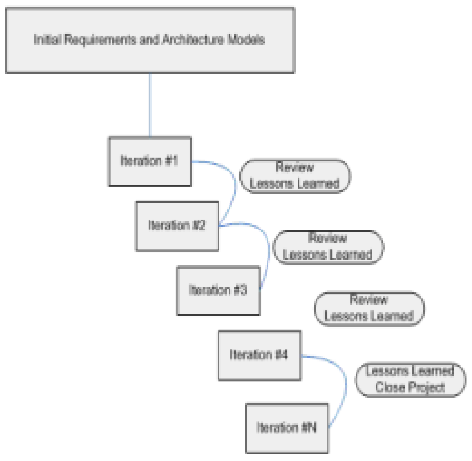


Figure 1: Agile Model Life Cycle (Balaji and Murugaiyan 2012)

Project plan

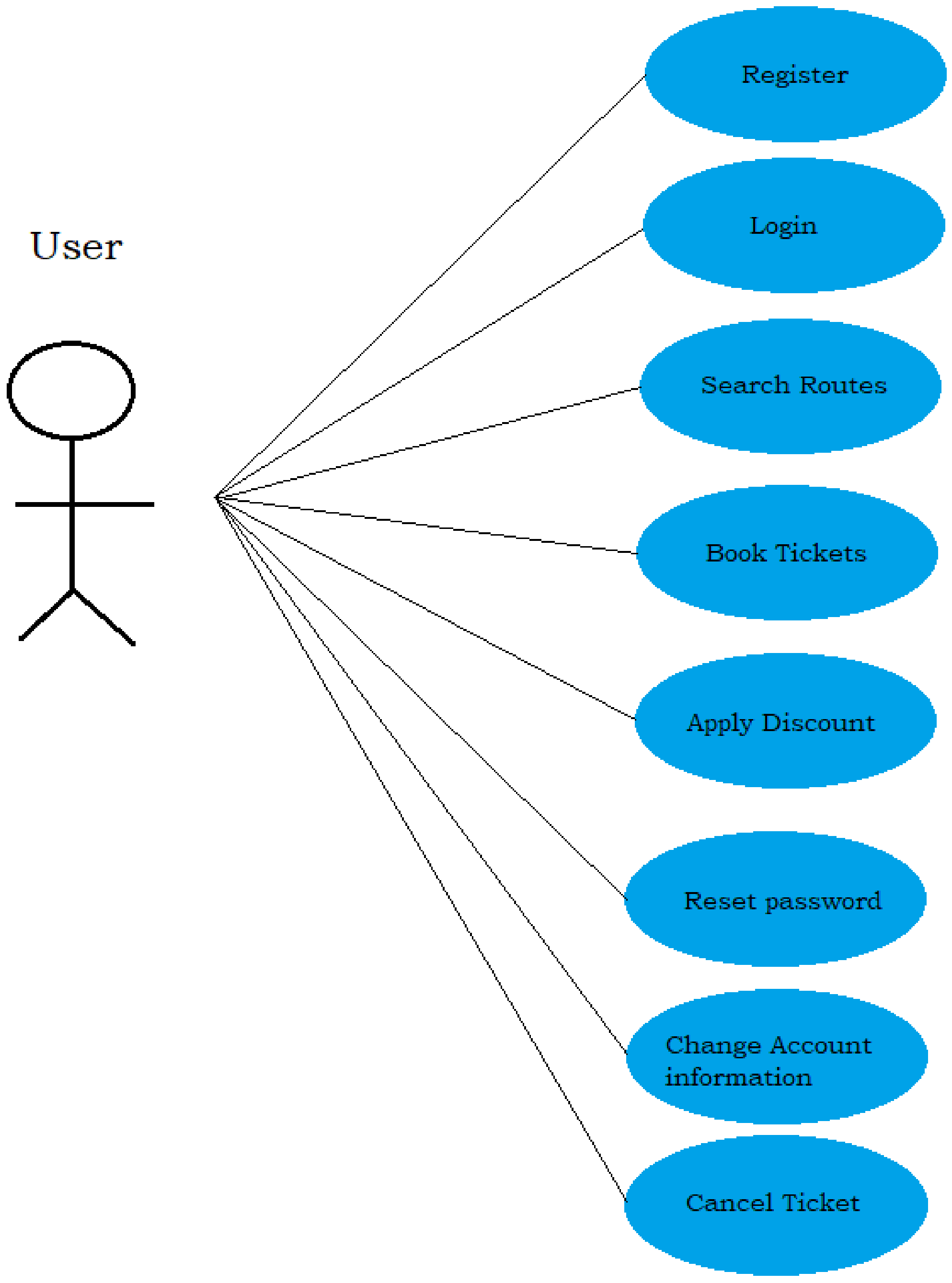
Deliverable	Description	Responsibility	Week
Business Scenario	• Concept	Darragh	4
Software Lifecycle	• Introduction • Agile	Eoghan	4
Project Plan	• Project Plan Table	Pawel	4
Established Roles	• Role Allocation Table	All	3
Requirements	• Use case Diagrams • Structured Use Case Descriptions • Detailed Use Case Descriptions • Non-functional requirements • Quality Attributes • GUI Prototypes	Darragh Darragh Darragh Damian Eoghan Pawel	5or6
System Architecture	• Package Diagram • Decision Details	Damian	6
System Analysis	• Candidate Classes • UML Analysis Diagram • Communication (Sequence) Diagram	Damian	6
Tabular Class Listing	• Code Breakdown Table	Eoghan/Darragh	10
Code	• Code Snippets • Model View/Architecture • Design Patterns • GUI/UI Screenshots and Explanation • Version Control • JUnit Testing • Code Style and Standards • Database Support • Functions is Used (e.g. Lambda)	All	10
Added Value	• GitHub • Database (SQL)	Eoghan/Damian	11
Recovered Architecture and Design Blueprints	• Architectural Diagram • Design Time Class Diagram • State Chart Diagram • Screen Correlation	Pawel	11
Critique	• Architecture • Design Patterns • Language • Framework	Eoghan	11
References	• List of Readings	Eoghan	12

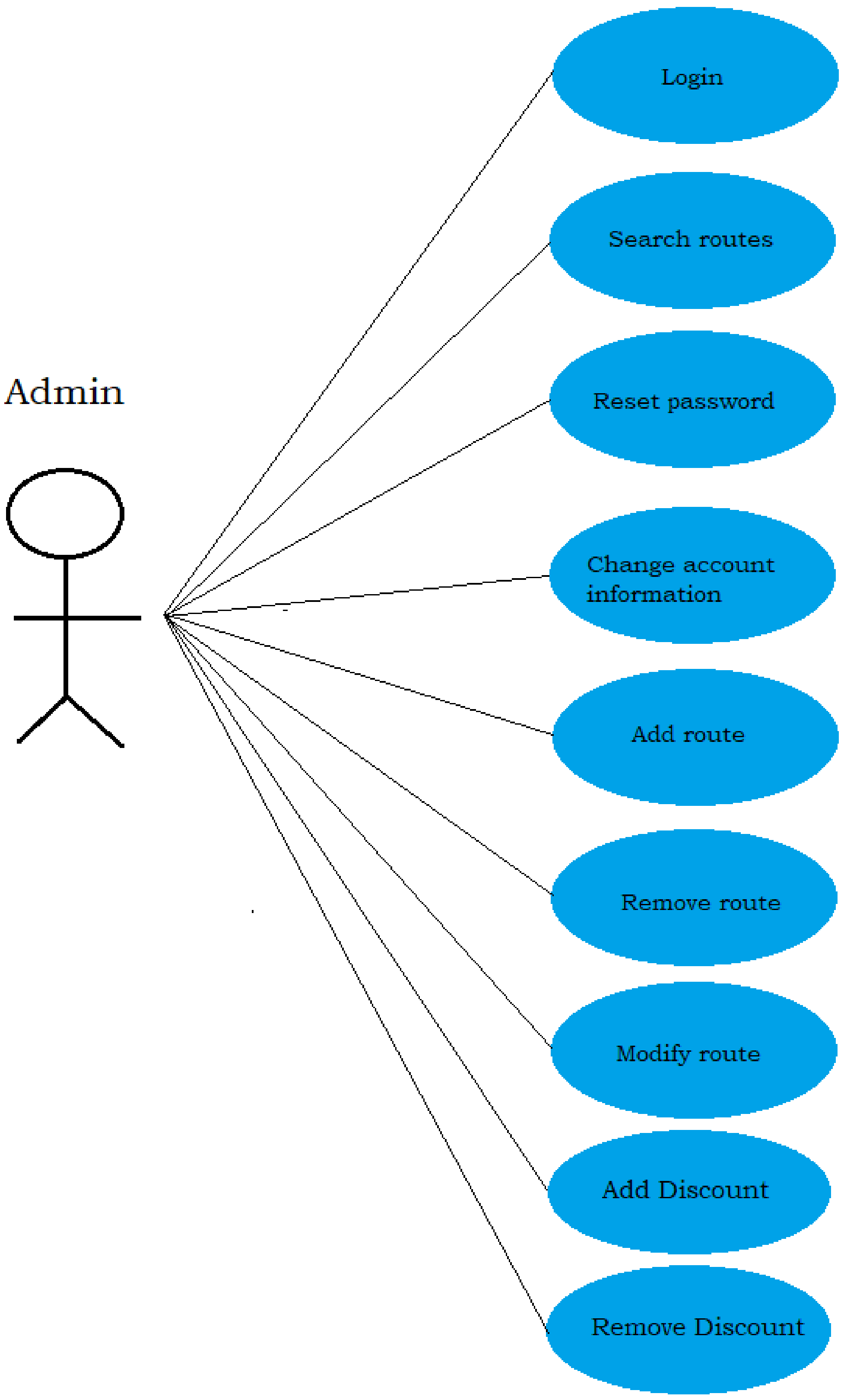
	Role	Description	Designated Team Member
1	Project Manager	Sets up group meetings, gets agreement on the project plan, and tracks progress.	Pawel Ostach
2	Documentation Manager	Responsible for sourcing relevant supporting documentation from each team member and composing it in the report.	Eoghan Russell
3	Business Analyst / Requirements Engineer	Responsible for section 6 - Requirements.	Darragh Kelly
4	Architect	Defines system architecture	Damian Skrzypek
5	Systems Analysts	Creates conceptual class model	Damian Skrzypek
6	Designer	Responsible for recovering design time blueprints from implementation.	Pawel Ostach
7	Technical Lead	Leads the implementation effort	Darragh Kelly
8	Programmers	Each team member to contribute equally to development of the project	ALL
9	Tester	Coding of automated test cases	Eoghan Russell

Requirements

Functional

Use Case Diagrams




Admin
Login
Search routes
Reset password
Change account information
Add route
Remove route
Modify route
Add Discount
Remove Discount

Use Case Descriptions

Use case 1 - Register

Actor Action	System Response
1. User enters credentials and clicks register	2. System ensures username is unique
	3. System creates new user account

Alternative Course:

- Username is already taken
- System will display an error message

Use case 2 - Login

Actor Action	System Response
1. User enters their credentials and selects the login button	2. System checks username and password is correct
	3. User is then logged in

Alternative Course:

- User enters wrong credentials
- System will display an error message

Use case 3 - Search routes

Actor Action	System Response
1. User select the search route option from the menu.	2. System displays the routes that are available
3. User selects the route they want to choose	4. System will display the information for the specific route

Alternative Course:

- System is unable to find a route to the destination that the user wants to travel to.

Use case 4 - Book ticket

Actor Action	System Response
1. User selects the route they want to choose	2. Route is selected
3. User selects the book option under the route	4. System displays a confirmation message

Alternative Course:

- User tries to book a ticket when the capacity of vehicle is full.
- System shows an error message

Use case 5 - Apply discount

Actor Action	System Response
1. User enters in discount during booking	2. System checks if valid discount
	3. System applies the discount to the price of booking

Alternative Course:

- Discount code is no longer viable.
- System shows an error message

Use case 6 - Reset password

Actor Action	System Response
1. User select into ‘Account’ option	2. System brings them to the Account section
3. User selects the ‘Change Password’ button	4. System redirects to a change password screen
5. User enters a new password and selects confirm button	6. Passwords get changed

Alternative Course:

- User tries to change password to the same password as they currently have.
- System shows an error message

Use case 7 - Change account information

Actor Action	System Response
1. User select into 'Account' option	2. System brings them to the Account section
3. User selects the change account information	4. System changes the account information

Alternative Course:

- User does not enter all mandatory information fields and tries to save
- An error message will appear.

Use case 8 - Cancel ticket

Actor Action	System Response
1. User selects cancel ticket option in menu	2. System shows the valid tickets that can be cancelled
3. User confirms the ticket which they want to cancel and selects 'Confirm'	4. System cancels the ticket

Alternative Course:

- User tries to cancel ticket during transit.
- System shows an error message

Use case 9 - Add route

Actor Action	System Response
1. Admin selects the option to add new route	2. New route option screen shows
3. Admin enters in the details about new route and selects add	4. System add the new route as an option

Alternative Course:

- The route already is at max route capacity
- System shows an error message

Use case 10 - Remove route

Actor Action	**System Response**
1. Admin selects the option to remove route	2. Remove route option screen shows
3. Admin confirms that the route is being removed	4. System removes route from the options

Alternative Course:

- Admin tries to remove route that is currently active
- System shows an error message

Use case 11- Modify route

Actor Action	**System Response**
1. Admin selects the option to modify route	2. Modify route screen shows
3. Admin changes information about the route	4. System changes the information about route

Alternative Course:

- Admin tries to modify route that is currently active
- System shows an error message

Use case 12 – Add Discount

Actor Action	**System Response**
1. Admin selects the option to add new Discount	2. Add Discount screen appears
3. Admin add new information about the discount e.g. how much, the code etc.	4. System add the new discount code.

Alternative Course:
The discount code already exists.
System shows an error.

Use case 13 – Remove Discount

Actor Action	**System Response**
1. Admin selects the option to remove Discount	2. Remove Discount screen appears
3. Admin selects the remove option on the discount in which they want to remove	4. System removes the discount.

Alternative Course:
The discount code does not exist.
System shows an error.

Detailed Use Case Descriptions

Use Case	Book Ticket
Goal in Context	User is able to select the route they wish to book and is able to book the ticket for it.
Scope and Level	System
Preconditions	1. User must be logged in 2. The user must have searched the routes and selected one 3. The route should not be at full capacity
Success End Conditions	User is sent a ticket which it can be used on the mode of transport the user selected.
Failed End Conditions	The route is at full capacity and will not take any more bookings
Primary, Secondary, Actors	User, admin
Trigger	User tries to book a ticket for a route.
Descriptions	**Steps:** **1.** User registers and logs into Booking App **2.** User searches routes and selects a route. **3.** User selects a ticket from the route and books this ticket **4.** Ticket is booked
Extensions	The route is fully book and cannot have any more passengers, ticket is not booked
Variations	N/A

RELATED INFORMATION	Book Ticket
Priority:	Very High
Performance	5 minutes
Frequency	200/day
Channel to actors	N/A
OPEN ISSUES	N/A
Due Date	Release 1.0
Superordinate’s	
Subordinates	

Use Case	Cancel Ticket
Goal in Context	User is able to cancel ticket if needed
Scope and Level	System
Preconditions	1. User must be logged in 2. User must have booked a ticket successfully
Success End Conditions	User cancels the ticket successfully
Failed End Conditions	The route is already in progress and tickets cannot be cancelled
Primary, Secondary, Actors	User
Trigger	User tries to cancel ticket
Descriptions	**Steps:** **1.** User registers and logs into Booking App **2.** User searches routes and selects a route. **3.** User selects a ticket from the route and books this ticket **4.** Ticket is booked **5.** User cancels ticket
Extensions	The route is in progress and cancel ticket do not work.
Variations	N/A

RELATED INFORMATION	Cancel Ticket
Priority:	High
Performance	5 minutes
Frequency	10/day
Channel to actors	N/A
OPEN ISSUES	N/A
Due Date	Release 1.0
Superordinate’s	
Subordinates	

Use Case	Modify route
Goal in Context	Admin can change the details of a route
Scope and Level	System
Preconditions	1. Admin must be logged in. 2. Must have Admin type of User 3. Route must already exist
Success End Conditions	Admin modifies route successfully
Failed End Conditions	Admin does not put in the mandatory information in the route and route details do not change
Primary, Secondary, Actors	Admin
Trigger	Admin selects modify route option on route
Descriptions	**Steps:** **1.** Admin must log in with an Admin type account **2.** Route must be created previously and still exist **3.** Admin must select the modify option on the route **4.** Admin changes the route details and saves changes
Extensions	Admin does not fill in the mandatory route details (e.g. Route name) and tries to save. An error message will appear
Variations	N/A

RELATED INFORMATION	Modify
Priority:	High
Performance	5 minutes
Frequency	5/week
Channel to actors	N/A
OPEN ISSUES	N/A
Due Date	Release 1.0
Superordinate’s	
Subordinates	

Non-Functional Requirements

Technical:

- Use Android studio for the frontend
- Must be developed in java and use AWS RDS for data base
- Spring for webservice

Security:

- any private information about the user should be encrypted and user should be aware of what is being saved.
- password should not be visible on the screen and should be encrypted.
- User accounts should not be able to access any resources that they are not authorised to do e.g. resources meant for admins.

Usability:

- UI should be easy to use and should not require much explanation for the user to understand what each button, textbox etc. does and how to interact with them.
- Each action that a user takes should provide feedback to the user, this will ensure the app feels responsive and allows users to quickly understand what the software is doing
- Searching for routes should be easy to do i.e. select from and to stations, date and with one click/tap the user should be able to view the results and select the one they want to buy.

Reliability & Performance:

- Searching for different routes should be quick and should return correct results as this is the main part of the system, responses should not exceed 1s.
- Buying tickets should also be quick but making sure that the correct one and correct quantity is top priority.
- If the system is slow and/or results in incorrect behaviour/result the user is less likely to use it.

Quality Attributes

Security

The use of the Java programming language adds to the security of our application. Upon compilation, the source code written in java gets compiled into bytecode, which is later interpreted by the Java virtual machine. “Bytecode is resistant to tampering by external agents” (*Why Is Java Preferred to Other Languages as a Building Block?*).

We will provide security of our user’s private information through encryption. We used sha256 encoding to encode the password (one-way hashing) and never return the password back. This means that we generate a fingerprint of the input, but there is no way to get back to the original input.

The user’s will not be able to access any unauthorised information, this will be hidden to users unless they are an admin.

Usability

The use of the android platform for our booking application contributes to the usability of the system. Android helps us implement the BYOD (Bring your Own Device) approach with ease, as android is installed on various devices. We will also utilise many android features that offer usability, the first being fragments. These are UI layouts, which can be seen as a modular section of an activity, having its own lifecycle, receiving its own events, and you can add or remove while the activity is running.

We will provide feedback to the user by using the inbuilt functionality of toasts in android. These are small popups that provide simple feedback when required.

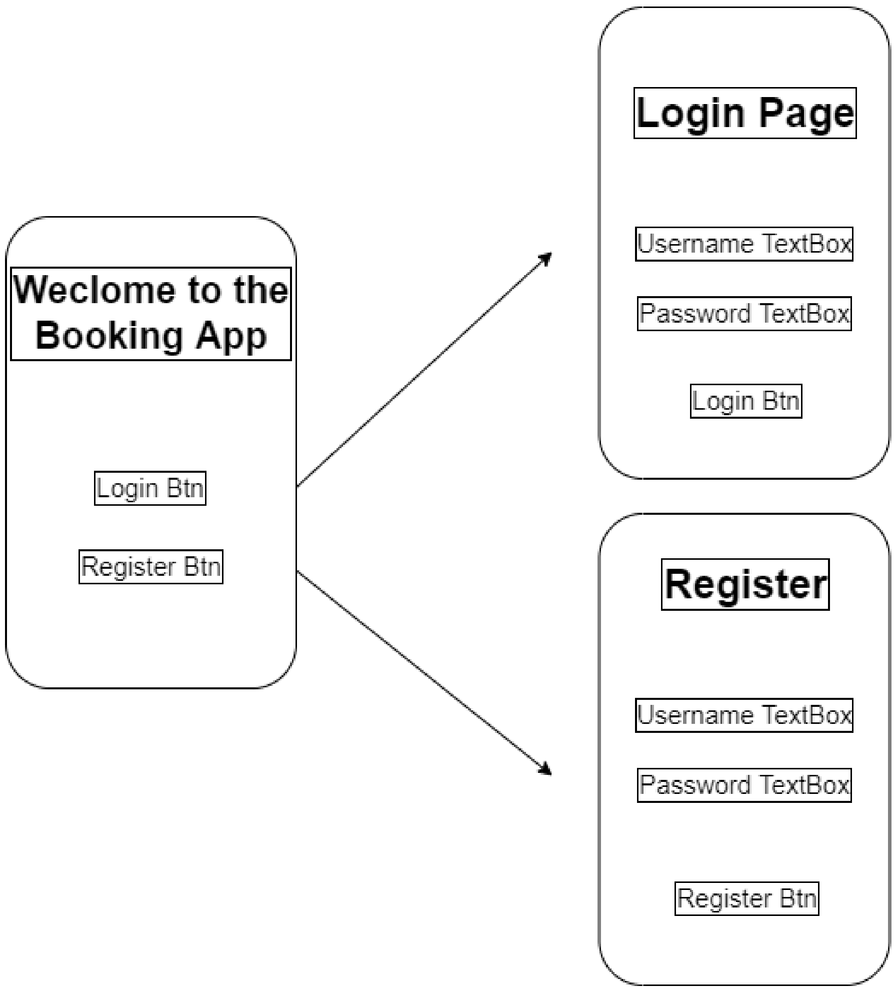
Reliability & Performance

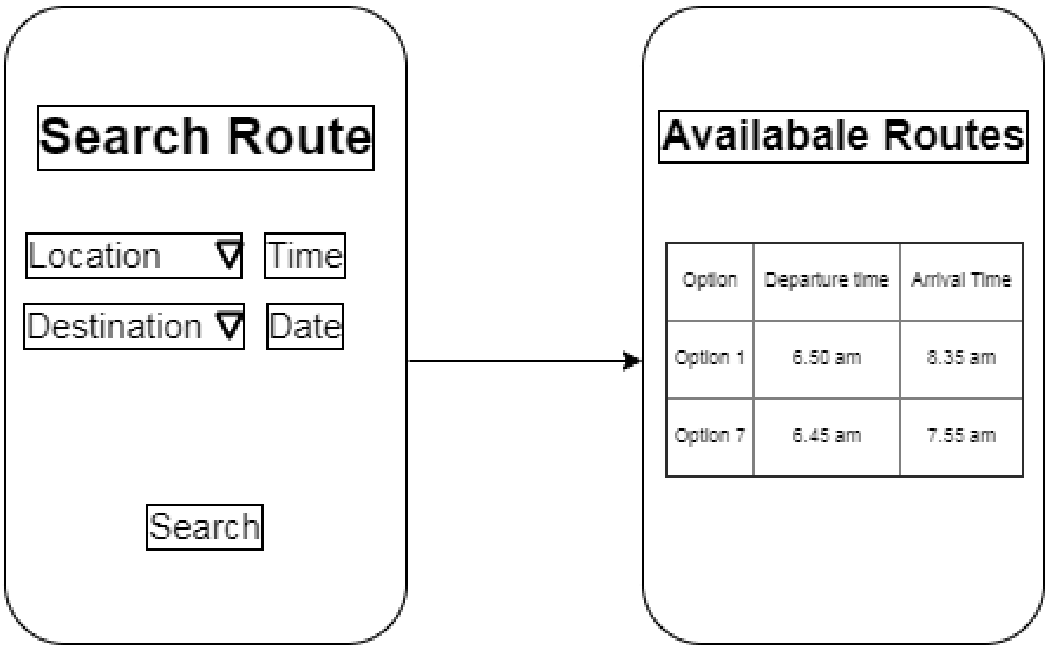
Through the use of android functionality such as fragments, we will see an increase in performance. The lifecycle of an activity is intense, whereas fragments remain in the background until called. Therefore, saving performance when they are not required.

The use of the java programming language adds to the reliability of our application. “Java supports reliable exception handling that can withstand all the major types of erroneous and exception conditions without breaking the system” (*Why Is Java Preferred to Other Languages as a Building Block?*). This means that our application is less likely to crash without giving us an error.

The use of an AWS (Amazon Web Service) database adds to the reliability of our application. The database has automated features that make it reliable, crash recovery being one of those. This means that it’s designed to recover instantaneously and continue to serve our application.

GUI Prototypes





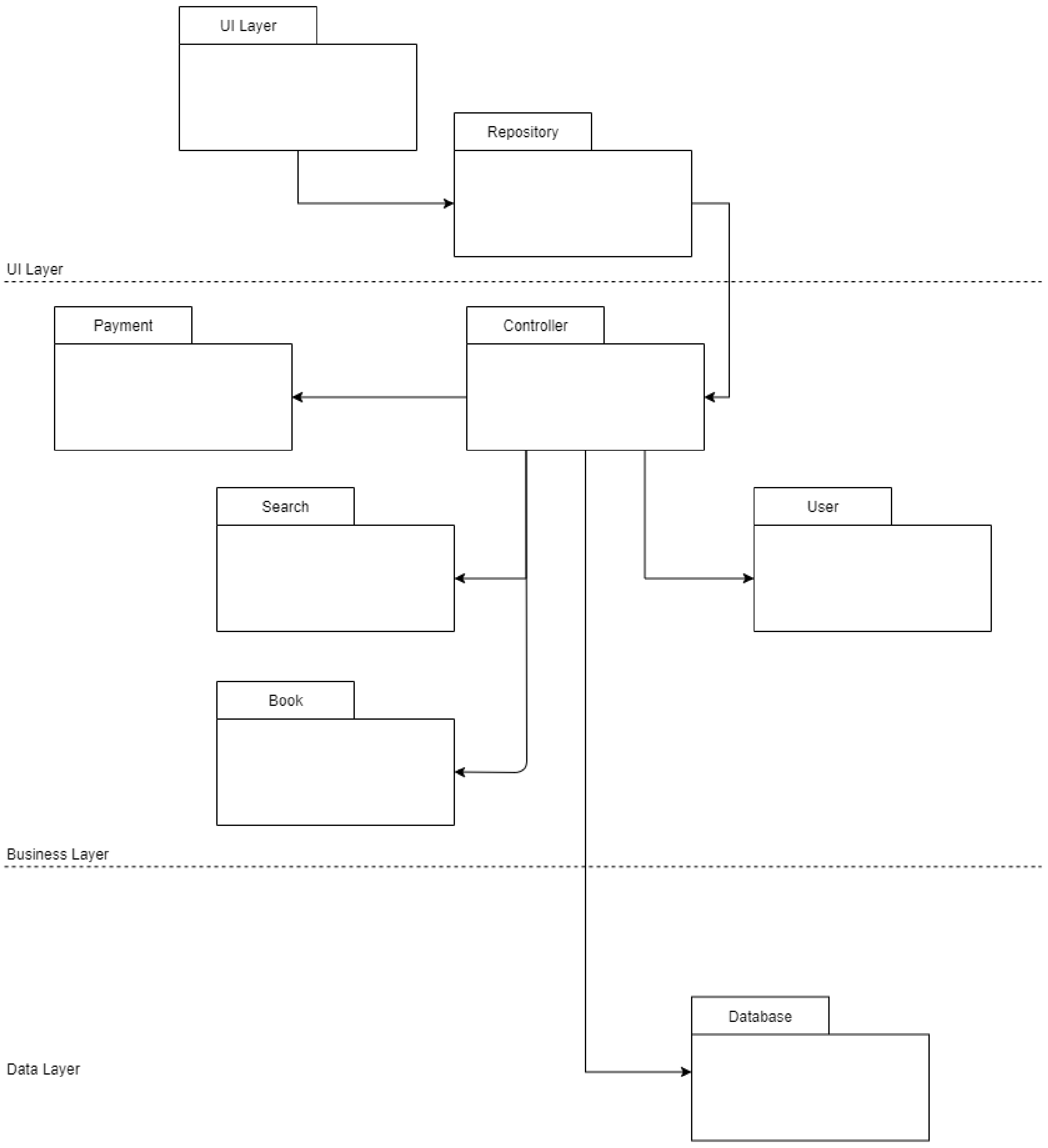
Option	Departure time	Arrival Time
Option 1	6.50 am	8.35 am
Option 7	6.45 am	7.55 am

These GUI Prototypes are a first draft of how we expect the final implementation to be. We will use these as guidelines for the front end UI development phase.

These GUI Prototypes will keep us on track and have been agreed upon as a team.

System Architecture

Package Diagram



Model View Controller (MVC) Discussion

Our system follows model-view-controller (MVC) design pattern by splitting into three layers, a user-interface (UI) layer, a business layer and a data layer.
UI is how the user interacts with our system, then the data from both the UI and the data layer is processed in the business layer.
This approach decouples the different parts of the system providing great flexibility when it comes to changing any of the components, for example if we are required to change the way our data is stored we can easily replace the implementation in data layer to the updated storage solution without breaking any functionality in the business or UI layers.
Another great advantage to using MVC and splitting the system like this is the ease of parallel development as different people can work on different parts without having dependencies on each other, of course for the system to fully work all parts need to be finished.

Android Studio:

Android studio is the main software used in developing android apps, it provides great support for creating apps for different types of phones and versions of android. It has built in layout editor for creating GUI allowing to create nice looking and responsive apps.
The system will be developed using Java as it is one of the main languages when developing apps for Android but also Java is an object-oriented language which will fit very well into the planned design/architecture.

AWS database & hosting the Spring backend:

Amazon web services provides an easy to use and secure environment for developing different array of applications. Thanks to its flexibility we are able to select the appropriate parameters for this project i.e. java, MySQL database and an Linux machine for hosting the Spring webservice backend for our app. Final great advantage to mention is the fact that AWS provides a lot of choices when it comes to pricing, allowing to start of free or with very limited investment and buy more space/computational power as the software’s userbase and scope expands.

JUnit:

For testing we will use JUnit as it is a widely adopted, efficient testing framework becoming a standard around the world when it comes to writing tests for Java programs, and it is supported by most IDE’s including Android studio.

System Analysis

Candidate Classes

Initially to create a class list we have looked at nouns in the functional requirements and use cases this gave us a good starting point for the possible classes required, after more in depth look we have noticed that some classes could be reduced as same functionality can be covered by one class e.g. we don't need separate class for an Admin and can be covered by just having user type in the User class.

Initial identified class list:

User (class)
~~Admin~~ (represented by user type attribute)
~~Customer~~ (represented by user type attribute)
~~Username~~ (attribute)
~~Password~~ (attribute)
~~Register~~(event/operation)
~~Login~~(event/operation)
~~Book~~(event/operation)
Booking(class)
Discount(class)
~~Account information~~ (attributes)
~~Ticket~~ (same as booking)
Route(class)
~~Add~~(event/operation)
~~Remove~~(event/operation)
~~Modify~~(event/operation)
~~Cancel~~(event/operation)
~~Payment~~ (became transaction class)

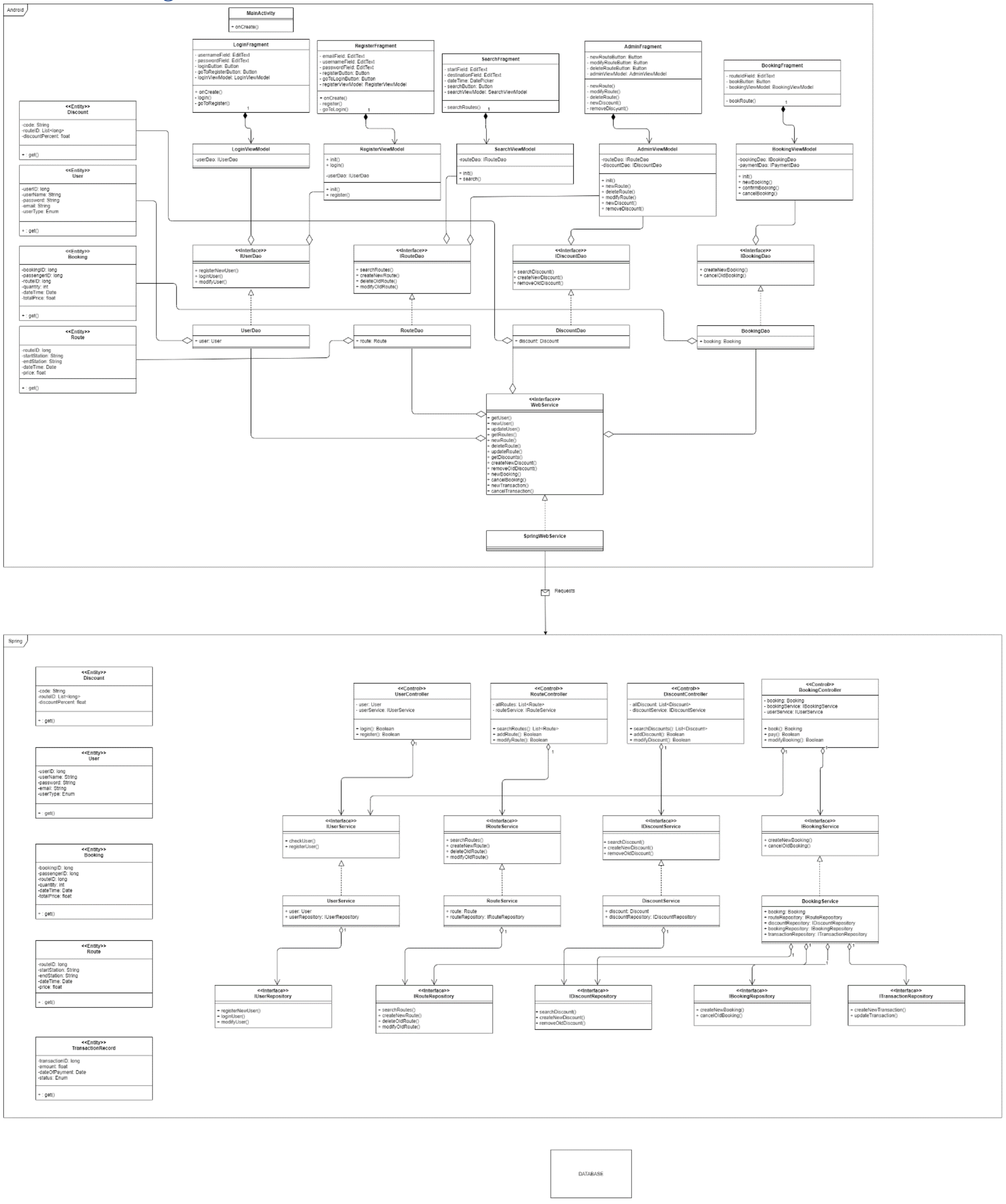
Further Break Down:
Entity classes (User, Booking, Discount, Route, Transaction)

Interface Services for business logic (UserService, RouteService, DiscountService, BookingService)

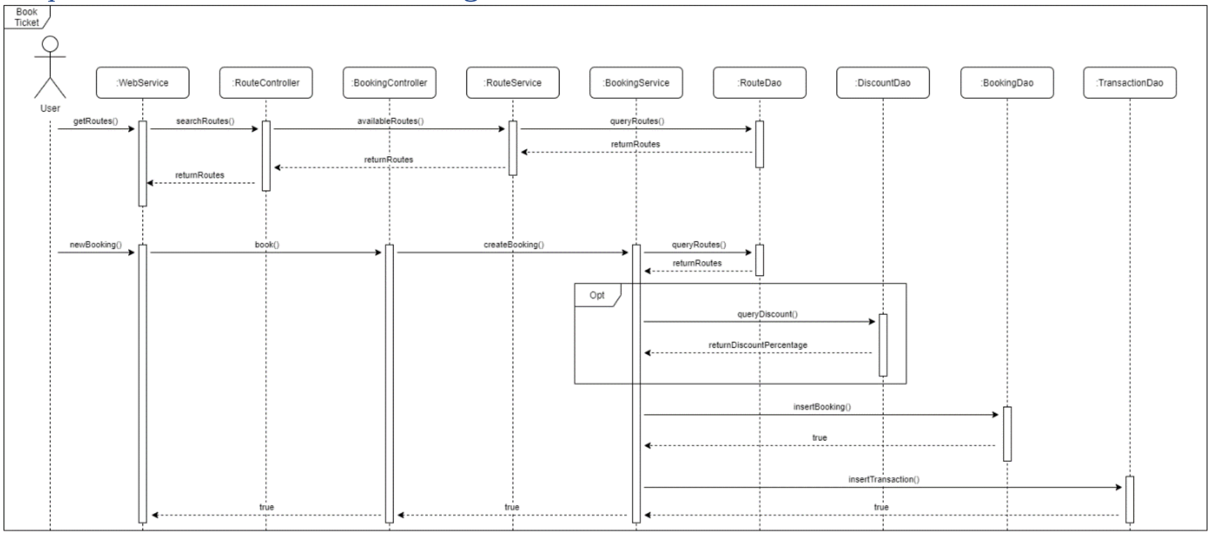
Interface Repository for retrieving data from webservice (UserRepository, RouteRepository, DiscountRepository, BookingRepository, PaymentRepository)

Note: words crossed out are marked as removed from the identified class list.

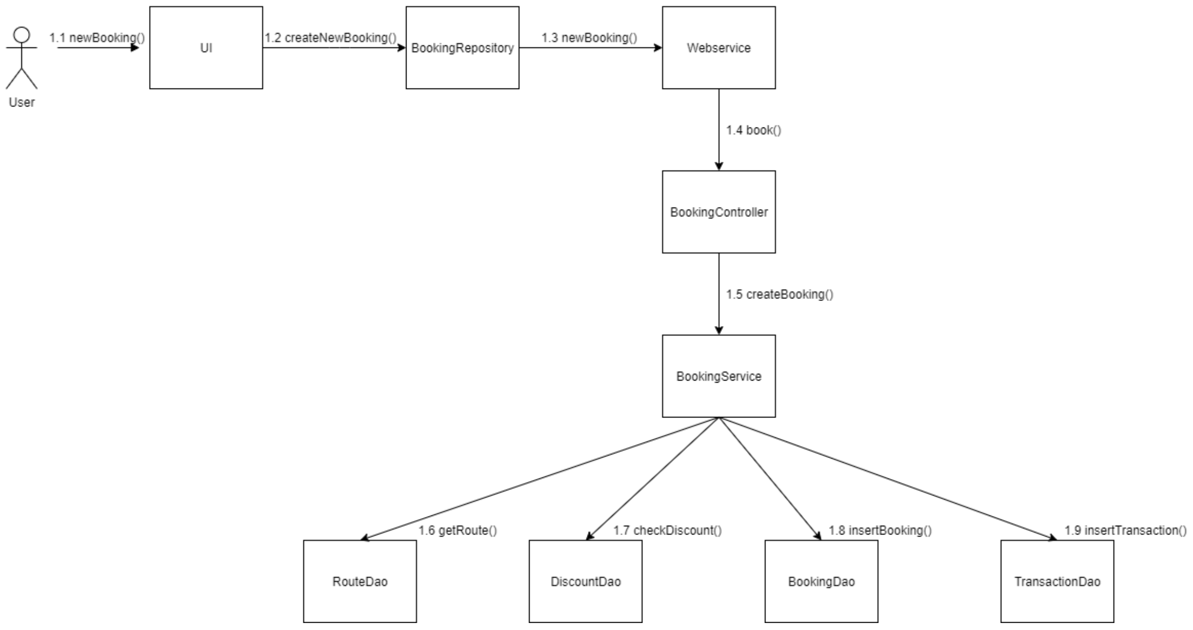
UML Class Diagram



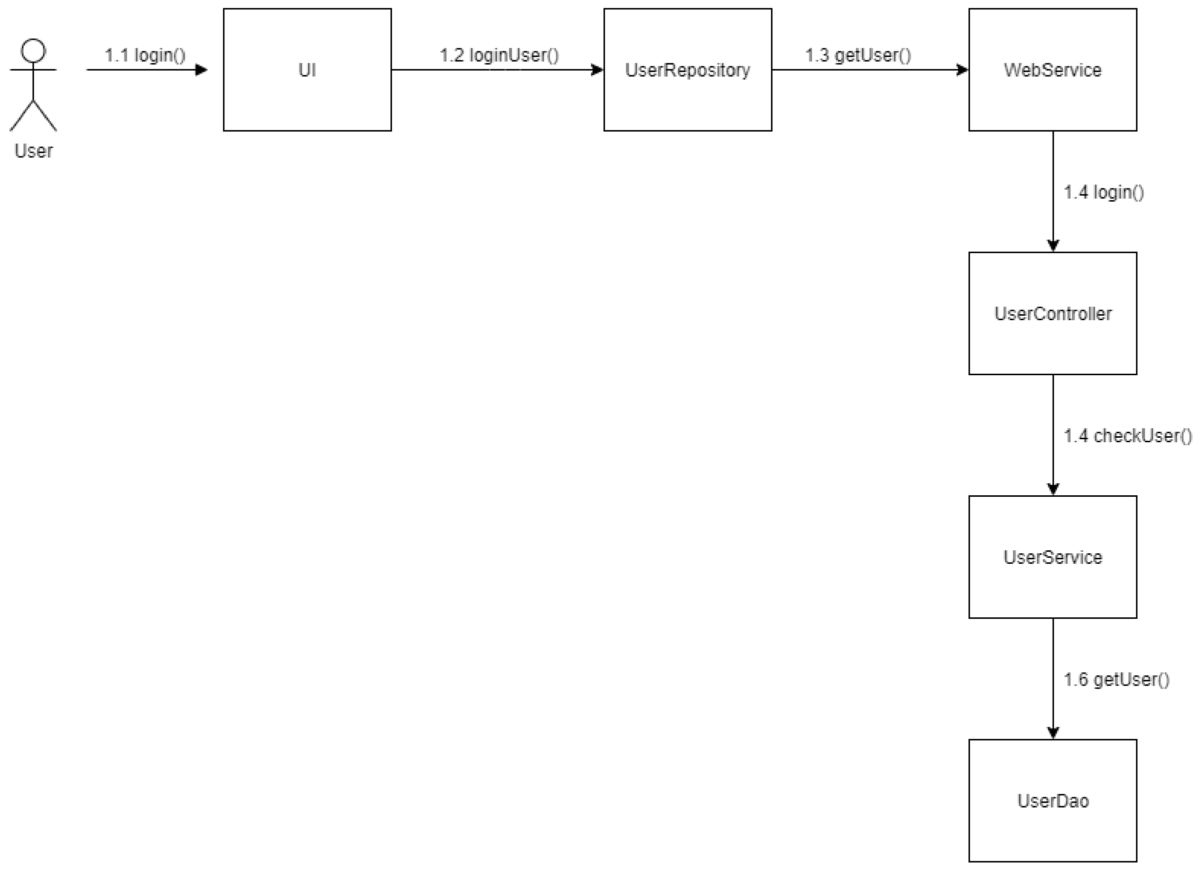
Sequence & Communication Diagrams



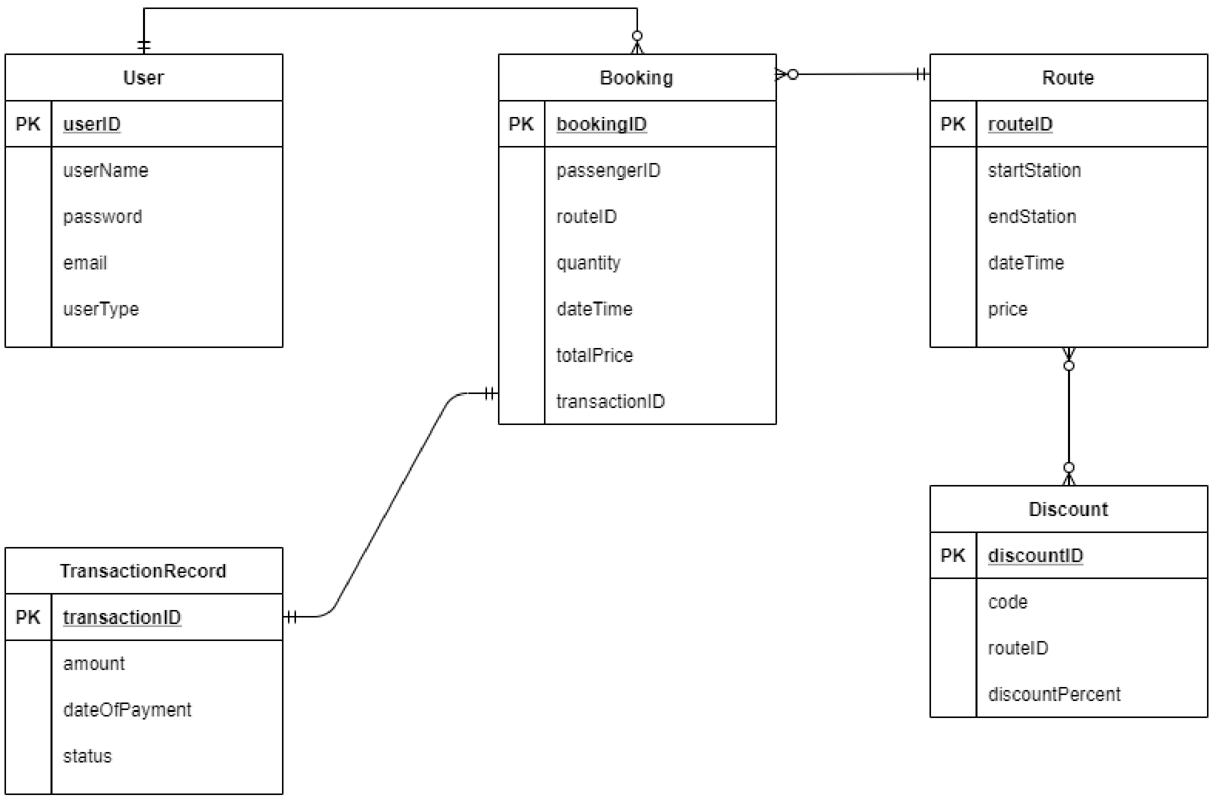
Book Route



Login in


User
1.1 login()
UI
1.2 loginUser()
UserRepository
1.3 getUser()
WebService
1.4 login()
UserController
1.4 checkUser()
UserService
1.6 getUser()
UserDao

Entity Relation Diagram



Tabular Class Listing

Spring Boot

Package	Class	Author	Lines of Code
com.cs4125.bookingapp.controllers	BookingController	Damian/ Eoghan	60
com.cs4125.bookingapp.controllers	DiscountController	Damian/ Eoghan	67
com.cs4125.bookingapp.controllers	RouteController	Damian/ Eoghan	71
com.cs4125.bookingapp.controllers	UserController	Damian/ Eoghan	30
com.cs4125.bookingapp.model.entities	Booking	Damian/ Eoghan	98
com.cs4125.bookingapp.model.entities	Discount	Damian/ Eoghan	67
com.cs4125.bookingapp.model.entities	Route	Damian/ Eoghan	76
com.cs4125.bookingapp.model.entities	TransactionRecord	Damian/ Eoghan	65
com.cs4125.bookingapp.model.entities	User	Damian/ Eoghan	75
com.cs4125.bookingapp.model.repositories	BookingRepository	Damian/ Eoghan	14
com.cs4125.bookingapp.model.repositories	DiscountRepository	Damian/ Eoghan	12
com.cs4125.bookingapp.model.repositories	RouteRepository	Damian/ Eoghan	16
com.cs4125.bookingapp.model.repositories	TransactionRepository	Damian/ Eoghan	12
com.cs4125.bookingapp.model.repositories	UserRepository	Damian/ Eoghan	14
com.cs4125.bookingapp.model	ConcreteRouteFactory	Damian	21
com.cs4125.bookingapp.model	ConcreteUserFactory	Damian	24
com.cs4125.bookingapp.model	RouteFactory	Damian	11
com.cs4125.bookingapp.model	UserFactory	Damian	10
com.cs4125.bookingapp.services	BookingService	Damian	15
com.cs4125.bookingapp.services	BookingServiceImpl	Damian	204
com.cs4125.bookingapp.services	DiscountService	Damian	16
com.cs4125.bookingapp.services	DiscountServiceImpl	Damian	136
com.cs4125.bookingapp.services	EncryptionService	Damian	9
com.cs4125.bookingapp.services	EncryptionServiceImpl	Damian	16
com.cs4125.bookingapp.services	RouteService	Damian	17
com.cs4125.bookingapp.services	RouteServiceImpl	Damian	141
com.cs4125.bookingapp.services	TransactionContext	Damian	57
com.cs4125.bookingapp.services	TransactionRecordCancelledState	Damian	22
com.cs4125.bookingapp.services	TransactionRecordCompletedState	Damian	30
com.cs4125.bookingapp.services	TransactionRecordInitialState	Damian	31
com.cs4125.bookingapp.services	TransactionRecordProgressState	Damian	35
com.cs4125.bookingapp.services	TransactionRecordState	Damian	12

com.cs4125.bookingapp.services	UserService	Damian	10
com.cs4125.bookingapp.services	UserServiceImpl	Damian	58
com.cs4125.bookingapp	BasicController	Eoghan	138
com.cs4125.bookingapp	BookingappApplication	Eoghan	13
com.cs4125.bookingapp.controller	UserTest	Eoghan	53
com.cs4125.bookingapp	BookingappApplicationTests	Eoghan	13

Android Application

Package	Class	Author	Lines of Code
com.cs4125.bookingapp.repositories	BookingRepository	Darragh	14
com.cs4125.bookingapp.repositories	DiscountRepositiory	Darragh/Damian	13
com.cs4125.bookingapp.repositories	PaymentRepository	N/A	5
com.cs4125.bookingapp.repositories	RouteRepository	Darragh	14
com.cs4125.bookingapp.repositories	UserRepository	Darragh	13
com.cs4125.bookingapp.repositories	BookingRepositoryImpl	Darragh	163
com.cs4125.bookingapp.repositories	DiscountRepositoryImpl	Darragh/Damian	193
com.cs4125.bookingapp.repositories	PaymentRepositoryImpl	Darragh	7
com.cs4125.bookingapp.repositories	RouteRepositoryImpl	Darragh	188
com.cs4125.bookingapp.repositories	UserRepositoryImpl	Darragh	79
com.cs4125.bookingapp.repositories	ResultCallback	Damian	8
com.cs4125.bookingapp.ui.main	AdminFragment	Pawel	40
com.cs4125.bookingapp.ui.main	BookingFragment	Pawel	115
com.cs4125.bookingapp.ui.main	BookingResultFragment	Pawel	122
com.cs4125.bookingapp.ui.main	DatePickerFragment	Pawel	33
com.cs4125.bookingapp.ui.main	LoginFragment	Pawel	100
com.cs4125.bookingapp.ui.main	MainFragment	Pawel	71
com.cs4125.bookingapp.ui.main	RegisterFragment	Pawel	104
com.cs4125.bookingapp.ui.main	SearchFragment	Pawel	177
com.cs4125.bookingapp.ui.main	SearchResultFragment	Pawel	90
com.cs4125.bookingapp.ui.main	TimePickerFragment	Pawel	37
com.cs4125.bookingapp.ui.main	RouteAdapter	Pawel	82
com.cs4125.bookingapp.ui.main	Utilities	Pawel	12
com.cs4125.bookingapp.ui.main	SearchViewModel	Darragh	82
com.cs4125.bookingapp.ui.main	AdminViewModel	Darragh	226
com.cs4125.bookingapp.ui.main	MainViewModel	Darragh	8
com.cs4125.bookingapp.ui.main	RegisterViewModel	Darragh	42
com.cs4125.bookingapp.ui.main	BookingViewModel	Darragh	103
com.cs4125.bookingapp.ui.main	LoginViewModel	Darragh	42
com.cs4125.bookingapp.web	SpringRetrofitService	Damian	86
com.cs4125.bookingapp.web	RetrofitClientInstance	Damian	43
com.cs4125.bookingapp.entities	Booking	Damian	123
com.cs4125.bookingapp.entities	Discount	Damian	87
com.cs4125.bookingapp.entities	Route	Damian	106
com.cs4125.bookingapp.entities	TransactionRecord	Damian	67
com.cs4125.bookingapp.entities	TramsactionStatus	Damian	7
com.cs4125.bookingapp.entities	User	Damian	107
com.cs4125.bookingapp.entities	UserType	Damian	7

Total Code Developed

~4585 lines of code written. This excludes auto generated files, e.g. android UI xml files.

Team Member Contribution

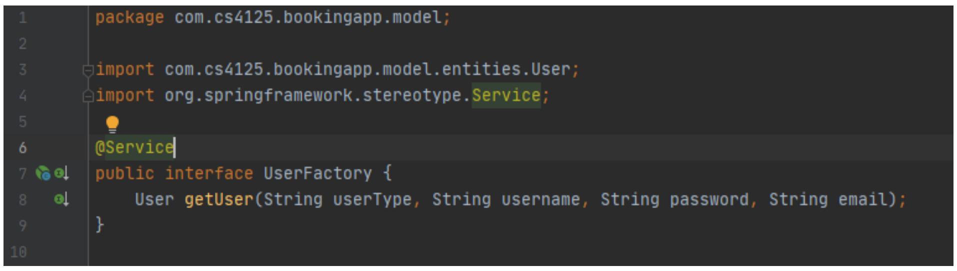
Team Member	Lines Contributed
Eoghan Russell	10,311
Damian Skrzypek	5,414
Pawel Ostach	1,218
Darragh Kelly	1,169

Note: Total Lines contributed taken from GitHub. Also adjusted due to .idea files being committed by mistake.

Code

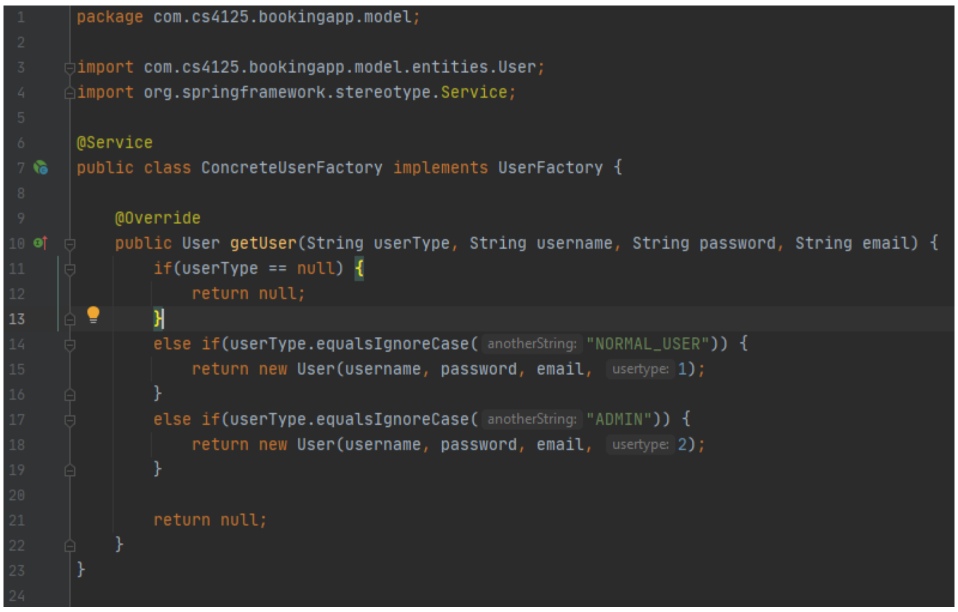
Factory Pattern

UserFactory.java



```java
package com.cs4125.bookingapp.model;

import com.cs4125.bookingapp.model.entities.User;
import org.springframework.stereotype.Service;

@Service
public interface UserFactory {
    User getUser(String userType, String username, String password, String email);
}
```

ConcreteUserFactory.java



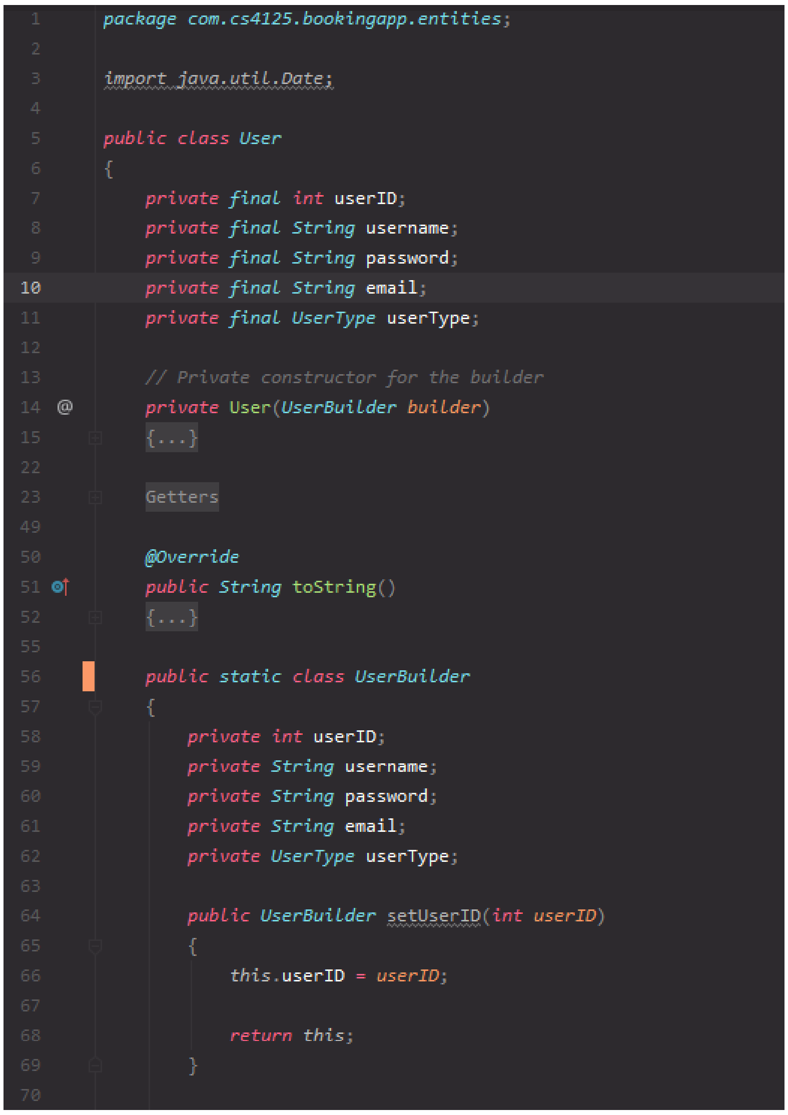
```java
package com.cs4125.bookingapp.model;

import com.cs4125.bookingapp.model.entities.User;
import org.springframework.stereotype.Service;

@Service
public class ConcreteUserFactory implements UserFactory {

    @Override
    public User getUser(String userType, String username, String password, String email) {
        if(userType == null) {
            return null;
        }
        else if(userType.equalsIgnoreCase( anotherString: "NORMAL_USER")) {
            return new User(username, password, email,  usertype: 1);
        }
        else if(userType.equalsIgnoreCase( anotherString: "ADMIN")) {
            return new User(username, password, email,  usertype: 2);
        }

        return null;
    }
}
```

We have used the factory design pattern for creating users and routes, this provides us with an interface for creating an object and let the subclasses decide which class to instantiate without specifying the concrete class in the code. This is very useful as the class doesn't need to be aware what subclasses are required to be created and factory can handle this for it. This also allows us to expand possible user types and route types in the future by providing their instantiations in the factory classes with very minor (or none) changes needed to the classes responsible for the business logic.

Builder Pattern

We have used the builder design pattern for entities on the android side of things, this allows us to create different representations of a complex object and control the steps of construction, if we ever needed to expand any of the entities it would be very easy to expand the entity and builder class without breaking old code as the build would still work.

User.java

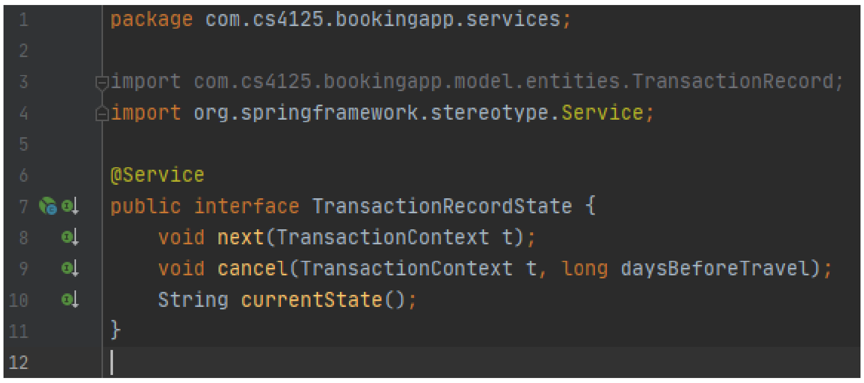


```java
package com.cs4125.bookingapp.entities;

import java.util.Date;

public class User
{
    private final int userID;
    private final String username;
    private final String password;
    private final String email;
    private final UserType userType;

    // Private constructor for the builder
    private User(UserBuilder builder)
    {...}

    Getters

    @Override
    public String toString()
    {...}

    public static class UserBuilder
    {
        private int userID;
        private String username;
        private String password;
        private String email;
        private UserType userType;

        public UserBuilder setUserID(int userID)
        {
            this.userID = userID;

            return this;
        }

```

State Pattern

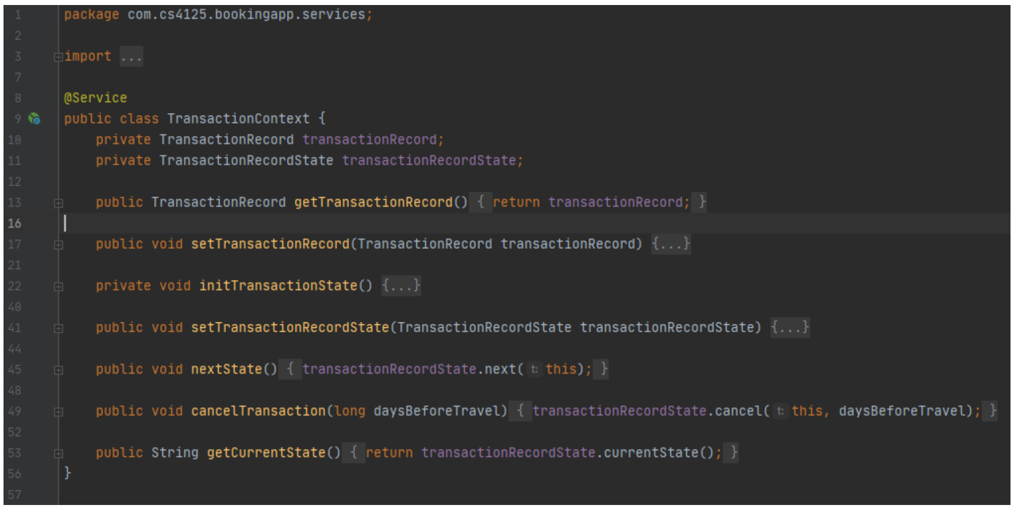
We have used the state design pattern for handling different stages of a transaction, this allows the *TransactionRecord* object to alter its behaviour when it's state changes. We have achieved it by wrapping the *TransactionRecord* object in a *TransactionContext* class, this context class is then able to handle moving to the next state or cancelling internally depending on what state it currently is in.

TransactionRecordState.java



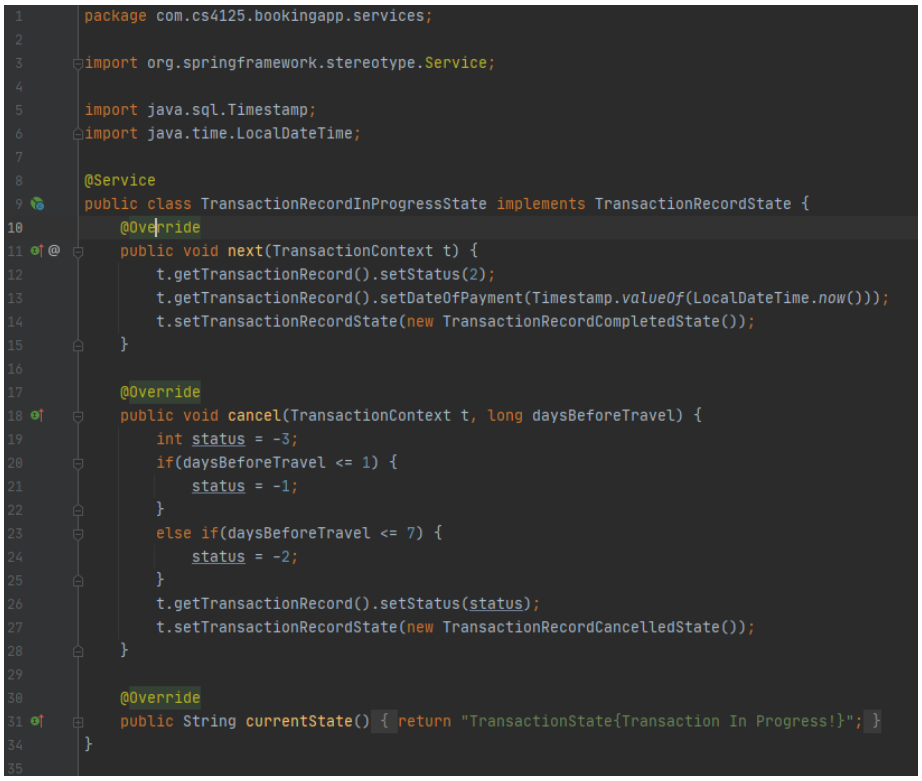
```java
package com.cs4125.bookingapp.services;

import com.cs4125.bookingapp.model.entities.TransactionRecord;
import org.springframework.stereotype.Service;

@Service
public interface TransactionRecordState {
    void next(TransactionContext t);
    void cancel(TransactionContext t, long daysBeforeTravel);
    String currentState();
}

```

TransactionContext.java



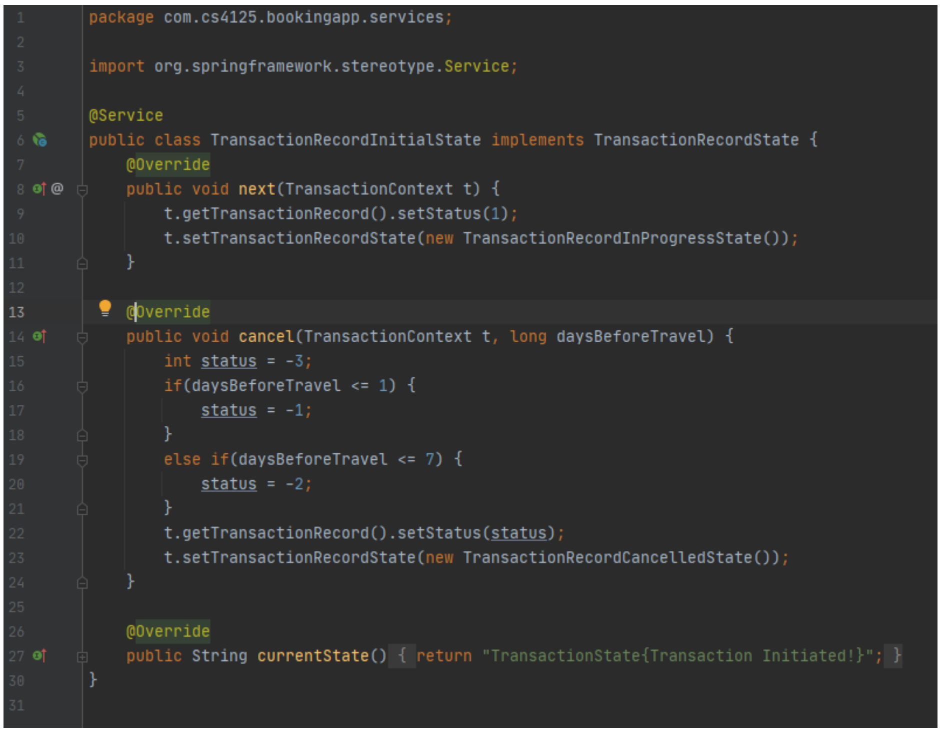
```java
package com.cs4125.bookingapp.services;

import ...

@Service
public class TransactionContext {
    private TransactionRecord transactionRecord;
    private TransactionRecordState transactionRecordState;

    public TransactionRecord getTransactionRecord() { return transactionRecord; }

    public void setTransactionRecord(TransactionRecord transactionRecord) {...}

    private void initTransactionState() {...}

    public void setTransactionRecordState(TransactionRecordState transactionRecordState) {...}

    public void nextState() { transactionRecordState.next( t: this); }

    public void cancelTransaction(long daysBeforeTravel) { transactionRecordState.cancel( t: this, daysBeforeTravel); }

    public String getCurrentState() { return transactionRecordState.currentState(); }
}

```

TransactionRecordInProgressState.java



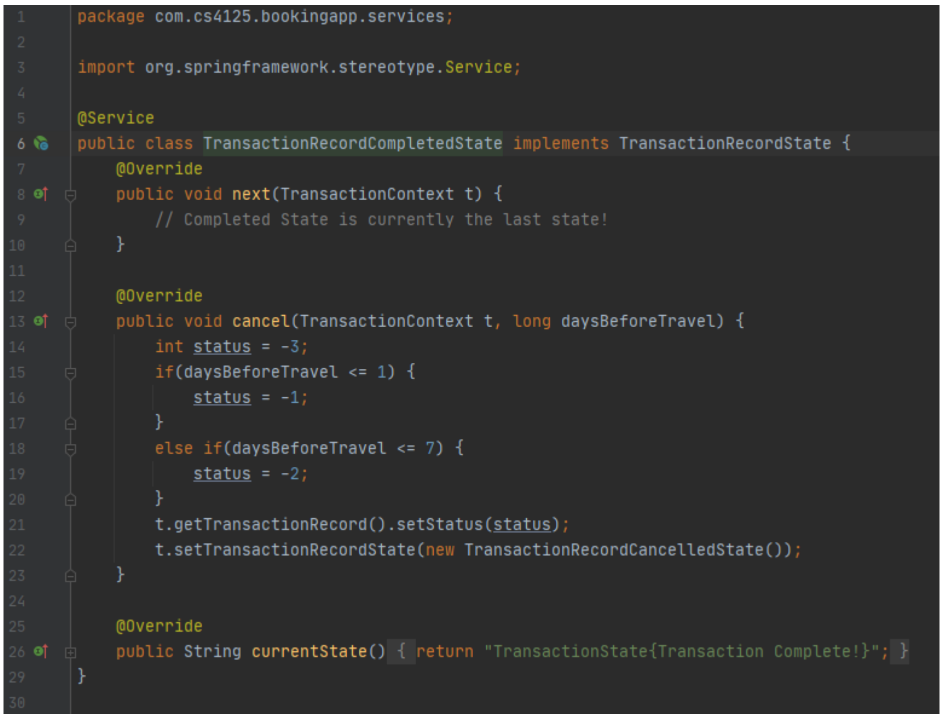
```java
package com.cs4125.bookingapp.services;

import org.springframework.stereotype.Service;

import java.sql.Timestamp;
import java.time.LocalDateTime;

@Service
public class TransactionRecordInProgressState implements TransactionRecordState {
    @Override
    public void next(TransactionContext t) {
        t.getTransactionRecord().setStatus(2);
        t.getTransactionRecord().setDateOfPayment(Timestamp.valueOf(LocalDateTime.now()));
        t.setTransactionRecordState(new TransactionRecordCompletedState());
    }

    @Override
    public void cancel(TransactionContext t, long daysBeforeTravel) {
        int status = -3;
        if(daysBeforeTravel <= 1) {
            status = -1;
        }
        else if(daysBeforeTravel <= 7) {
            status = -2;
        }
        t.getTransactionRecord().setStatus(status);
        t.setTransactionRecordState(new TransactionRecordCancelledState());
    }

    @Override
    public String currentState() { return "TransactionState{Transaction In Progress!}"; }
}
```

TransactionRecordInitialState.java



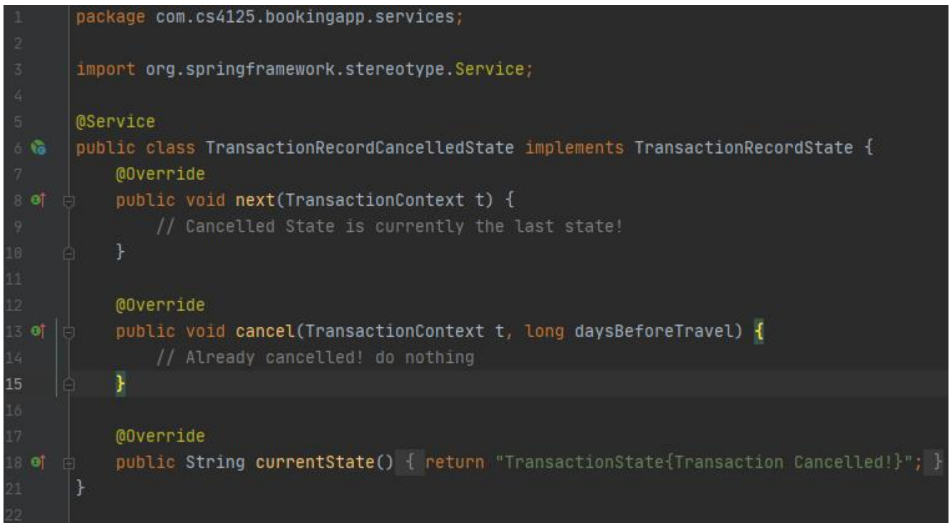
```java
package com.cs4125.bookingapp.services;

import org.springframework.stereotype.Service;

@Service
public class TransactionRecordInitialState implements TransactionRecordState {
    @Override
    public void next(TransactionContext t) {
        t.getTransactionRecord().setStatus(1);
        t.setTransactionRecordState(new TransactionRecordInProgressState());
    }

    @Override
    public void cancel(TransactionContext t, long daysBeforeTravel) {
        int status = -3;
        if(daysBeforeTravel <= 1) {
            status = -1;
        }
        else if(daysBeforeTravel <= 7) {
            status = -2;
        }
        t.getTransactionRecord().setStatus(status);
        t.setTransactionRecordState(new TransactionRecordCancelledState());
    }

    @Override
    public String currentState() { return "TransactionState{Transaction Initiated!}"; }
}
```

TransactionRecordCompleteState.java



```java
package com.cs4125.bookingapp.services;

import org.springframework.stereotype.Service;

@Service
public class TransactionRecordCompletedState implements TransactionRecordState {
    @Override
    public void next(TransactionContext t) {
        // Completed State is currently the last state!
    }

    @Override
    public void cancel(TransactionContext t, long daysBeforeTravel) {
        int status = -3;
        if(daysBeforeTravel <= 1) {
            status = -1;
        }
        else if(daysBeforeTravel <= 7) {
            status = -2;
        }
        t.getTransactionRecord().setStatus(status);
        t.setTransactionRecordState(new TransactionRecordCancelledState());
    }

    @Override
    public String currentState() { return "TransactionState{Transaction Complete!}"; }
}
```

TransactionRecordCancelledState.java

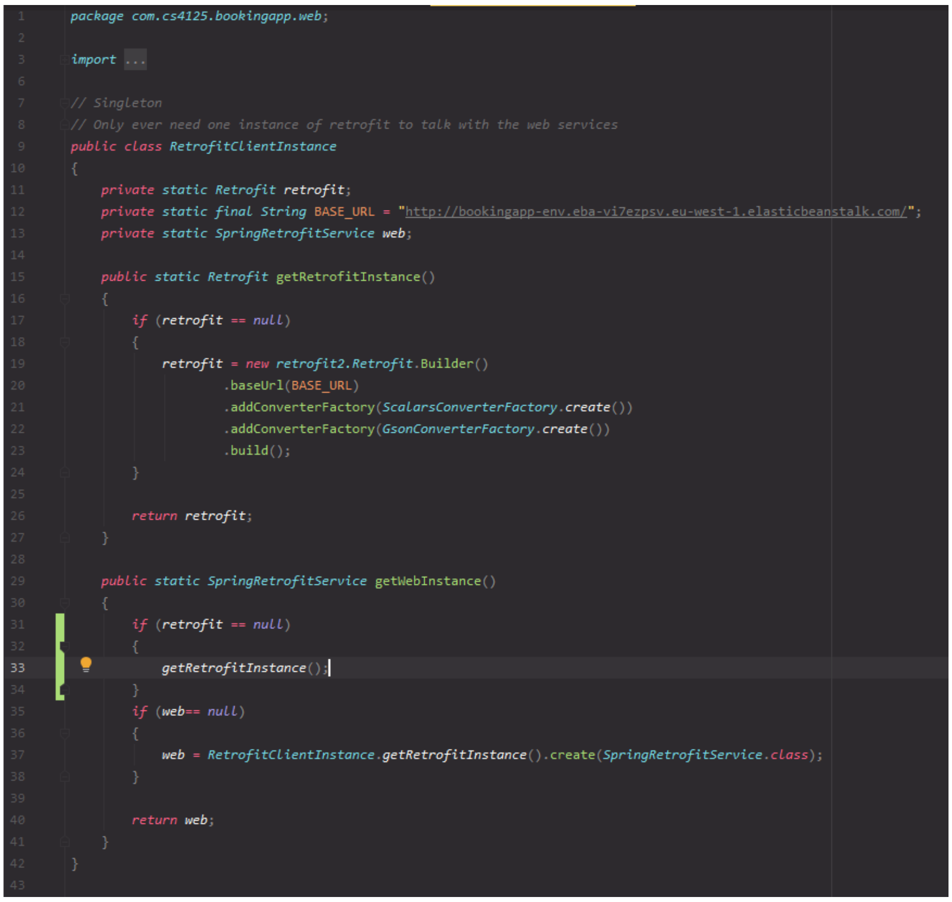


```java
package com.cs4125.bookingapp.services;

import org.springframework.stereotype.Service;

@Service
public class TransactionRecordCancelledState implements TransactionRecordState {
    @Override
    public void next(TransactionContext t) {
        // Cancelled State is currently the last state!
    }

    @Override
    public void cancel(TransactionContext t, long daysBeforeTravel) {
        // Already cancelled! do nothing
    }

    @Override
    public String currentState() { return "TransactionState{Transaction Cancelled!}"; }
}
```

Singleton Pattern

We have used the Singleton design pattern for our retrofit2 class, this was to ensure we only ever have instance that is globally accessible. Main reason for this was that the retrofit2 class is responsible for the connection to the Spring API and as such there should ever only be one instance of this class which represents this connectivity. The repository objects are then able to use this instance to fire requests.

RetrofitClientInstance.java



```java
package com.cs4125.bookingapp.web;

import ...

// Singleton
// Only ever need one instance of retrofit to talk with the web services
public class RetrofitClientInstance
{
    private static Retrofit retrofit;
    private static final String BASE_URL = "http://bookingapp-env.eba-vi7ezpsv.eu-west-1.elasticbeanstalk.com/";
    private static SpringRetrofitService web;

    public static Retrofit getRetrofitInstance()
    {
        if (retrofit == null)
        {
            retrofit = new retrofit2.Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .addConverterFactory(ScalarsConverterFactory.create())
                    .addConverterFactory(GsonConverterFactory.create())
                    .build();
        }

        return retrofit;
    }

    public static SpringRetrofitService getWebInstance()
    {
        if (retrofit == null)
        {
            getRetrofitInstance();
        }
        if (web== null)
        {
            web = RetrofitClientInstance.getRetrofitInstance().create(SpringRetrofitService.class);
        }

        return web;
    }
}
```

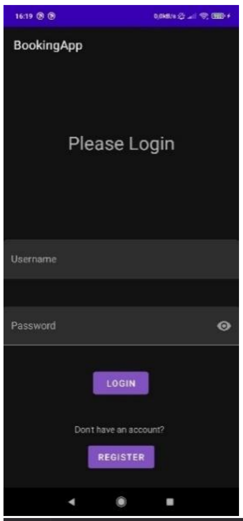
Android Studio

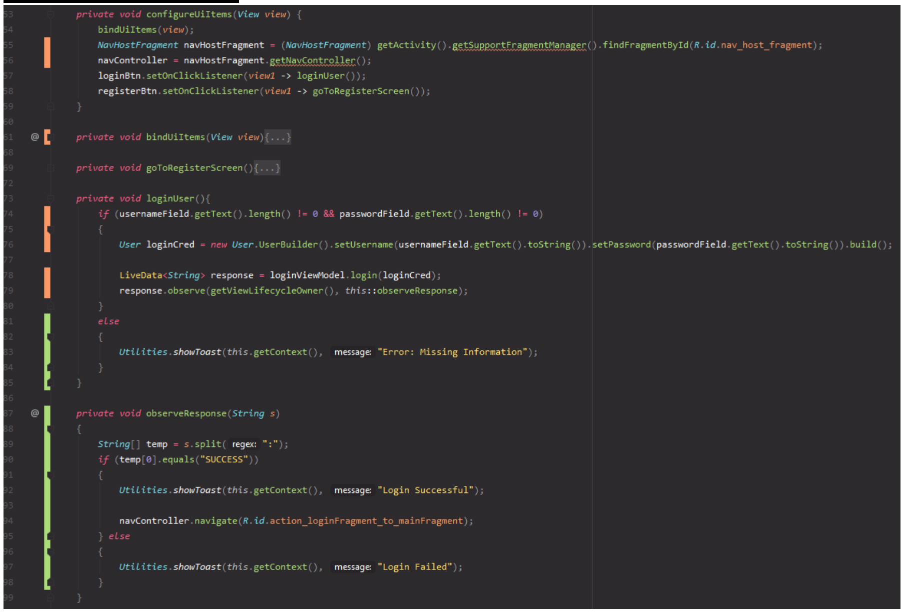
The reason we chose android studio was due to the familiarity of the software on the team and each member had adequate experience in using android studio. From a business point of view, the android market is a huge market. According to Brandom (2019), as of 2019 there were approx. 2.5 billion active android devices creating a robust market for mobile applications.

We can also separate the Ui fragments from the data with the use of the ViewModels, this means multiple fragments can use the same ViewModel to get the required data.

MVC

We have implemented an app in Android Studio as the View part of Software, user sees the different screens in the app.

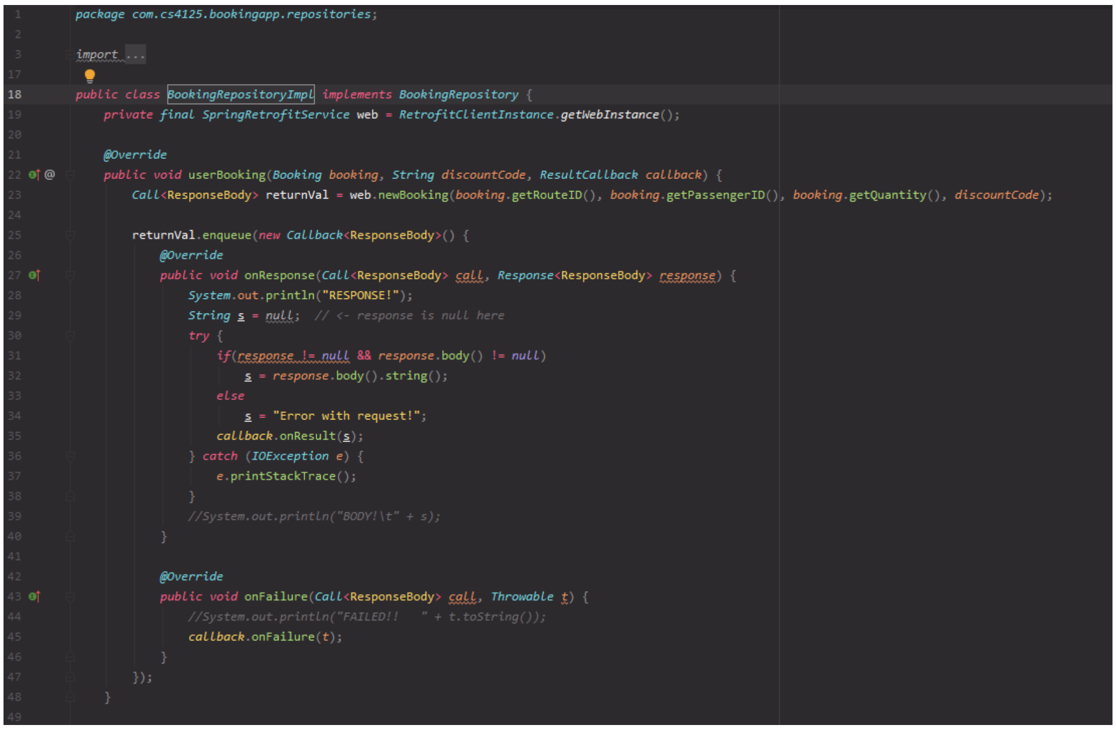




```java
private void configureUiItems(View view) {
    bindUiItems(view);
    NavHostFragment navHostFragment = (NavHostFragment) getActivity().getSupportFragmentManager().findFragmentById(R.id.nav_host_fragment);
    navController = navHostFragment.getNavController();
    loginBtn.setOnClickListener(view1 -> loginUser());
    registerBtn.setOnClickListener(view1 -> goToRegisterScreen());
}

private void bindUiItems(View view){...}

private void goToRegisterScreen(){...}

private void loginUser(){
    if (usernameField.getText().length() != 0 && passwordField.getText().length() != 0)
    {
        User loginCred = new User.UserBuilder().setUsername(usernameField.getText().toString()).setPassword(passwordField.getText().toString()).build();

        LiveData<String> response = loginViewModel.login(loginCred);
        response.observe(getViewLifecycleOwner(), this::observeResponse);
    }
    else
    {
        Utilities.showToast(this.getContext(),  message: "Error: Missing Information");
    }
}

private void observeResponse(String s)
{
    String[] temp = s.split( regex: ":");
    if (temp[0].equals("SUCCESS"))
    {
        Utilities.showToast(this.getContext(),  message: "Login Successful");

        navController.navigate(R.id.action_loginFragment_to_mainFragment);
    } else
    {
        Utilities.showToast(this.getContext(),  message: "Login Failed");
    }
}
```

User can then interact with the system through the GUI by inputting information into text fields and pushing buttons, this is the Controller. The ViewModels are responsible for requesting data from the Spring Boot webservice, it achieves this by calling Repository classes(abstraction layer of retrieving data), by having this abstraction layer it does allow us to possibly expand our app to different databases but local and online, for now it only connects to the Spring API through REST implemented using retrofit2.

BookingRepositoryImpl.java



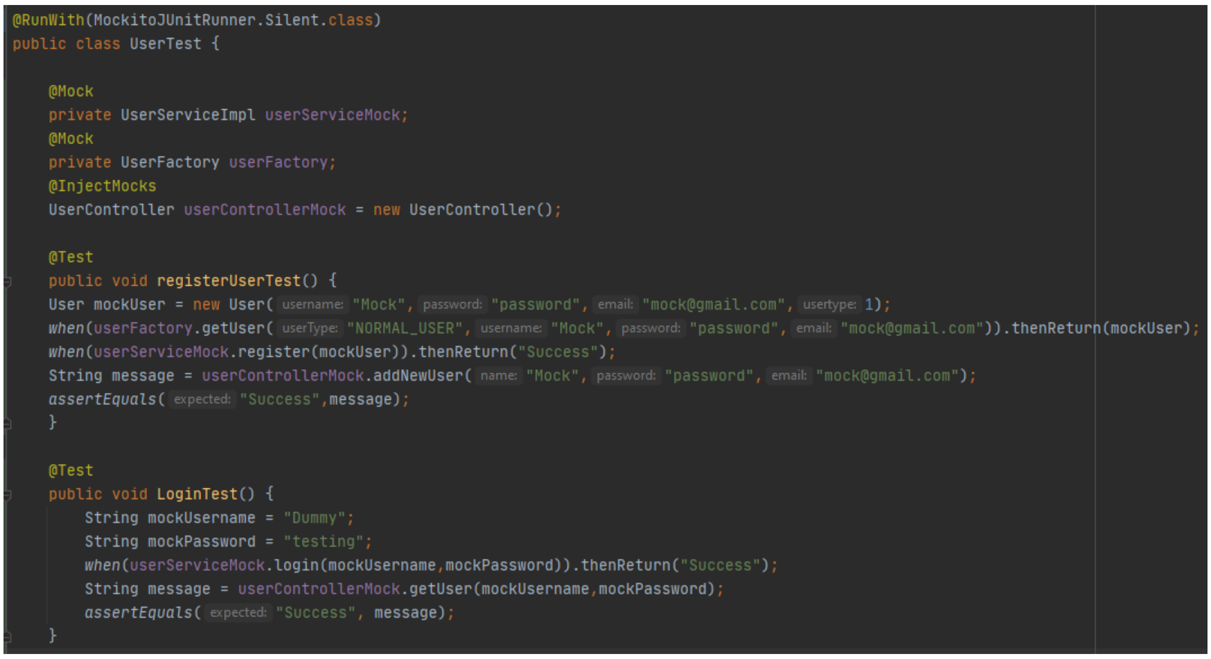
```java
package com.cs4125.bookingapp.repositories;

import ...

public class BookingRepositoryImpl implements BookingRepository {
    private final SpringRetrofitService web = RetrofitClientInstance.getWebInstance();

    @Override
    public void userBooking(Booking booking, String discountCode, ResultCallback callback) {
        Call<ResponseBody> returnVal = web.newBooking(booking.getRouteID(), booking.getPassengerID(), booking.getQuantity(), discountCode);

        returnVal.enqueue(new Callback<ResponseBody>() {
            @Override
            public void onResponse(Call<ResponseBody> call, Response<ResponseBody> response) {
                System.out.println("RESPONSE!");
                String s = null;  // <- response is null here
                try {
                    if(response != null && response.body() != null)
                        s = response.body().string();
                    else
                        s = "Error with request!";
                    callback.onResult(s);
                } catch (IOException e) {
                    e.printStackTrace();
                }
                //System.out.println("BODY!\t" + s);
            }

            @Override
            public void onFailure(Call<ResponseBody> call, Throwable t) {
                //System.out.println("FAILED!!   " + t.toString());
                callback.onFailure(t);
            }
        });
    }

```

Once a request comes into Spring, a corresponding controller is called which then passes the data to the correct service to handle the business logic behind the request, for example a login request would be handled by the UserController who would call the UserService to handle encoding the password and to see if it matches any entry in the database through a UserRepository class. The repository classes at the spring level are responsible for the connection to the Database i.e. retrieving and inserting data. Through this sequence of actions our model is updated.

Junit Testing

We decided on Junit for the testing of our application. We used Mockito along with Junit, this allowed us to create mock objects and define the output of certain method calls.

The @Mock variables are mock objects created for our unit tests. The @InjectMocks variable is the mock class that utilises the mock objects upon creation. This allows for easy expandability as nothing is hard coded and the behaviour is defined at run-time. @Test defines a unit test.



```
@RunWith(MockitoJUnitRunner.Silent.class)
public class UserTest {

    @Mock
    private UserServiceImpl userServiceMock;
    @Mock
    private UserFactory userFactory;
    @InjectMocks
    UserController userControllerMock = new UserController();

    @Test
    public void registerUserTest() {
    User mockUser = new User( username: "Mock", password: "password", email: "mock@gmail.com", usertype: 1);
    when(userFactory.getUser( userType: "NORMAL_USER", username: "Mock", password: "password", email: "mock@gmail.com")).thenReturn(mockUser);
    when(userServiceMock.register(mockUser)).thenReturn("Success");
    String message = userControllerMock.addNewUser( name: "Mock", password: "password", email: "mock@gmail.com");
    assertEquals( expected: "Success",message);
    }

    @Test
    public void LoginTest() {
        String mockUsername = "Dummy";
        String mockPassword = "testing";
        when(userServiceMock.login(mockUsername,mockPassword)).thenReturn("Success");
        String message = userControllerMock.getUser(mockUsername,mockPassword);
        assertEquals( expected: "Success", message);
    }
```

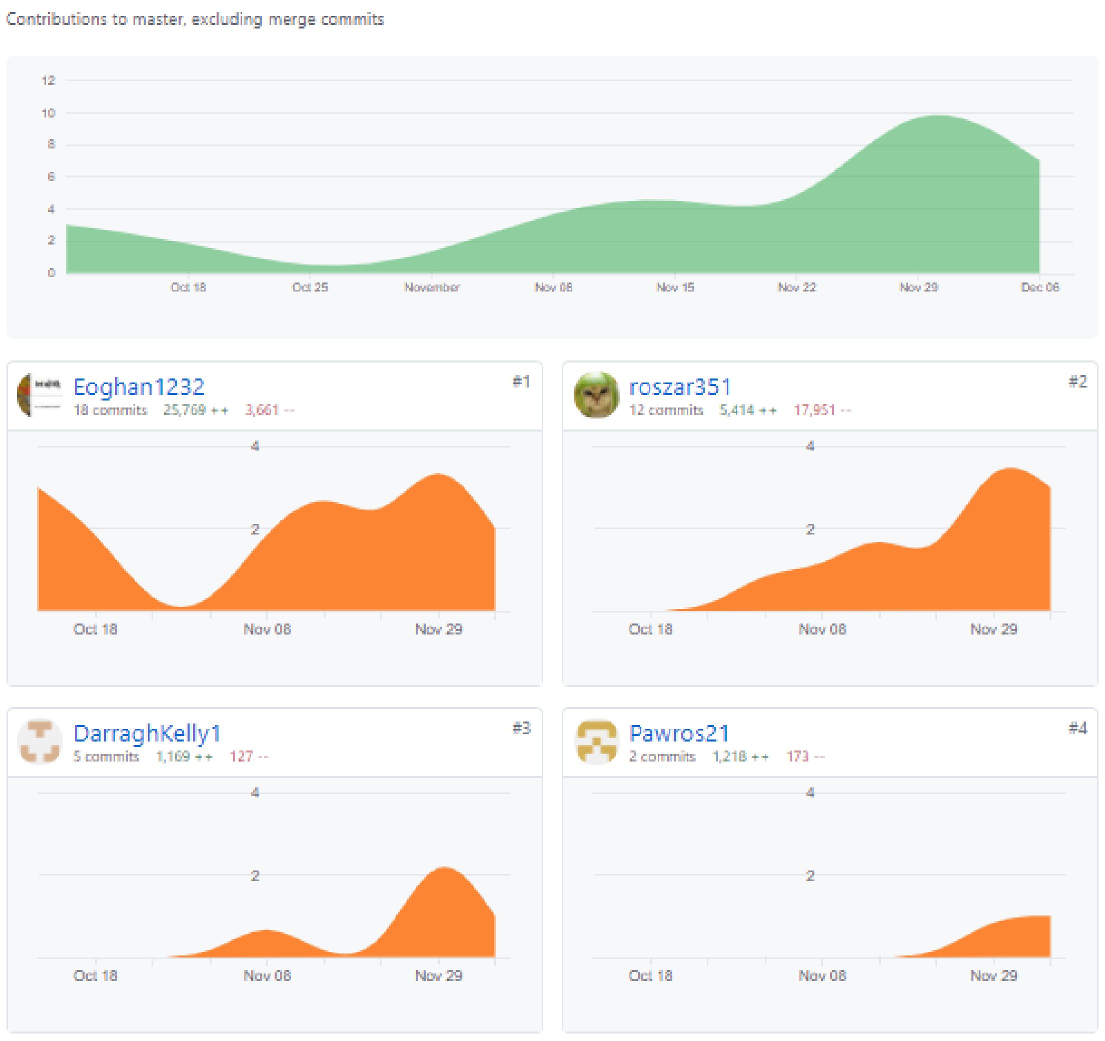
This Junit and Mockito unit testing implementation allow for easy automation and expandability. It could simply be set up to run nightly on the automation server Jenkins, testing our applications functionality to pick up on bugs if anything changes.

These tests will allow us to ensure that nothing breaks after each team member commits code. This would be done through Continuous Integration (CI), running the tests nightly. We could also expand the tests to further help develop new features and ensure that it does not break the current implementation. This approach is Test Driven Development (TDD), ensuring the software is fully developed before deployment.

GitHub – Version Control

https://github.com/Eoghan1232/CS4125_Project1_TeamBased

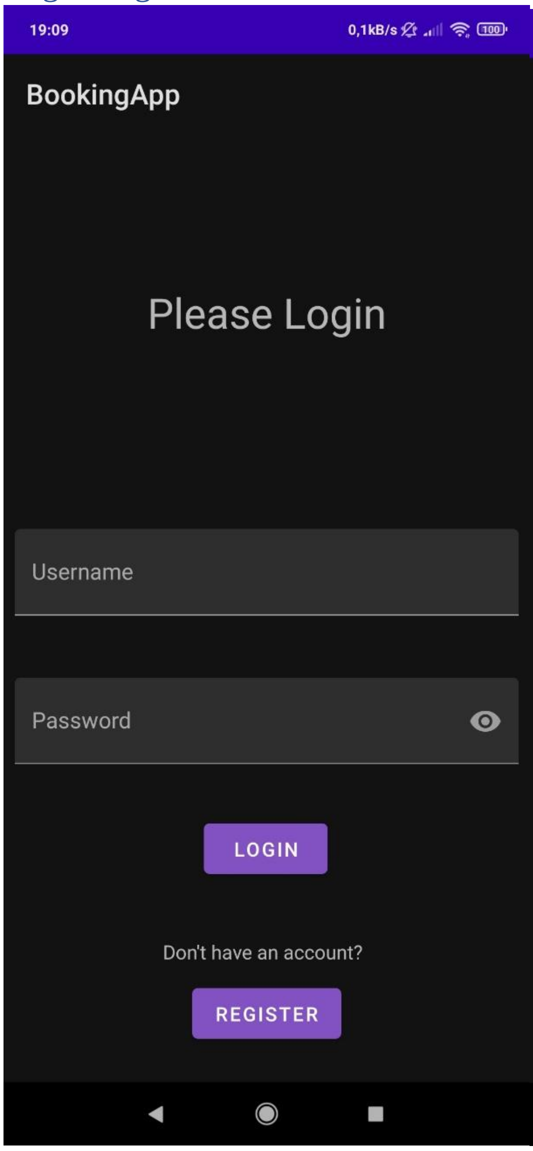
GitHub was used as a version control system for our project throughout the implementation stage. Once someone was finished or had started part of their implementation, they committed to the master branch. This helped us keep track with everyone's progress throughout the implementation. If something was broken or wrong, we could simply revert the specific commit.

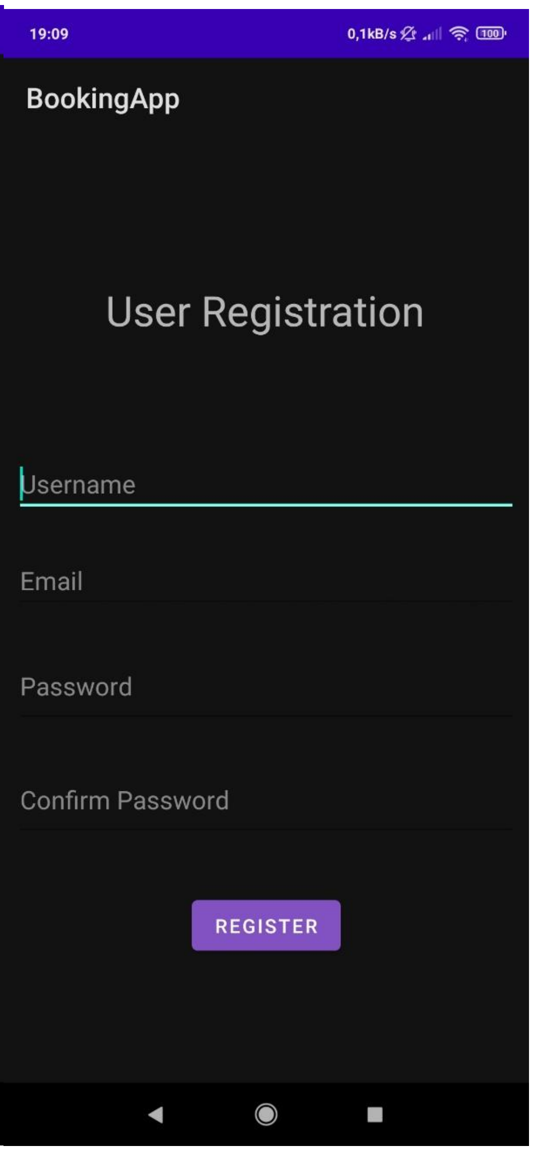


Note: Eoghan1232 - Eoghan Russell, roszar351 – Damian Skrzypek, Pawros21 – Pawel Ostach, DarraghKelly1 – Darragh Kelly

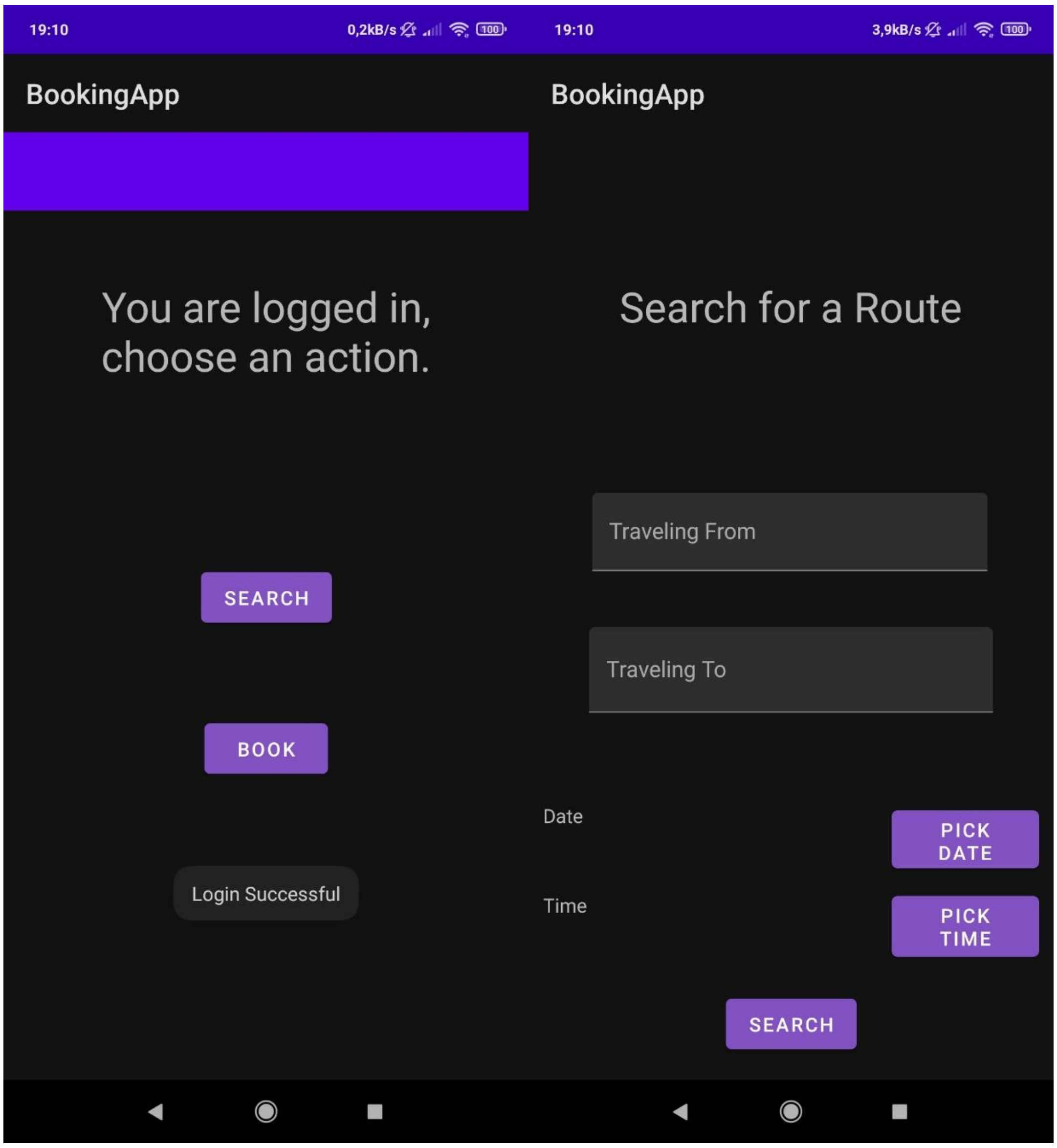
GUI

Login/Registration





Main Screen/Search

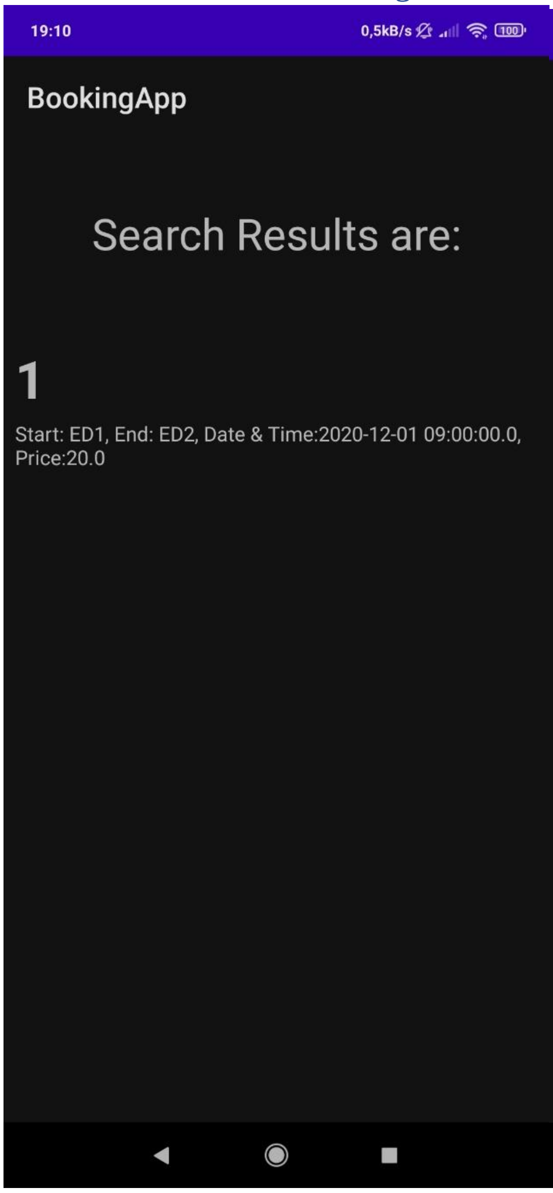


You are logged in, choose an action.

Login Successful

Search for a Route

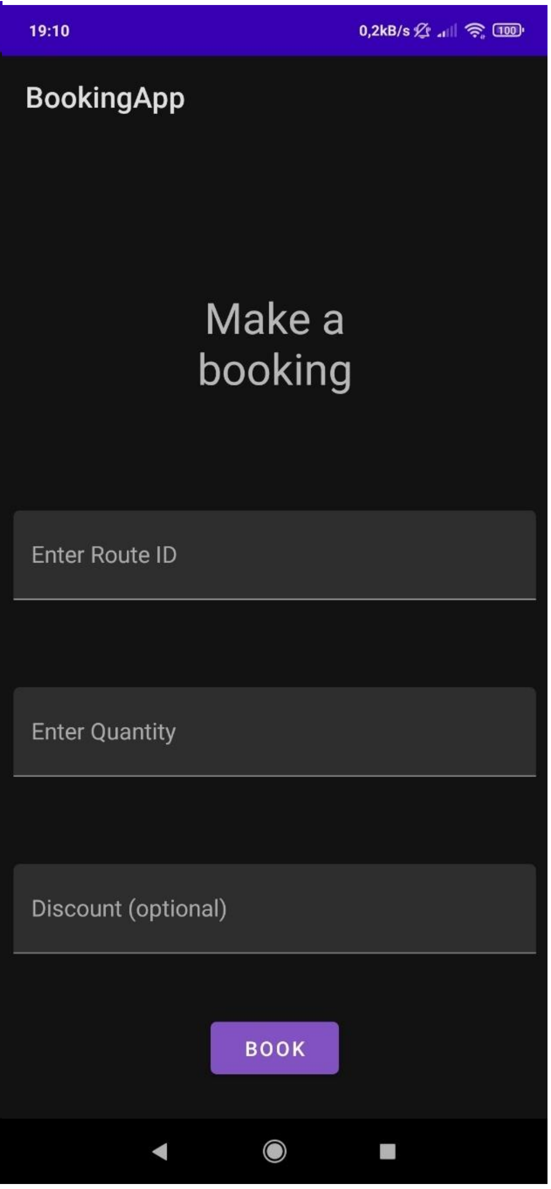
Search Result/Make Booking



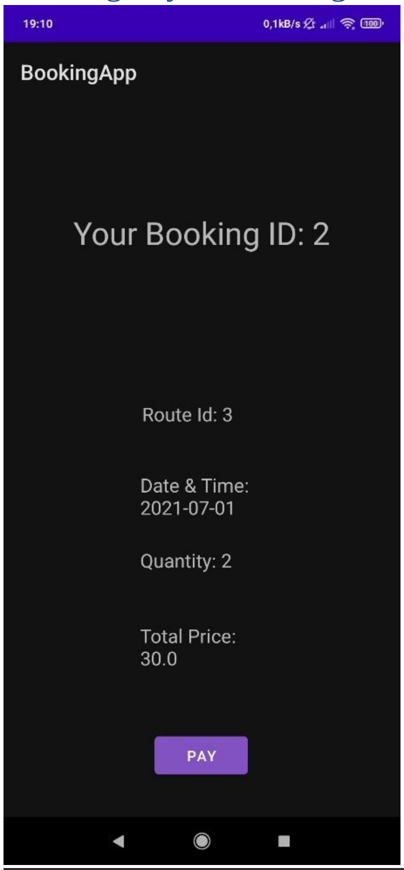
Search Results are:

1

Start: ED1, End: ED2, Date & Time:2020-12-01 09:00:00.0, Price:20.0



Booking Payment/Navigation Graph


19:10 0,1kB/s

BookingApp

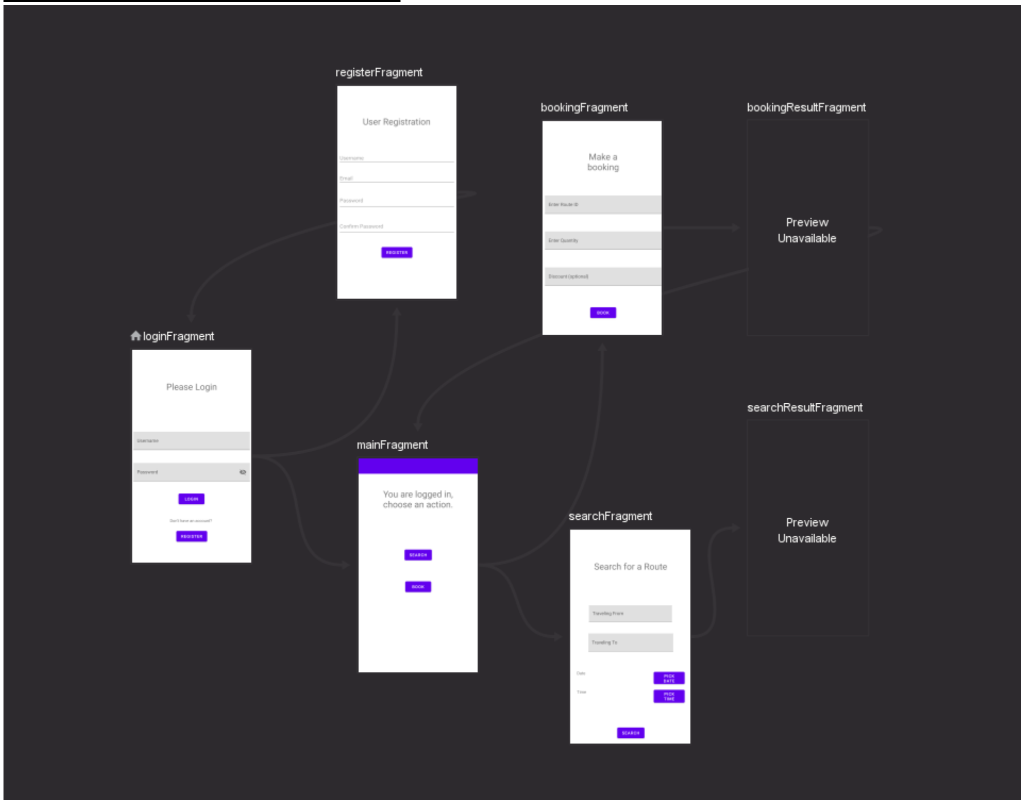
Your Booking ID: 2

Route Id: 3

Date & Time:
2021-07-01

Quantity: 2

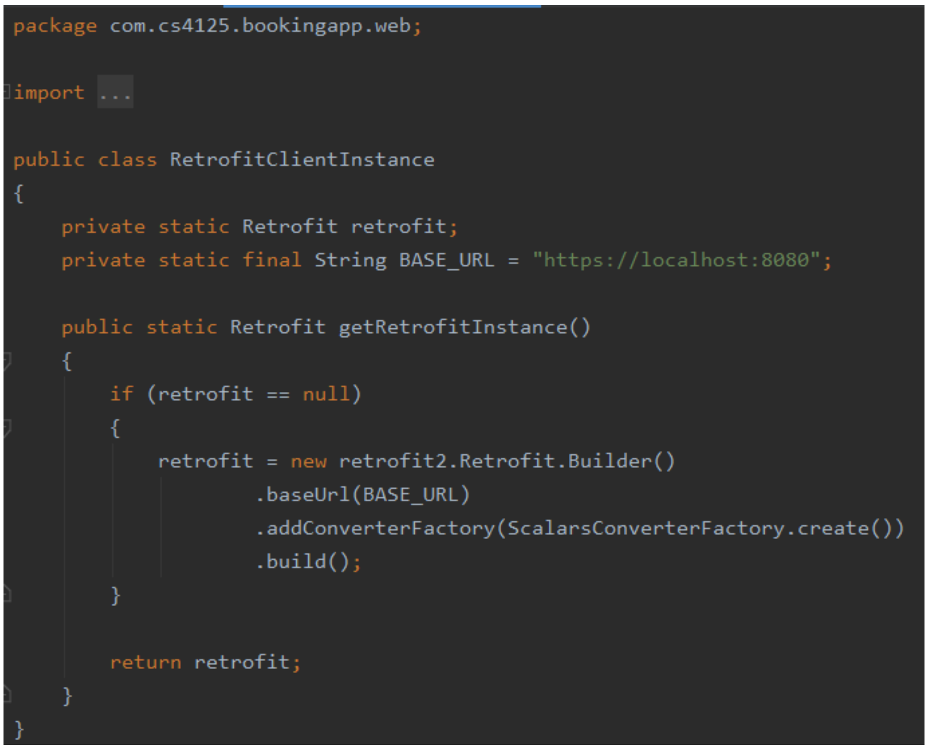
Total Price:
30.0



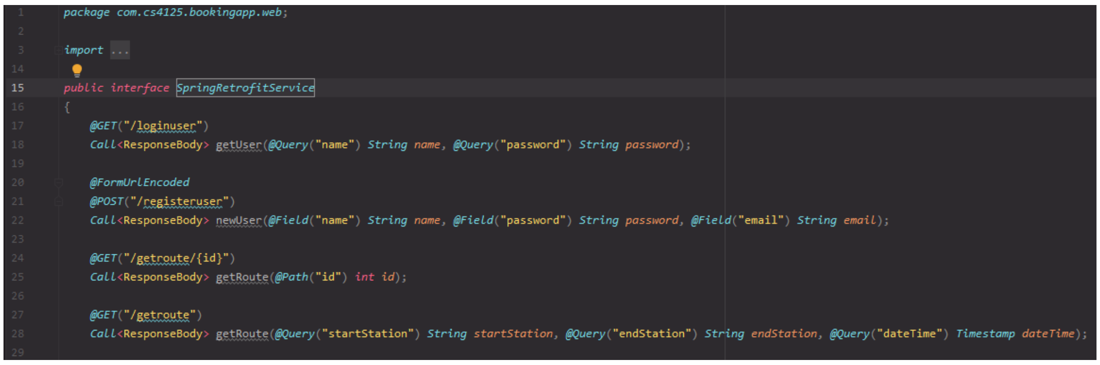
Added Value

REST architectural pattern

For our REST architectural implementation, we integrated the Retrofit API into our android application. Retrofit is a type-safe REST client, allowing us to send network requests. This connected our android application with our Spring Boot application (REST based webservice). This turned the REST API into a java interface for us to use. Retrofit uses the webservice interface we created to build the corresponding requests required.



```java
package com.cs4125.bookingapp.web;

import ...

public class RetrofitClientInstance
{
    private static Retrofit retrofit;
    private static final String BASE_URL = "https://localhost:8080";

    public static Retrofit getRetrofitInstance()
    {
        if (retrofit == null)
        {
            retrofit = new retrofit2.Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .addConverterFactory(ScalarsConverterFactory.create())
                    .build();
        }

        return retrofit;
    }
}
```

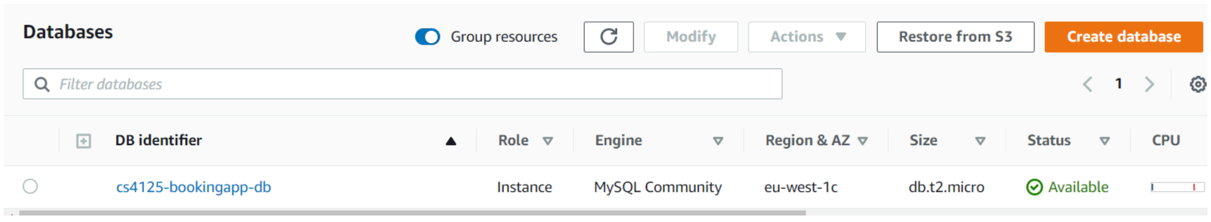


```java
package com.cs4125.bookingapp.web;

import ...

public interface SpringRetrofitService
{
    @GET("/loginuser")
    Call<ResponseBody> getUser(@Query("name") String name, @Query("password") String password);

    @FormUrlEncoded
    @POST("/registeruser")
    Call<ResponseBody> newUser(@Field("name") String name, @Field("password") String password, @Field("email") String email);

    @GET("/getroute/{id}")
    Call<ResponseBody> getRoute(@Path("id") int id);

    @GET("/getroute")
    Call<ResponseBody> getRoute(@Query("startStation") String startStation, @Query("endStation") String endStation, @Query("dateTime") Timestamp dateTime);

```

With REST architectural pattern we have created client-server constraints, this lets us separate user interface related concerns from the data storage concerns. The Spring Boot application deployed is fully concerned with business logic and data storage, this allows us to create separate software responsible for the user interface, in this case we have created an android app that uses retrofit2 to send REST requests to create/retrieve/change data. This is very useful in case we every need to create different user interfaces for example for different operating systems (IOS, Windows etc.) as we would be able to connect to the same services without having to recreate them for that specific OS.

AWS Database

For our database, we used AWS RDS (Relational Database Service) for our project. This allowed us to easily setup a database that can be accessed globally. We opted for AWS's free tier and created an instance of our database.

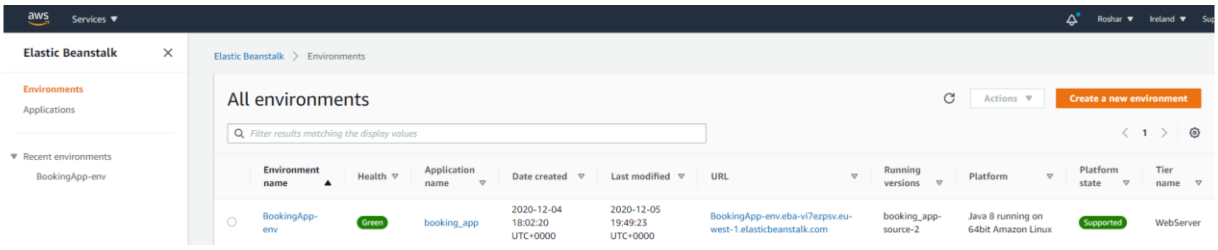


DB identifier	Role	Engine	Region & AZ	Size	Status	CPU
cs4125-bookingapp-db	Instance	MySQL Community	eu-west-1c	db.t2.micro	Available	

This database is in SQL, we could check the database through SQL Workbench to check the contents and tables. Spring Boot webservices was used to connect to the database, then we use Retrofit API to connect our android application to the AWS database.

Spring Boot Deployment

We have decided to use AWS Elastic Beanstalk (EBS) to host our Spring Boot Services, it allows for easy and quick deployment and management of applications in the AWS Cloud. EBS cuts out the complexity from setting up this type of web service, it allows to simply upload the packaged code (in our case JAR file) and EBS automatically handles the details of capacity provisioning, load balancing, scaling and monitoring of the applications health/errors.

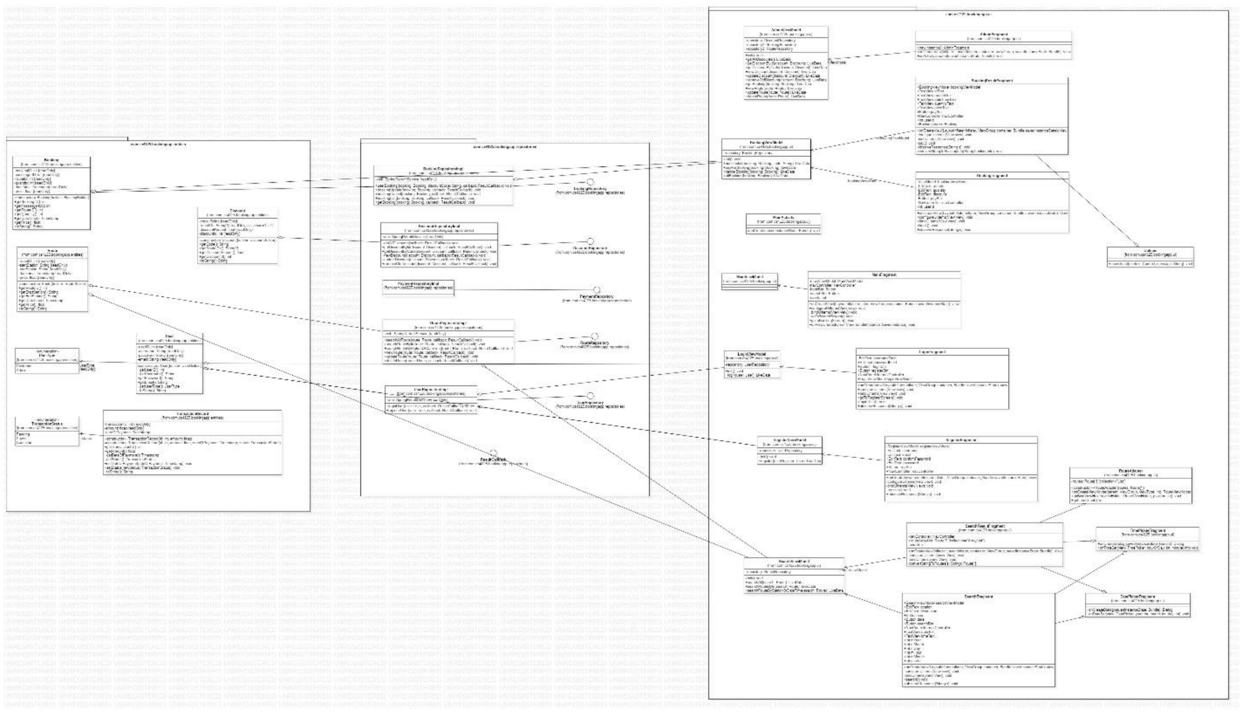


Environment name	Health	Application name	Date created	Last modified	URL	Running versions	Platform	Platform state	Tier name
BookingApp-env	Green	booking_app	2020-12-04 18:02:20 UTC+0000	2020-12-05 19:49:23 UTC+0000	BookingApp-env.eba-vi7ezpsv.eu-west-1.elasticbeanstalk.com	booking_app-source-2	Java 8 running on 64bit Amazon Linux	Supported	WebServer

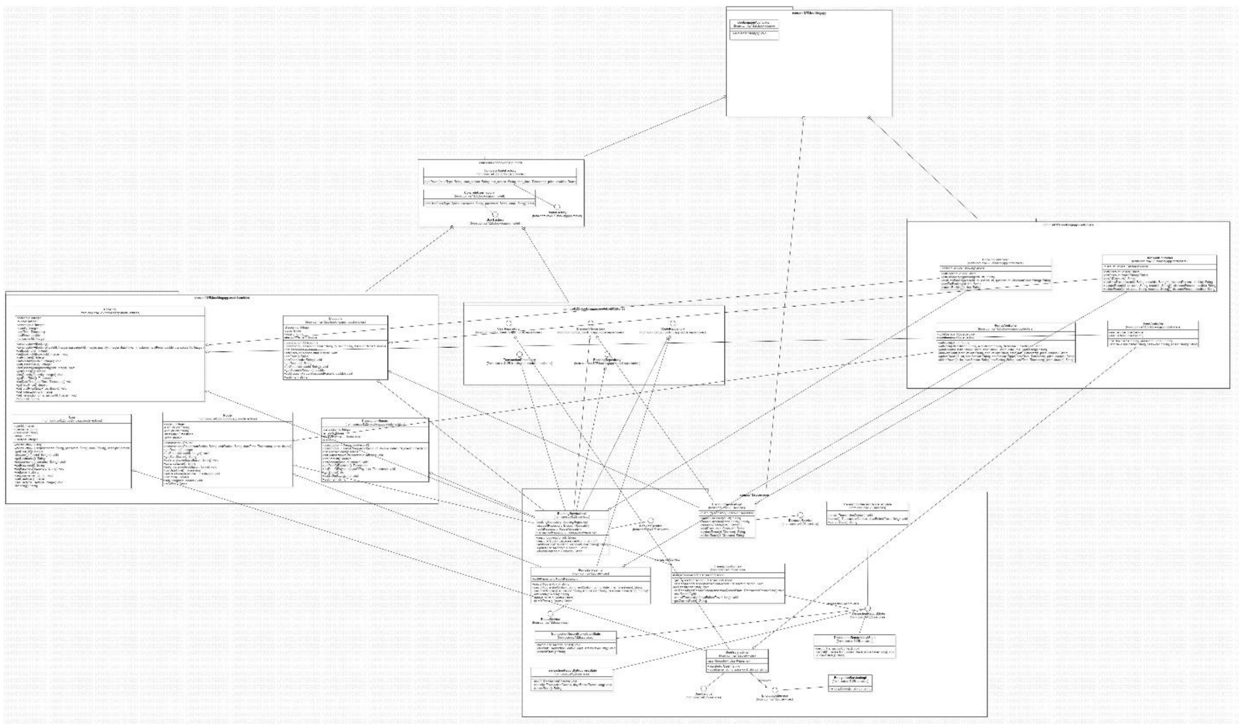
WE have opted for the free tier as it is sufficient for current state of the project, this deployment proved very useful when testing the GUI/Android Studio part of the project as all team members were able to connect to this EBS instance and don't have to host their own instance of the Spring Boot program.

Recovered Architecture and Design Blueprints

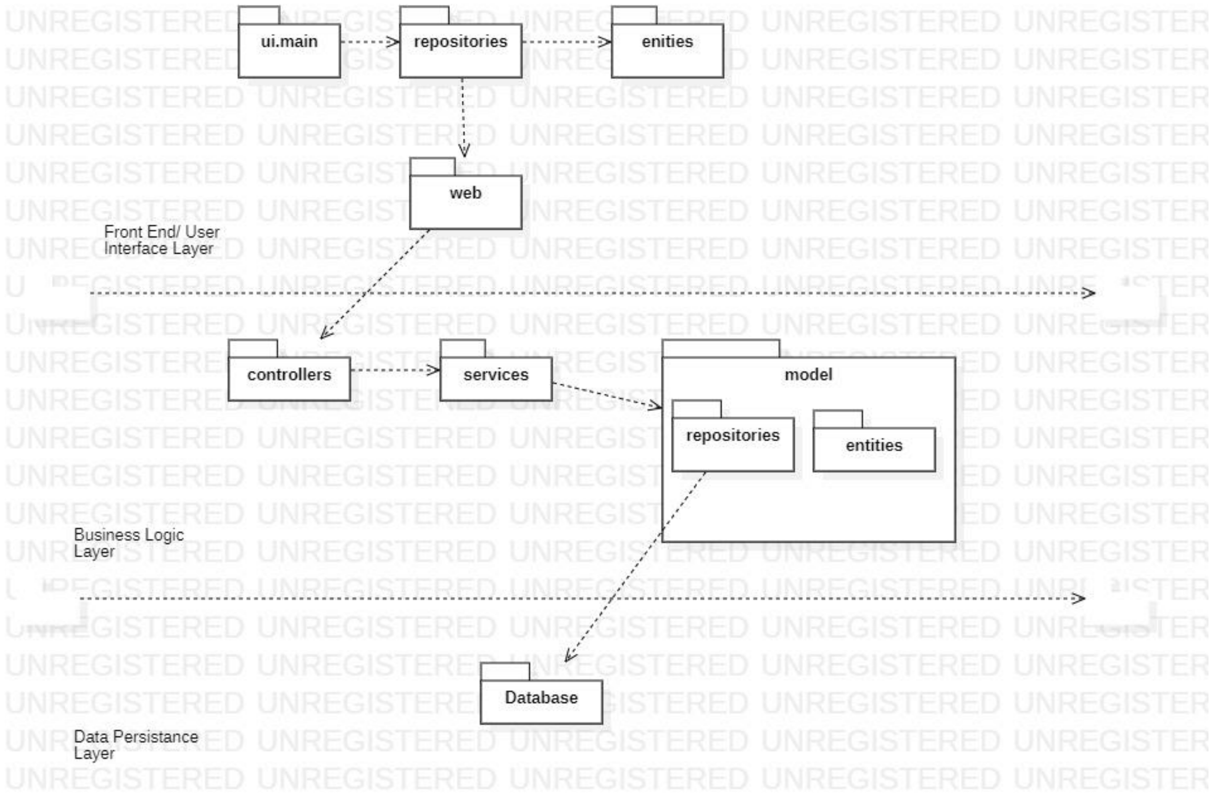
Android API



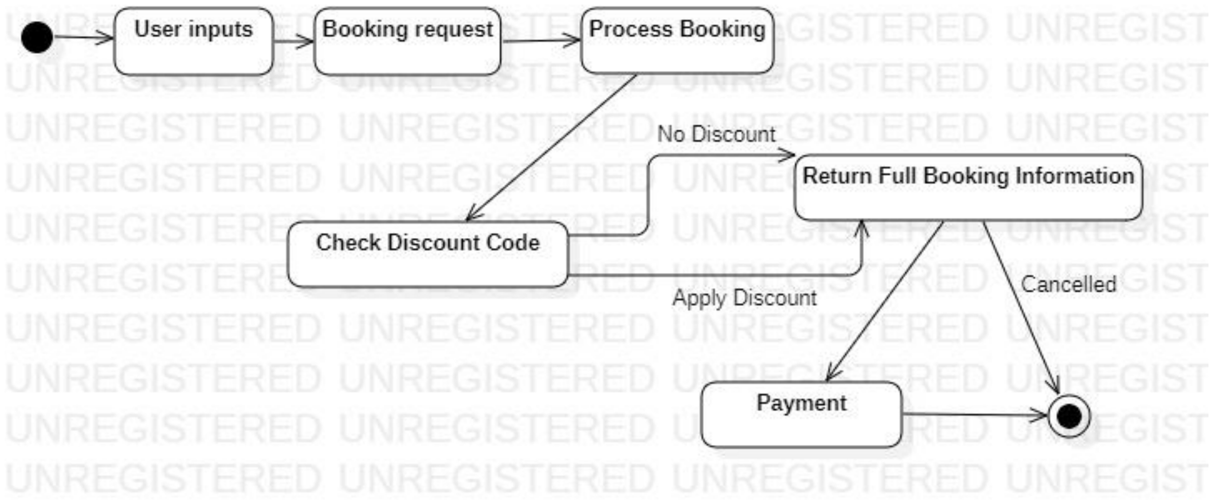
Spring API



Architectural Diagram



State Chart Diagram



Critique

Overview

During the implementation period, packages were added that weren't originally planned or evident in the design phase. Our initial designs were based on the initial plan set out after the first few weeks after the project specifications were released. Naturally, this was built upon and changed during the implementation stage. The differences between the analysis sketch versus the blueprint sketches outlined below illustrate the changes that were made. Overall, our implementation was consistent with our analysis sketches set out at the start of the project.

Changes between Analysis Sketches & Blueprint

Design Patterns

Factory Pattern: Our use of this pattern currently only creates the basic objects of the user. This would be expanded in the future to create other types of users and routes.

Analysis vs Design Artefacts

Substantial revisions were executed since the initial analysis sketch was completed. The initial concept was carried through, however, the number of classes has increased significantly since then.

Initially, we had a rough outline of our implementation, our diagrams at this point were basic and not very complex, they were in the one diagram. The number of classes grew greatly, this was mostly due to not considering how many classes are required to implement design patterns and the UI.

We were also unable to fully implement all the functionality we initially planned. We never got to implement the Admin user in the UI, this included the adding/removing routes/discounts. The business logic behind this is ready, however, there is no UI to implement these.

Future implementation possibilities

For future implementation, we would fully implement the Admin functionality in the front-end. This would include allowing the admin to add/remove routes/discounts.

We would like to Improve the UI usability. E.g. booking straight from the search result screen instead of having to go back out from searching to make a booking.

References

Balaji, S. and Murugaiyan, M.S. (2012) 'Waterfall vs. V-Model vs. Agile: A comparative study on SDLC', *International Journal of Information Technology and Business Management*, 2(1), 26-30.

Bassil, Y. (2012) 'A simulation model for the waterfall software development life cycle', *arXiv preprint arXiv:1205.6904*.

Brandom, R. (2019) *There are now 2.5 billion active Android devices*, The Verge, available: https://www.theverge.com/2019/5/7/18528297/google-io-2019-android-devices-play-store-total-number-statistic-keynote [accessed 08/12/2020].

Diagram Software and Flowchart Maker, available: https://www.diagrams.net/ [accessed 05/12/2020].

Download Android Studio and SDK tools, Android Developers, available: https://developer.android.com/studio [accessed 26/11/2020].

Spring Boot, available: https://spring.io/projects/spring-boot [accessed 28/11/2020].

StarUML, available: https://staruml.io/ [accessed 29/11/2020].

Why Is Java Preferred to Other Languages as a Building Block? , Techopedia.com, available: https://www.techopedia.com/2/28705/development/programming-languages/why-is-java-preferred-to-other-languages-in-building-technological-blocks [accessed 6th November].