Submitted by 44Mistaek (Team 09):

Eoghan Russell 17202124

Pawel Ostach 17214211

Damian Skrzypek 17217679

Darragh Kelly 17235545

CS4227 Booking App

Software Design & Architecture

Lecturer: JJ Collins & Conor Stephens

GitHub: <https://github.com/Eoghan1232/CS4227_Project>

Table of Contents

[1. Requirements 5](#_Toc69380728)

[1.1 Scenario 5](#_Toc69380729)

[1.2 Use Case Descriptions 5](#_Toc69380730)

[1.2.1 Use Case Diagram 5](#_Toc69380731)

[1.2.2.1 Use Case Description #1 6](#_Toc69380732)

[1.2.2.2 Use Case Description #2 7](#_Toc69380733)

[1.2.2.3 Use Case Description #3 8](#_Toc69380734)

[1.3 Quality Attributes 9](#_Toc69380735)

[1.3.1 Security 9](#_Toc69380736)

[1.3.2 Extensibility 9](#_Toc69380737)

[1.3.3 Usability 9](#_Toc69380738)

[1.3.4 Reliability & Performance 9](#_Toc69380739)

[2. Design Patterns 11](#_Toc69380740)

[2.1 Interceptor 11](#_Toc69380741)

[2.2 Memento 11](#_Toc69380742)

[2.3 Strategy 11](#_Toc69380743)

[2.4 Abstract Factory 11](#_Toc69380744)

[2.5 Command 11](#_Toc69380745)

[2.6 Proxy – 7th pattern 11](#_Toc69380746)

[2.7 Bridge Pattern 12](#_Toc69380747)

[2.8 Filter Pattern 12](#_Toc69380748)

[3. Architecture 13](#_Toc69380749)

[3.1 Architectural Diagram 13](#_Toc69380750)

[3.2 Class Diagram 14](#_Toc69380751)

[3.3 Sequence Diagram 14](#_Toc69380752)

[4. Code 15](#_Toc69380753)

[4.1 Abstract Factory 15](#_Toc69380754)

[4.2 Command Pattern 19](#_Toc69380755)

[4.3 Proxy 23](#_Toc69380756)

[4.4 Interceptor Pattern 26](#_Toc69380757)

[4.5 Strategy Pattern 28](#_Toc69380758)

[4.6 Memento Pattern 30](#_Toc69380759)

[4.7 Filter Pattern 33](#_Toc69380760)

[4.8 Bridge Pattern 36](#_Toc69380761)

[4.9 GitHub – Version Control 39](#_Toc69380762)

[5. Added Value 40](#_Toc69380763)

[5.1 Jenkins 40](#_Toc69380764)

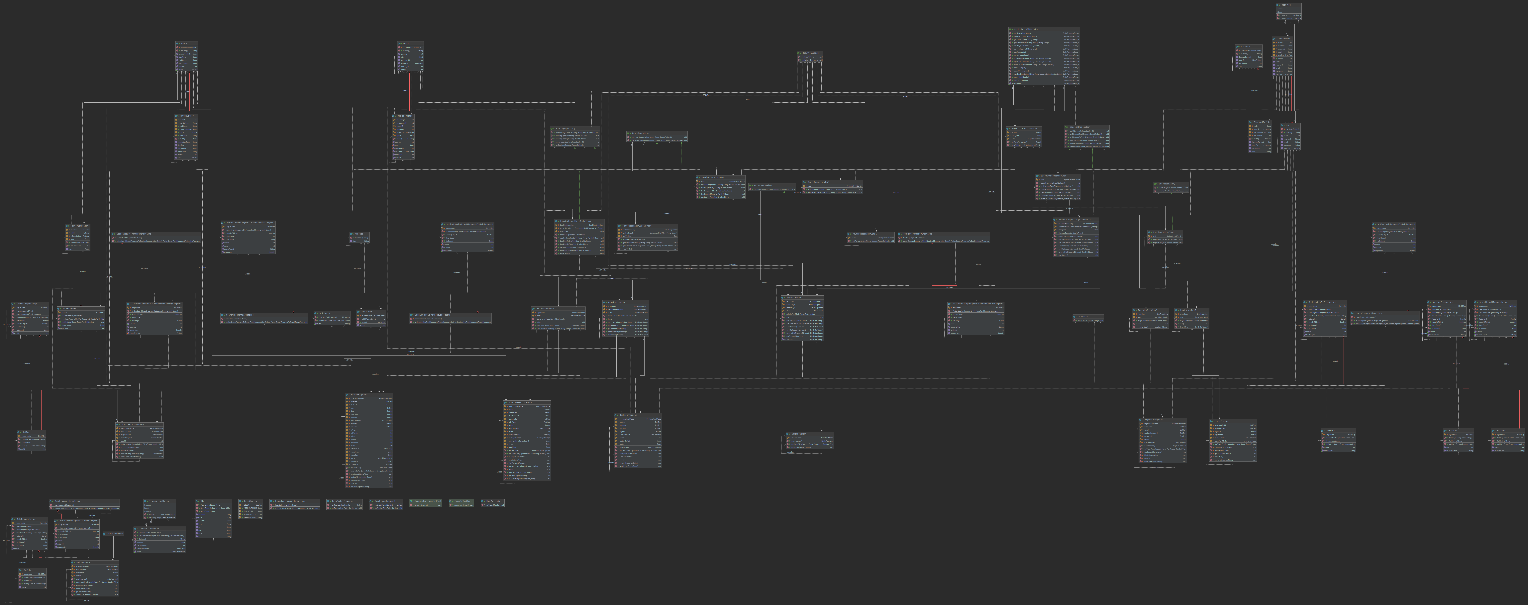
[5.2 Pathfinding Algorithm – Dijkstra’s 42](#_Toc69380765)

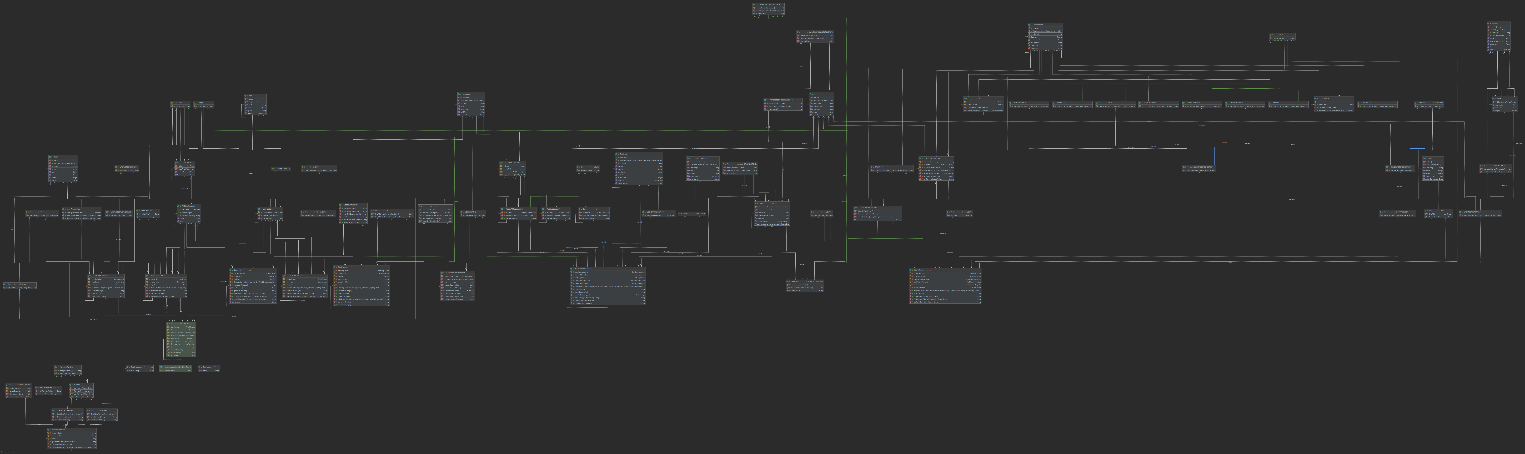
[5.3 Stripe (Payment Sandbox) 47](#_Toc69380766)

[5.4 CS4125 Class Diagram vs. CS4227 Class Diagram 51](#_Toc69380767)

[Diagram, schematic

Description automatically generated 51](#_Toc69380768)

[ 52](#_Toc69380769)

[ 52](#_Toc69380770)

[6. Testing 54](#_Toc69380771)

[6.1 Junit Testing 54](#_Toc69380772)

[6.2 REST Testing 57](#_Toc69380773)

[7. Problems Encountered 59](#_Toc69380774)

[7.1 Dependencies 59](#_Toc69380775)

[7.2 Other path finding algorithms 59](#_Toc69380776)

[8. Evaluation and Critique 60](#_Toc69380777)

[8.1 Critique of Admin User 60](#_Toc69380778)

[8.2 Critique of Graphical User Interface 60](#_Toc69380779)

[8.3 Critique of Command Pattern 60](#_Toc69380780)

[8.4 Critique of Memento Pattern 60](#_Toc69380781)

[8.5 Evaluation of Interceptor Pattern 61](#_Toc69380782)

[8.6 Evaluation of Bridge Pattern 61](#_Toc69380783)

[8.7 Evaluation of Jenkins 61](#_Toc69380784)

[9. Contributions of each team member 62](#_Toc69380785)

[9.1 Spring Boot 62](#_Toc69380786)

[9.2 Android Application 65](#_Toc69380787)

[9.3 Rest Assured Testing 67](#_Toc69380788)

[9.4 Team Member Contribution 67](#_Toc69380789)

[9.5 Total Code Developed 67](#_Toc69380790)

[References 68](#_Toc69380791)

# 1. Requirements

## 1.1 Scenario

This project is a transport ticket booking mobile application. The application will be capable of letting the user being able to search for a route between two destinations. The user can then apply a discount to the routes. The user then can book this route. They will be able to filter the routes based on the transport type.

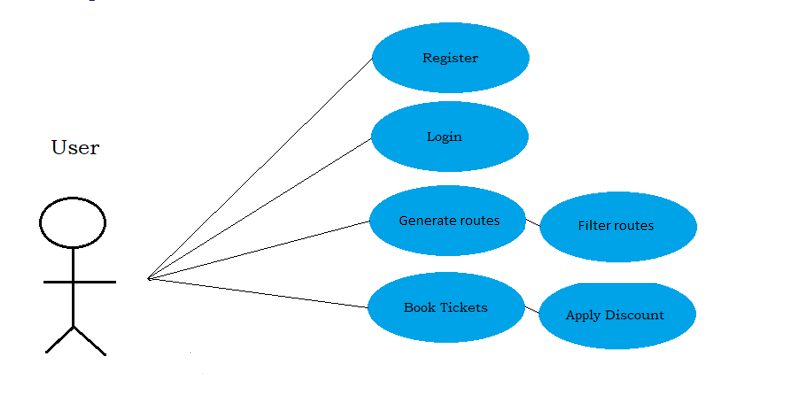
This system works as follows:

This project will be based on a mobile application for the practicality of this app being used on public transport. This application will have an easy to use user interface with a user friendly GUI, it will be simple and easy to navigate for any user.

The user will be able to download and install the app on their phones/smart devices. The user will be able to register and login into the app. The user contain will contain the details about them, this includes their username, email and password. When a user is logged in they will be able to search for a route between two destinations/stations. They will be able to filter this on their preferred transport type. Once they have selected their route they will be able to apply discounts to this. Then they will be able to book and pay for the ticket in the payment screen.

## 1.2 Use Case Descriptions

### 1.2.1 Use Case Diagram



### 1.2.2.1 Use Case Description #1

|  |  |
| --- | --- |
| **Use Case** | Book Route |
| **Goal in Context** | User is able to generate route and the book the route |
| **Scope and Level** | System |
| **Preconditions** | 1. User must be logged in 2. The user must have generated route 3. The user must pay if needed for the route |
| **Success End Conditions** | The user successfully books and pays for the generated route. |
| **Failed End Conditions** | The stations that the user input it does not exist |
| **Primary, Secondary, Actors** | User |
| **Trigger** | User tries to book a ticket for a route. |
| **Descriptions** | **Steps:**   1. User registers and logs into Booking App 2. User generates a route between two stations 3. User selects the route and books this ticket 4. The user pays for the ticket 5. Ticket is booked |
| **Extensions** | The station that user inputs does not exit. The route is not booked.  The connections does not exist between the user inputted stations. The route is not booked. |
| **Variations** | N/A |

|  |  |
| --- | --- |
| RELATED INFORMATION | Book Route |
| Priority: | Very High |
| Performance | 5 minutes |
| Frequency | 200/day |
| Channel to actors | N/A |
| OPEN ISSUES | N/A |
| Due Date | Release 2.0 |
| Superordinate’s |  |
| Subordinates |  |
|  |  |

### 1.2.2.2 Use Case Description #2

|  |  |
| --- | --- |
| **Use Case** | Book Filtered Route |
| **Goal in Context** | User is able to generate route, filter on these routes and the book the route |
| **Scope and Level** | System |
| **Preconditions** | 1. User must be logged in 2. The user must have generated route 3. The user must have filtered the route 4. The user must pay if needed for the route |
| **Success End Conditions** | The user successfully books and pays for the filtered generated route. |
| **Failed End Conditions** | The filters that the user input it does not exist for the connections to the stations |
| **Primary, Secondary, Actors** | User |
| **Trigger** | User tries to book a ticket for a filtered route. |
| **Descriptions** | **Steps:**   1. User registers and logs into Booking App 2. User generates a route between two stations 3. User filters the route using the filters that are available 4. User selects the route and books this ticket 5. The user pays for the ticket 6. Ticket is booked |
| **Extensions** | The filters that the user input it does not exist for the connections to the stations |
| **Variations** | N/A |

|  |  |
| --- | --- |
| RELATED INFORMATION | Book Filtered Ticket |
| Priority: | Very High |
| Performance | 5 minutes |
| Frequency | 200/day |
| Channel to actors | N/A |
| OPEN ISSUES | N/A |
| Due Date | Release 2.0 |
| Superordinate’s |  |
| Subordinates |  |

### 1.2.2.3 Use Case Description #3

|  |  |
| --- | --- |
| **Use Case** | Book Free Route |
| **Goal in Context** | User is able to generate route, user will be able book the route that is free. |
| **Scope and Level** | System |
| **Preconditions** | 1. User must be logged in 2. The user must have generated route 3. The user must pick the free route to book |
| **Success End Conditions** | The user successfully books the free route. |
| **Failed End Conditions** | The route cannot be booked as there are no free modes of transport on the connections |
| **Primary, Secondary, Actors** | User |
| **Trigger** | User tries to book a ticket for a route that is free. |
| **Descriptions** | **Steps:**   1. User registers and logs into Booking App 2. User generates a route between two stations 3. User selects the route and books this ticket that is free 4. Ticket is booked |
| **Extensions** | The route cannot be booked as there are no free modes of transport on the connections |
| **Variations** | N/A |

|  |  |
| --- | --- |
| RELATED INFORMATION | Book Free Ticket |
| Priority: | Very High |
| Performance | 5 minutes |
| Frequency | 200/day |
| Channel to actors | N/A |
| OPEN ISSUES | N/A |
| Due Date | Release 2.0 |
| Superordinate’s |  |
| Subordinates |  |

## 1.3 Quality Attributes

## 1.3.1 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?).

Previously transactions were handled by our backend entirely but with the addition of the bridge pattern we are now able to implement different payment service providers to handle payments, currently supporting Stripe which uses secure connections and encryption to ensure all user data is protected and safe (https://stripe.com/docs/security/stripe).

## 1.3.2 Extensibility

This application was built using a client-server architecture, and by using a Spring boot backend(server) and Android app frontend(client) this decouples them completely. This decoupling allows to reuse the backend for different applications for example allowing to create a website as a front end in the future or any other platforms if required.

The addition of the interceptor pattern has also greatly increased extensibility as it allows for new services being added without breaking the old code. This is done by registering new services in the interceptor chain which would execute them before the request finally reaches its destination where it would get processed and return the result.

## 1.3.3 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.

## 1.3.4 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.

Thanks to the addition of the cache proxy the Android app now has improved performance. The app no longer will need to communicate with the Spring boot instance every time a request is fired by the user, results of some of the more costly operation are now saved in the proxy. This allows for quick retrieval of data stored in memory, in case of a duplicate request.

# 2. Design Patterns

Brief description of what the patterns are, not how they are done in our project.

## 2.1 Interceptor

Interceptor pattern is a design pattern that comes under the behavioural pattern category. It is used when we want to do some pre-processing/post-processing with a request or response of the application. The request is essentially intercepted, and some filters are then applied to it. The setup for the interceptor pattern is as follows; filters perform certain tasks before and/or after executing a request, filter chain carries the filters and executes them in a defined order, target is the target object, filter manager manages the filters and filter chain, client is the object sending the request to the target. More information on how this pattern is implemented in this project is found in Code section.

## 2.2 Memento

Memento pattern is a design pattern which falls under the behavioural pattern category. Its used to restore the state of an object to a previous state. Memento uses three actor classes, memento which contains the state of an object to be restored, Originator which creates and restores states in memento objects and Caretaker which is responsible for restoring the object state from memento. More on how this pattern is implemented in this project is found in Code section.

## 2.3 Strategy

Strategy pattern is a design pattern that comes under the behavioural pattern category. The strategy pattern is used to change the class behaviour, or the algorithm being used at run time. Objects are created which represent different strategies and a context objects whose behaviour varies depending on the strategy object. The strategy object changes the executing algorithm of the context object. This can be done using a strategy interface defining an action, concrete strategy classes implementing the interface and context classes which uses a strategy. More information on how these are used in this project is found in Code section.

## 2.4 Abstract Factory

Abstract Factory Pattern comes under creational patterns and includes a super-factory class which is responsible for creating other factories. In this pattern an interface is responsible for creating a factory of related objects without specifying their classes. The generated factories can give the objects required as per the Factory Pattern. More information how this pattern is used in this project is found in Code section.

## 2.5 Command

Command Pattern is a design pattern that falls under the behavioural pattern category. In this pattern all requests are wrapped under an object as a command and are passed to the invoker object. This invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object where it is executed. More information how this pattern is used in this project is found in Code section.

## 2.6 Proxy – 7th pattern

Proxy Pattern is a structural design pattern, it provides a placeholder for another object. This proxy object controls the access to the original object, allowing to perform actions before or after a request gets through to the original object. This pattern is often used when there is some sort of massive object in the project that consumes huge amounts of resources and is not necessarily needed at all times. A proxy is able to receive the request and then decide if the object should be created or if the request can be handled through some other way for example retrieving a cached result. This is achieved by having the proxy class use the same interface as the original service object, and then the original object use this proxy class instead of the original service object. If proxy is not able or is not required to do anything with the request it can just pass the request to the original object and return the result. More information how this pattern is used in this project is found in Code section.

## 2.7 Bridge Pattern

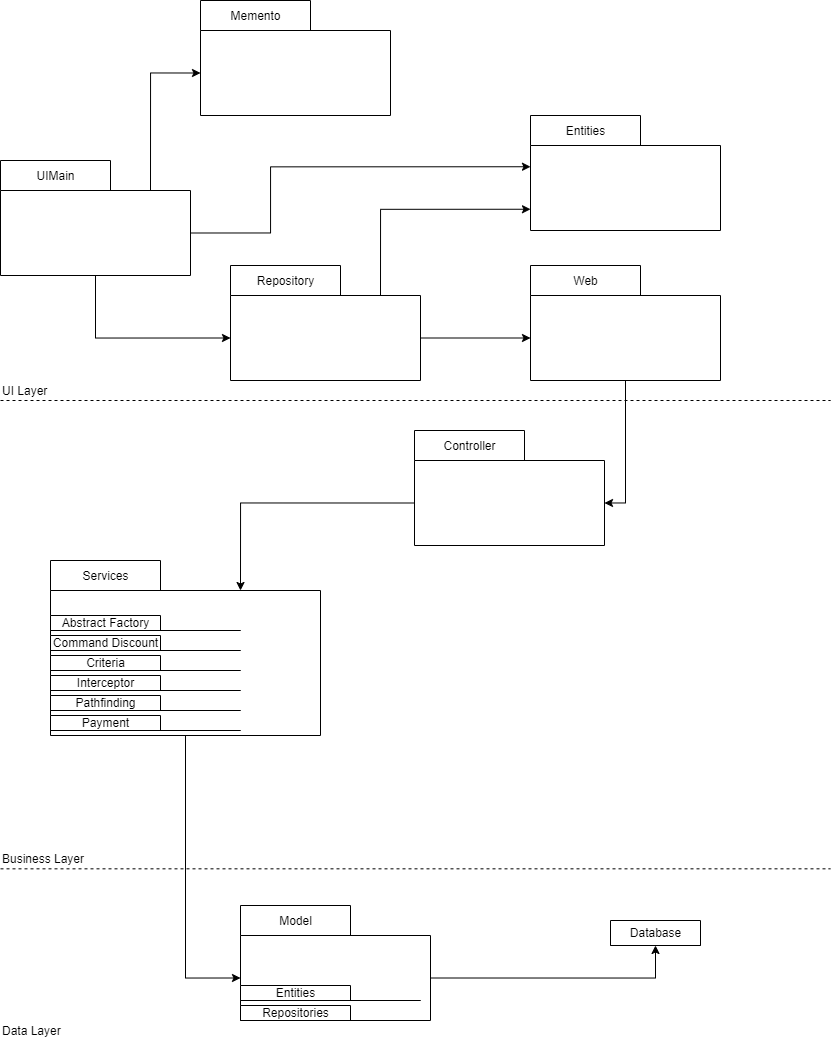
Bridge Pattern is a design pattern that is used when we need to decouple an abstraction from its implementation. With this these two can vary independently. This type of design patter is the structural pattern type as this pattern decouples implementation class and abstract class by giving them a bridge structure between these two. There will be more information on how this pattern was used in this project in the Code section.

## 2.8 Filter Pattern

Filter pattern also known as the Criteria pattern is a structural design patterns that lets a developer be able to filter on a set of objects. Then chain them in a decoupled way through logical operations. It lets us combine multiple criteria to obtain a single criteria. There will be more information on how this pattern was used in this project in the Code section.

# 3. Architecture

## 3.1 Architectural Diagram



## 3.2 Class Diagram

These diagrams are quite large and might be best viewed in their respective png files.

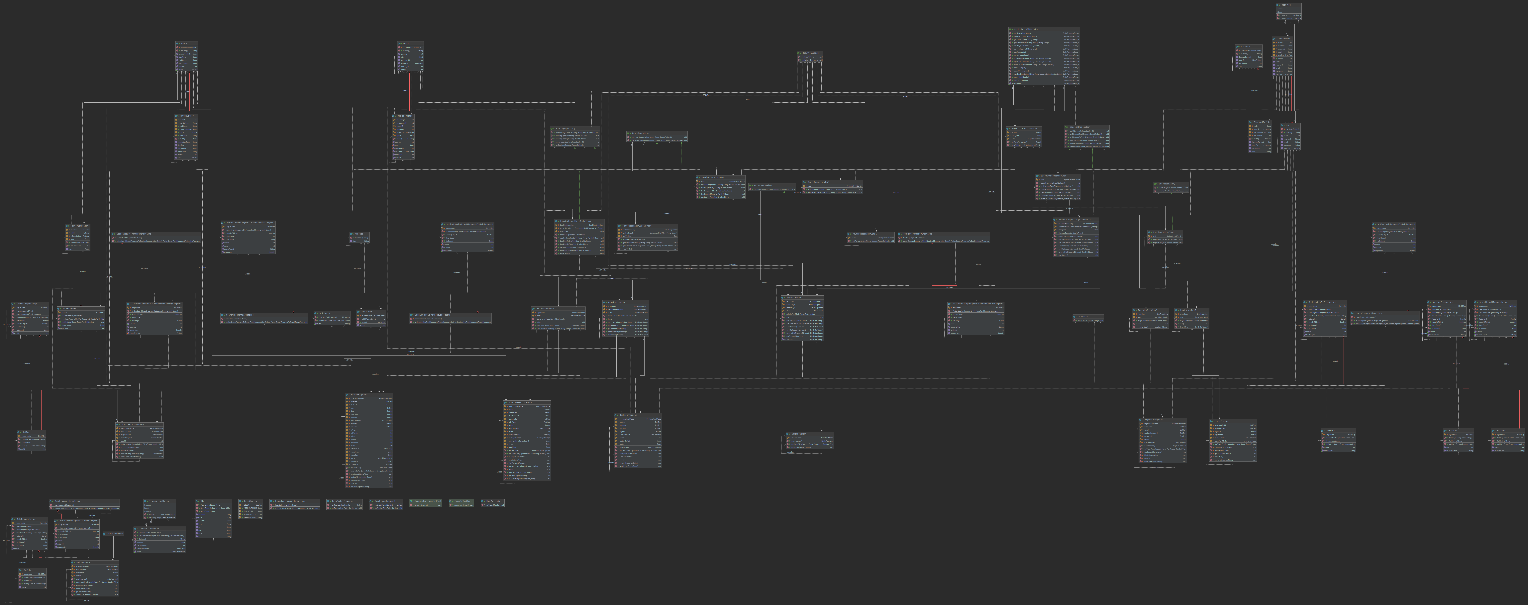


Figure : UML Class Diagram Android

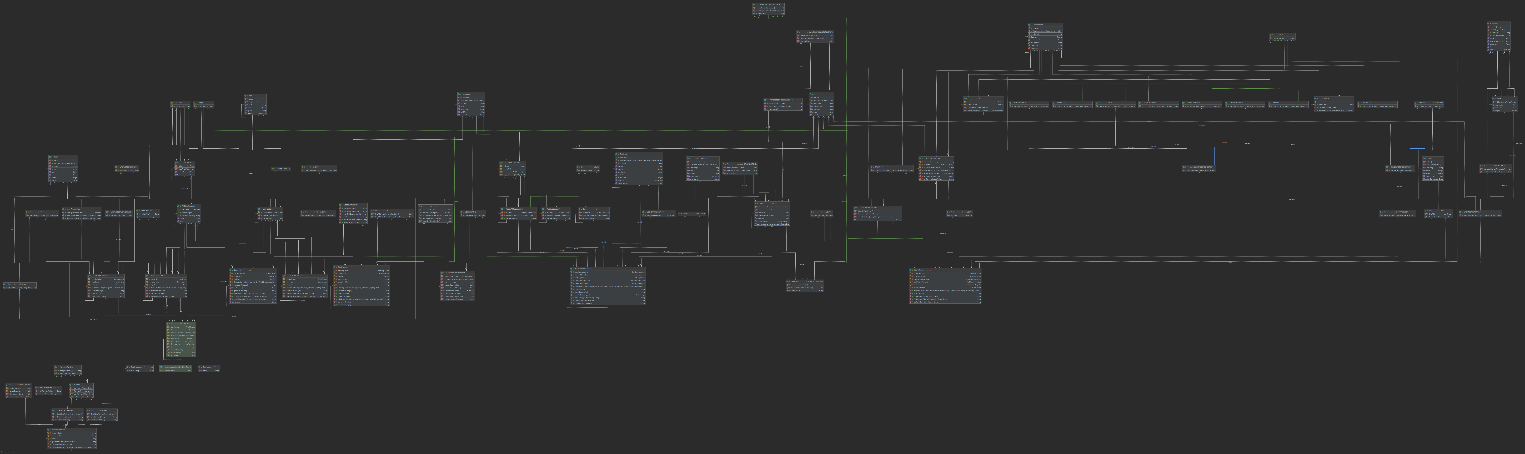
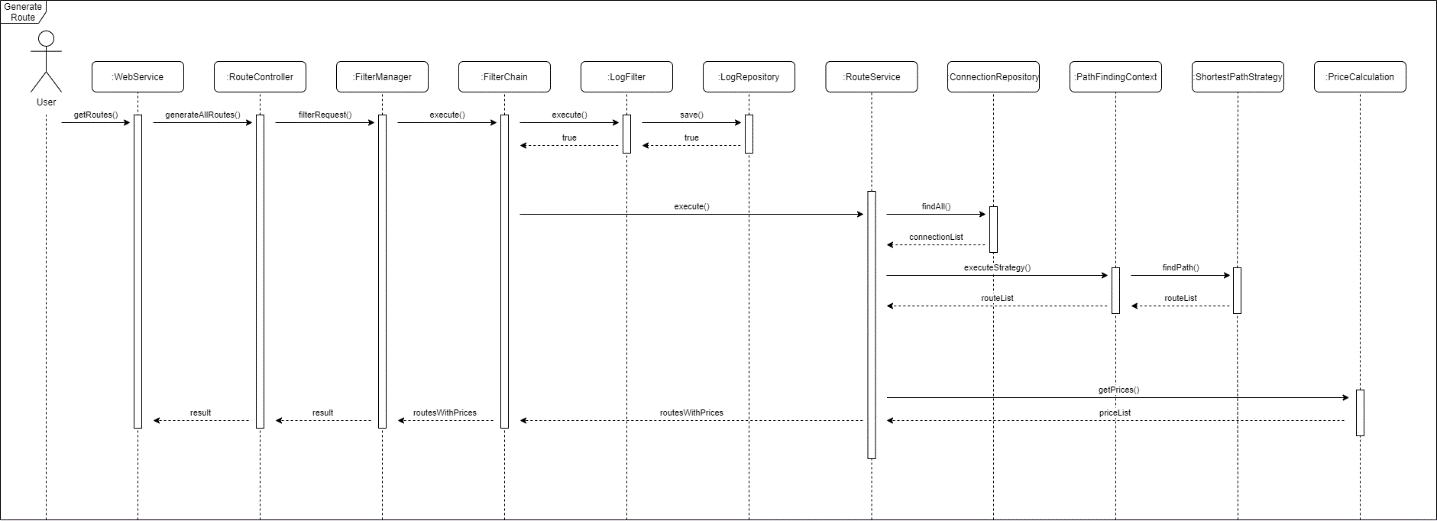


Figure : UML Class Diagram Spring

## 3.3 Sequence Diagram

Generating a route.



# 4. Code

## 4.1 Abstract Factory

The Abstract Factory Pattern was implemented to handle the creation of different filters/criteria for generating routes. The first type of criteria filters out connections based on the transport type selected by the user. The second type of criteria has currently not been implemented as we were only able to implement one type of path finding algorithm i.e. the shortest path algorithm which only returns one path found, more on this in the Section 7.2. This second type of criteria would filter routes based on different parameters the user specified e.g. sort by cheapest routes, sort by shortest distance, only free routes etc. Filters come into the route service class as strings and using the Abstract Factory pattern allows to pass the type of criteria to the CriteriaFactoryProducer to receive the corresponding factory and then use this factory to produce the required criteria e.g. use the ConnectionCriteriaFactory and pass it the filter “WALK” to receive a WalkCriteria which is then used to filter out the connection list. The use of this pattern allows the user to pick their preferred filters and then the backend can determine the required factory at runtime, as mentioned earlier currently only criteria for connections is used and as such this pattern couldn’t be fully utilised.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . AbstractCriteriaFactory.java

Above figure shows the code for the AbstractCeriteriaFactory class which factories extend and return the corresponding criteria.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . CriteriaFactoryProducer.java

Above figure shows CriteriaFactoryProducer which is used to create the required factory depending on the factoryType parameter i.e. “CONNECTION” for connection factory and “ROUTE” for route factory.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . ConnectionCriteriaFactory.java

Figure 5 shows a ConnectionCriteriaFactory which creates and returns connection criteria.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . RouteCriteriaFactory.java

Figure 6 shows a RouteCriteriaFactory which creates and returns route criteria.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . Use of the Abstract Factory Pattern in RouteServiceImpl.java

In above figure lines 86 to 115 contains a switch case statement that was used to create the required criteria, this got replaced by lines 82 to 85 by using the AbstractFactory pattern. This makes the code more readable.

## 4.2 Command Pattern

The Command Pattern was used for applying discounts once a user requests to book a ticket. Discounts are found using their code which is a string variable. Before the Command Pattern was implemented all the logic for finding and applying discounts was done completely in the BookingServiceImpl class, this made the addBooking method inside it very large, hard to read and understand, it also would prove very difficult if the system required to support new type of discounts or if an old discount type had to be changed.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . DiscountOperation.java

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . ApplyDiscount.java

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . ApplyAllDiscount.java

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . DiscountContext.java

The DiscountContext seen in Figure 11 is used to keep track of the discount object and if it has already been applied. The two currently supported operations are applying a single discount seen in Figure 9 and applying multiple discounts seen in Figure 10, these operations are used in the service class depending on if the user has specified multiple discounts i.e. if the string contains a ‘&’ character then the multiple discounts command is used, otherwise the single discount command is used, this process can be seen in Figure 13.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . DiscountInvoker.java

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . Use of Command Pattern for Discounts in BookingServiceImpl.java

With the pattern implemented the service class seen in Figure 13, is able to go through the codes to create the required discounts and commands, finally the DiscountInvoker class seen in Figure 12, handles executing all the discount operations. This allows the system to be more extensible allowing future discounts to be added as operations very easily e.g. in the future a timed discount could be added and a time discounts operation would be responsible to perform the check if the discount is eligible to be applied.

## 4.3 Proxy

Proxy Pattern was used on the client side in the form of a Cache Proxy which allows to store results in some form a cache, in the case of this project in the form of a ConcurrentHashMap. Before the Proxy class implementation classes would directly use the Repository classes to request data from the Spring Boot backend. With the Proxy classes implemented, requests now how to first go through a Proxy, this class then checks the cache to see if the request has already been called recently i.e. under 20 minutes, and if so it will return the cached result. In the case where the cache result is not present, 20 minutes have passed or the request should not be cached as it always has to be processed on the backend the proxy will send the request to the original class and that class will handle firing the request to the backend.

The implementation of the Proxy class has two main advantages, first it helps with performance as returning the cached results is much quicker than sending the request to the backend and waiting for result. Second it helps with extensibility as Proxy allows to perform actions on the request before it is sent or on the result before it is returned back the requesting class. For example if the project were to be extended and requests now required some form of encoding of string variables, the Proxy could easily be extended to perform this encoding and leaving the original classes unchanged.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . BookingRepositoryCacheProxy.java

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . DiscountRepositoryCacheProxy.java

Obraz zawierający tekst

Opis wygenerowany automatycznie

Figure . RouteRepositoryCacheProxy.java

Figures 14, 15 and 16 showcase the different proxy classes implemented, they consist of a ConcurrentHashMap seen at line 13, which saves response values for certain requests e.g. line 40 in Figure 16. When the request is called again the map is checked e.g. line 33 and if value is found it is returned. Finally, currently all of the proxy classes are set to get new value if the time between requests have been longer than 20 minutes, this is done by code on lines 23-33.

## 4.4 Interceptor Pattern

The Interceptor Pattern was implemented to add functionality on requests handling. It adds a log filter to allow logging timestamps on each request done between the database and UI. For the case of this project only one filter was applied as proof of concept but the code has been implemented in a way to add extensibility for another filters when required. This was done through the use of request which in our implementation specifies the area of the application it refers to and later provides the request query.

Previously each Controller was returning the request directly to the respective Service Implementation class. Now the request is being passed to the filter manager of the interceptor pattern where the required filters are applied and executed along with passing the call to the execute method of the target using which the result is passed further to the Service Implementation class where the required methods are called and executed. In each implementation class the request is handled using switch statement which improves the code quality and readability and provides extensibility to each class if new methods were implemented. Passing the request result itself allows for applying different filters in the future that would be able to alter the state of the result String. Please find the code in the order of execution, based on the booking example, below.



Figure : Request Creation based on Response

Figure 17 shows the creation of the request and passing it to the manager i.e. the interceptor to be processed.

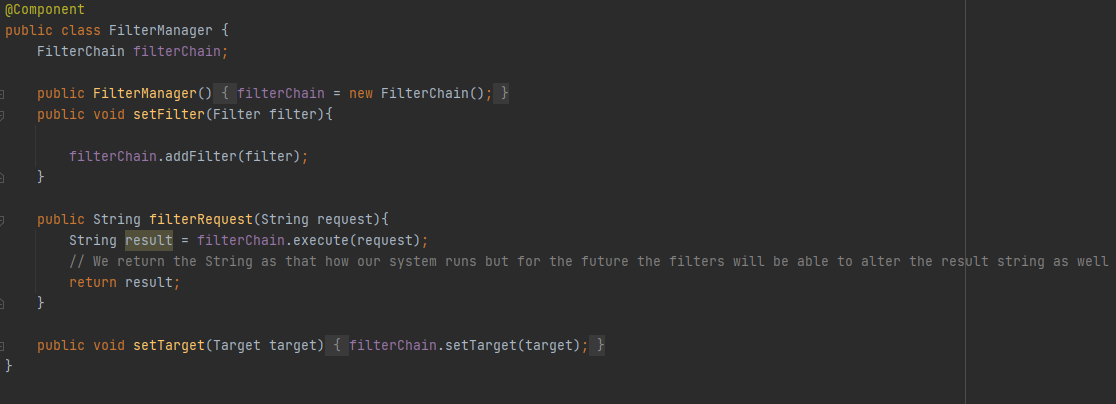


Figure : FilterManager class

FilterManager as seen in Figure 18, is able to contain multiple filters through the filterChain variable, this filter chain is then use to execute all of the added filters.

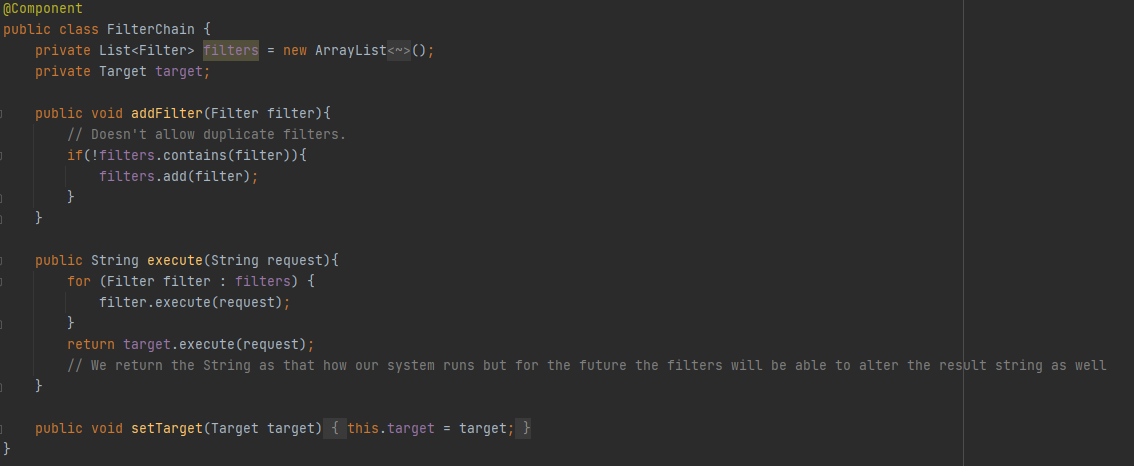


Figure : FilterChain class

FilterChain class contains filters and the final target. It calls the execute method on all of the filters and the target processing the request and returns the result.



Figure : LogFilter class implementing Filter interface

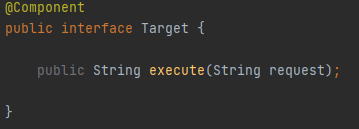


Figure : Target interface

LogFilter is a filter implemented for this project that stores the request in a log table in the database, while Target is an interface that allows services to be used as targets for the interceptors.

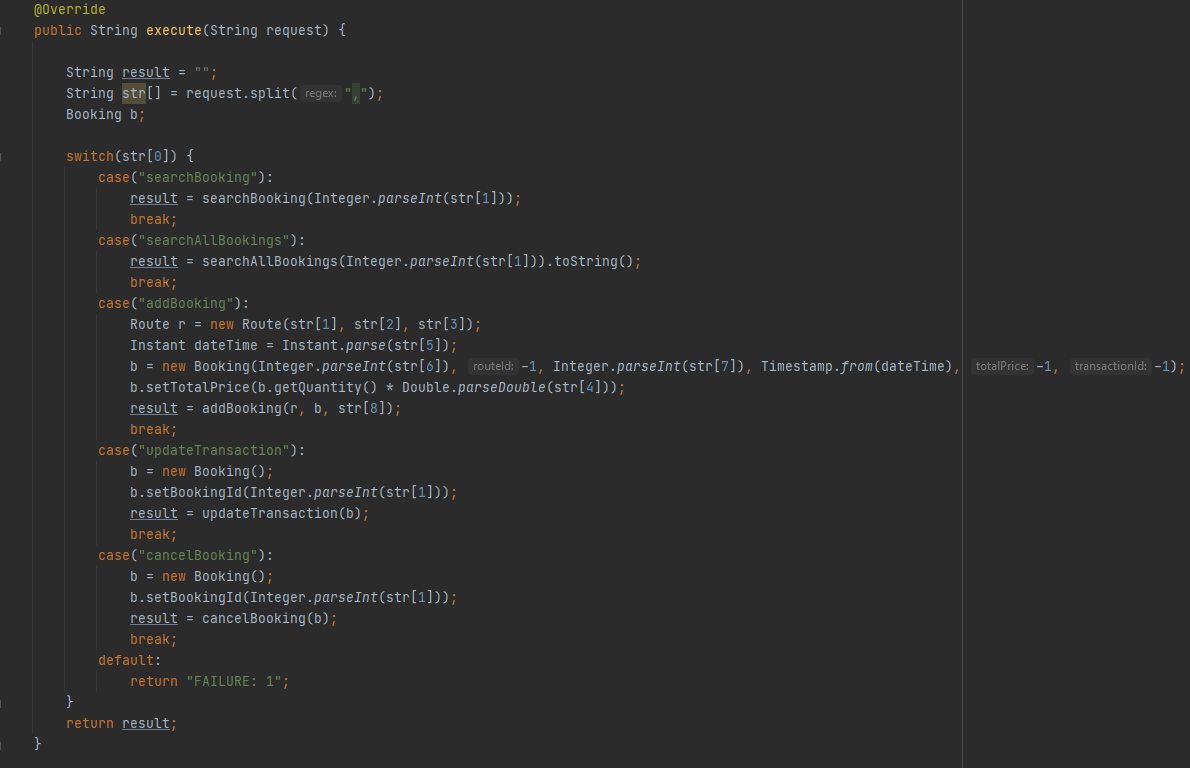


Figure : BookingServiceImplementation class overriding execute method from target class

Figure 22 showcases BookingServiceImplementation class being used as the target and in its execute method it breaks down the request and calls the corresponding function with needed parameters.

## 4.5 Strategy Pattern

The strategy pattern was created for applying the path finding algorithms to route services based on the desired criteria. At the moment there is only one search algorithm implemented, more on this in Section 7.2. The application of this pattern creates a context for the strategies to be implemented and the strategies itself.

Whenever routes are being searched for the strategy context is being set. Further through that context class the individual desired strategy is being called. In this case it is the shortest path strategy which applies shortest path algorithm to the list of nodes to find the best route based on the user defined criteria. The algorithm implemented in this project is Dijkstra’s shortest path algorithm. Please find the code in the order of execution based on the example below.

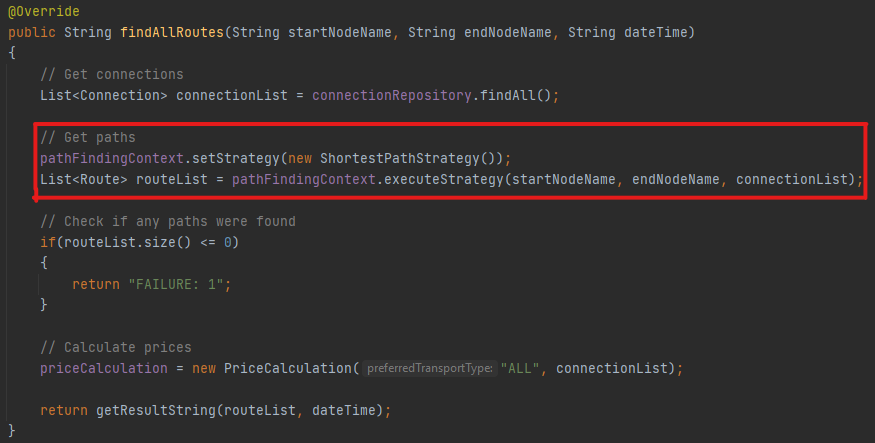


Figure : RouteServiceImplementation class with the strategy part highlighted

In Figure 23, the shortest path finding algorithm is set, and the strategy is then executed. For this project, it calls our Dijkstra’s path finding algorithm to return the shortest path from the start and end node.

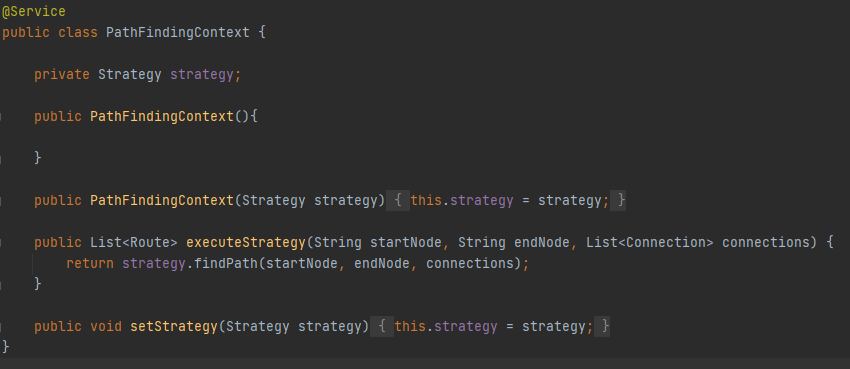


Figure : PathFindingContext class

The PathFindingContext is where the strategy is set, the executeStrategy method is where the strategies algorithm gets executed.



Figure : Dijkstra's Path finding algorithm

This is part of the Dijkstra’s algorithm, more on this in Added Value, Section 5.2.

## 4.6 Memento Pattern

The Memento Patter was implemented to save the states of textboxes on certain screens on the Android side. This allows a user to come back on its action and click the undo button to have the previous state of the text field. That functionality was applied to the search booking fragment where the user inputs values for location, destination, date and time and also can change those values freely at any given time as all the values are updated on change. The undo button is shown on Figure 14. Please find the code in the order of execution for the Memento Pattern below.

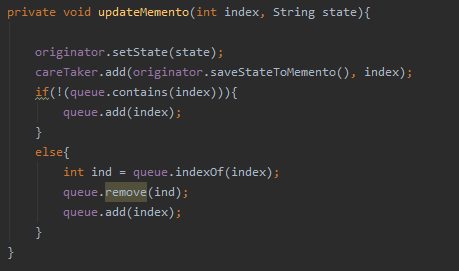


Figure : UpdateMemento method inside the SearchFragment class

The updateMemento method is called during operations, such as when a button is clicked, etc. It keeps track of what text box, variable has been changed and updates the memento state to store it so it can be undone later when needed. This is done by using a queue, checking if the index referring to the specific box is in there already, if its not, add it to the queue, otherwise find it in the queue, remove it and add it to the end of the queue.

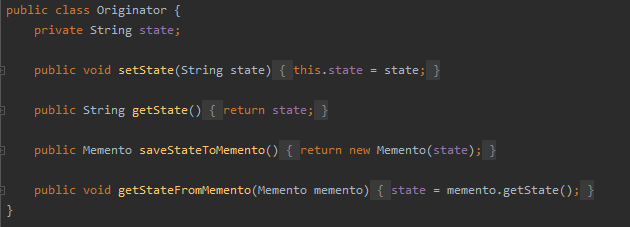


Figure : Originator class

The originator class used to set the state, and save it to Momento.

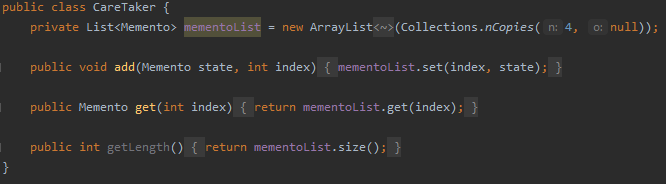


Figure : CareTaker class

The CareTaker class, stores a list of Memento objects. It can add and get the specific memento state by a given index.

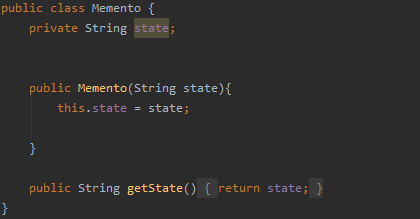


Figure : Memento class

The memento class stores the state of the object.

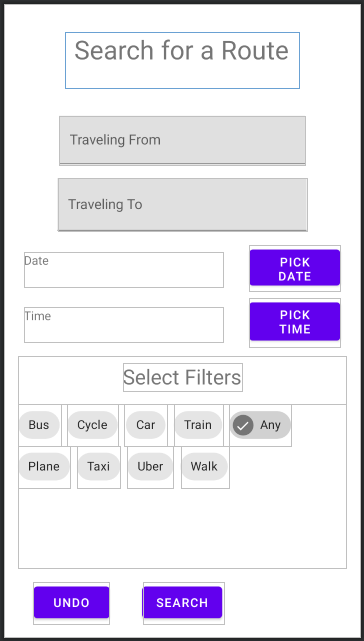


Figure : Search Fragment of the UI with the UNDO button on the bottom left side

## 4.7 Filter Pattern

Filter pattern was used for applying filters when the user wanted to generating routes from stations. On the android side the user can select the available filter that are there. We have a Criteria interface having a meetCriteria method which takes a list of type T as input then returns a subset of T who satisfy the filter criteria. Every filter criteria class must implement this interface.

We have implemented filters for And and Or Criteria, we also have all the criteria for the types of transport types.

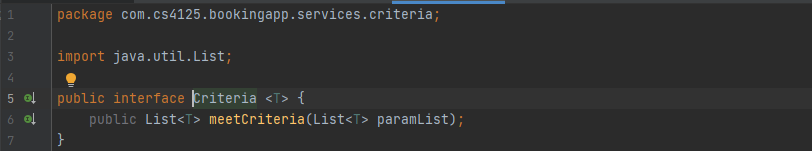


Figure : Criteria.java

Figure 31 is the interface used by all criterias.

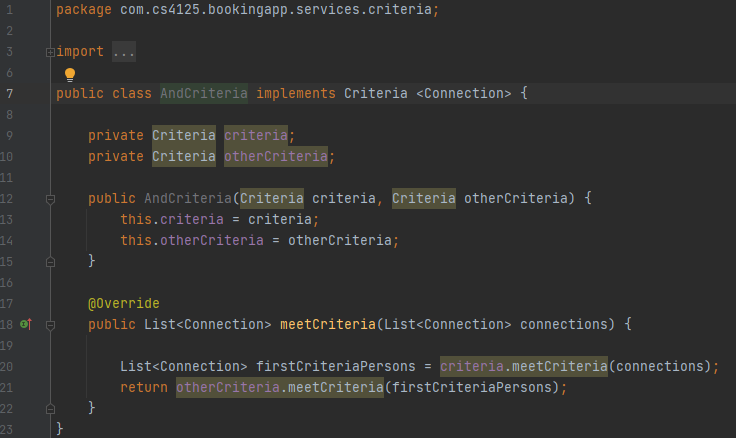


Figure : AndCriteria.java

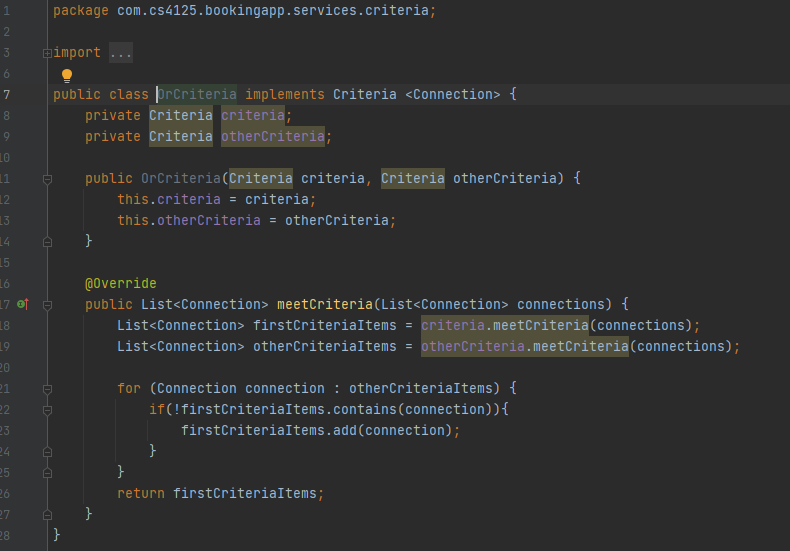


Figure : OrCriteria.java

The ‘and’ and ‘or’ criteria allow combining multiple criteria together for more complex filters.

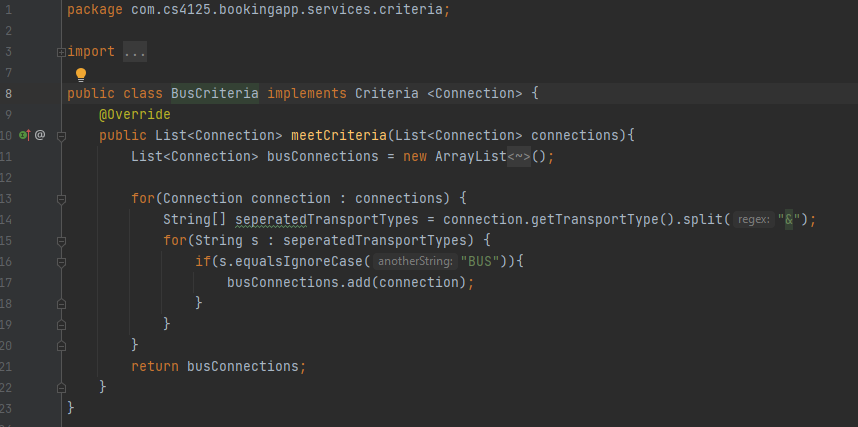


Figure : BusCriteria.java

Figure 34 shows an example of a BusCriteria in which it checks if a connection has bus as a valid transport type, if it does it will add that connection to the list if not it will be skipped, at the end it returns all connections with the desired transport types.

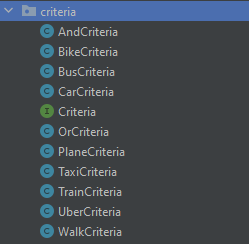


Figure : List of Filter/Criteria

## 4.8 Bridge Pattern

Bridge pattern is used to handle the payments in this project. We have our implementer interface which is our PaymentSystem.java class. We then have our concrete implementer which is our StripePaymentSystem.java, in this class we handle our payment using the stripe sandbox. We then create our abstract class which has the methods to make payments. Then we have the refined abstract class to implement these payment methods DebitCardPayment.java.

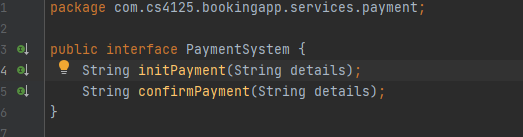


Figure : PaymentSystem.java

The PaymentSystem interface specifies that payment can be initialised and finalised/confirmed, allowing for other classes to specify the logic for performing these operations.



Figure : StripePayment.java

Figure 37 shows the StripePaymentSystem which defines how a payment is initialised and confirmed using Stripe’s payment processing.

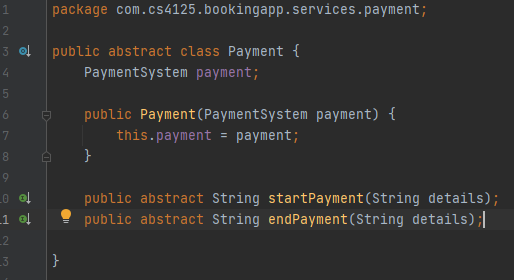


Figure : Payment.java

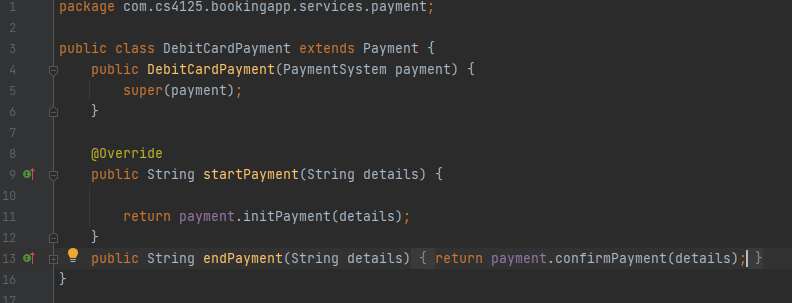


Figure : DebitCardPayment.java

Payment allows for different type of payments, currently we are focused on credit and debit cards, but this can be expanded in the future to support bank transfers, or other payment methods as required.

## 4.9 GitHub – Version Control

<https://github.com/Eoghan1232/CS4227_Project>

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. All CS4227 commits are from February and onwards.

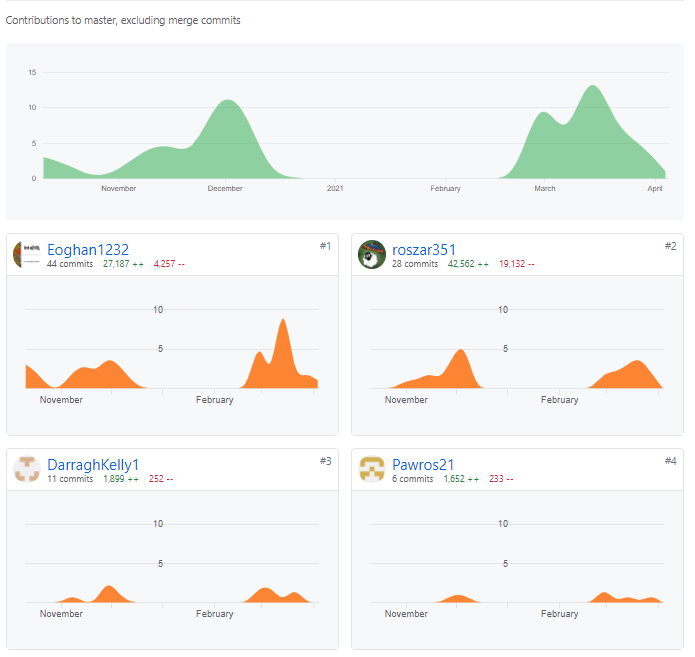


Figure : GitHub Contributors

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

# 5. Added Value

## 5.1 Jenkins

Jenkins is a free and open source automation server. For our project, we used it for building and testing our project throughout its implementation. The Jenkins server was setup on a Linux virtual machine, so it ran locally. A project is created once Jenkins is up and running, as you can see in Figure 41, it shows different runs and the stages in each run. For our project, it was broken up into four stages, cloning the GitHub repository, building the project using maven, running the tests, and finally generating the test results.

Jenkins can execute all tests using maven, if any tests failed, it would show in the stage as a red box. All tests currently pass so every stage is green.

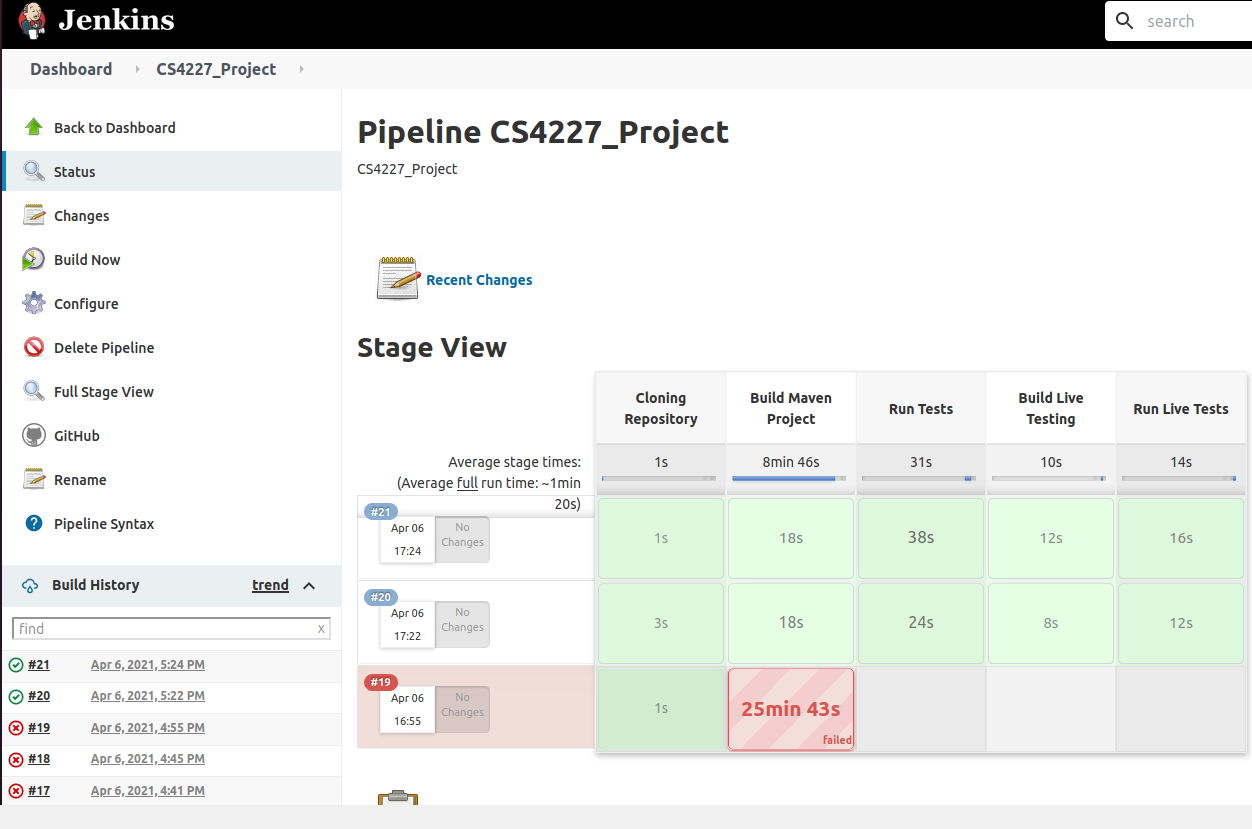


Figure : Jenkins

Figure 42 shows the pipeline script for our project, this is how Jenkins knows what to do. The pipeline is split into the different stages and Jenkins executes what’s inside of them. The post stage at the end gets executed no matter what, meaning a report will always be generated even if the project fails to build, etc.

All of our tests were set up to run automatically on Jenkins, a graph is generated showing the trend in results over each run, this is shown in Figure 43. All of our tests are currently all passing and you can see when new tests were added between run 15 and 16. If any tests failed, the graph would show it and this helped us make sure that the functionality still worked after each code commit.



Figure : Jenkins Pipeline Script

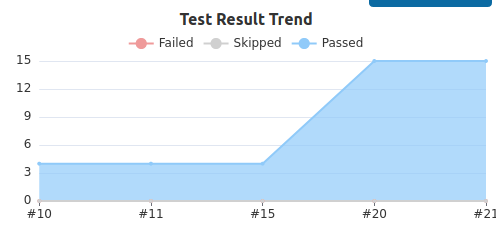


Figure : Jenkins Test Results Graph

## 5.2 Pathfinding Algorithm – Dijkstra’s

For our project, we implemented Dijkstra’s algorithm for our path finding algorithm. Dijkstra’s algorithm is an algorithm for finding the shortest path between nodes in a graph. For us, it calculated the shortest path between two given stations.

For our project, paths were interchangeable, meaning that you could travel from station 1 to station 2 and travel from station 2 to station 1.

As input, we took the start and end node (stations), and the list of all connections from the Database. Vertices and Edges were then created by iterating through all the connections. A Vertices list is created, which contained each vertex, their neighbouring vertex, and the distance between them.

The setup of the vertices and edges are shown in Figure 44, Figure 45, Figure 46. This is essentially to setup the nodes correctly to input into the Dijkstra’s algorithm.

The method returns the shortest path found between the start and end station, if there is no path, it will just return the start station. The classes for the Edge and Vertex are shown in Figure 47 and Figure 48.

Figure 49 shows how the shortest path from the starting node is calculated. Figure 50 shows how the path is returned from the end station to the start station.



Figure : Path Finding algorithm. Image 1

The for loop iterating through each connection is used to build the vertices (stations) uniquely.

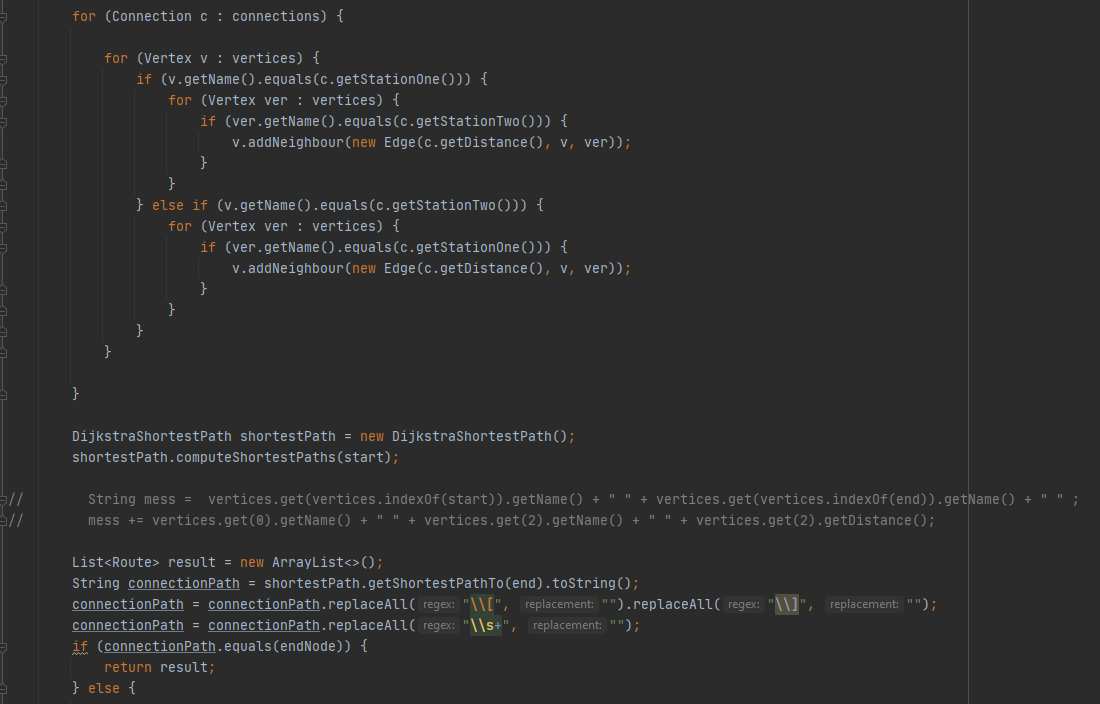


Figure : Path Finding Algorithm. Image 2

The for loop iterating through each vertex is setting up the connections, and the distance between the nodes. E.g. Node 1 can go to Node 2 and Node 2 can go to Node 1.

The line shortestPath.computeShortestPaths(start) computes all paths from the start to all other nodes. The later the shortestPath.getShortestPathTo(end) returns the shortest path from the start to the desired node. Finally, a list of Routes is setup and is returned.

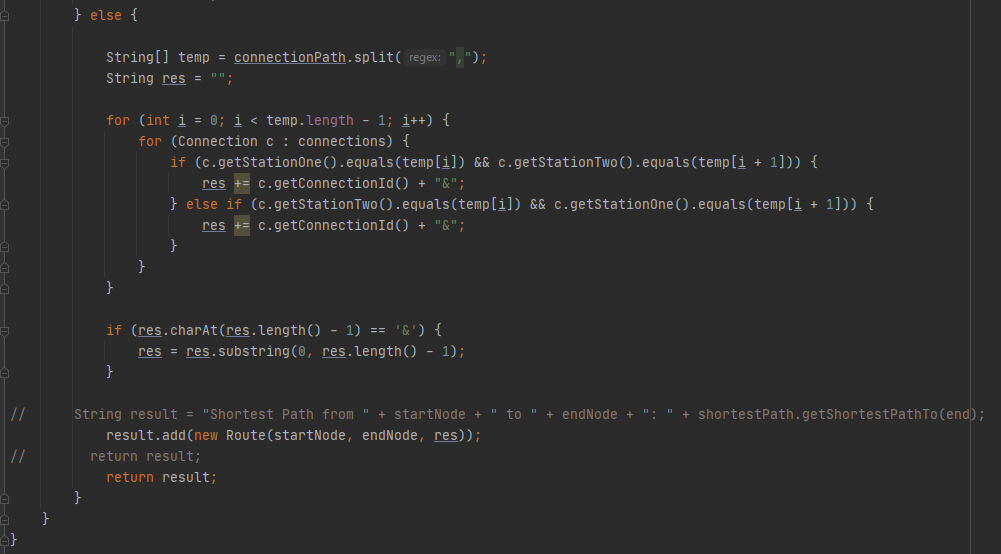


Figure : Path Finding Algorithm Image 3

The path is properly setup, the path is given as 1&2 meaning use connection 1 and 2. This is return as a Route.

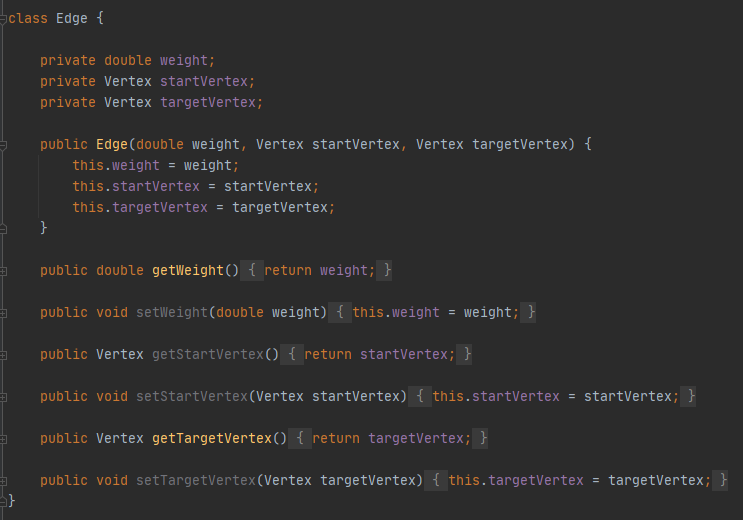


Figure : Path Finding Algorithm Edge Class

The Edge class used for the algorithm. The connection between 2 vertices.

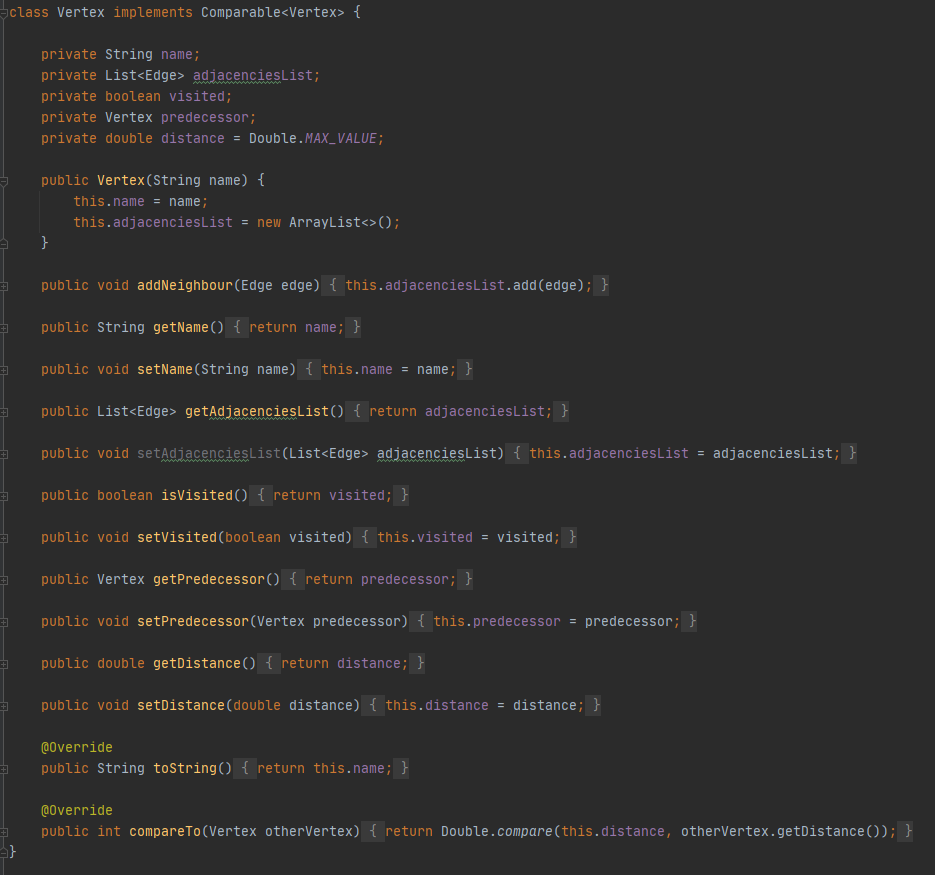


Figure : Path Finding Algorithm Vertex Class

The Vertex class used for the algorithm. Keeps track of Neighbours, if a node was visited, etc.

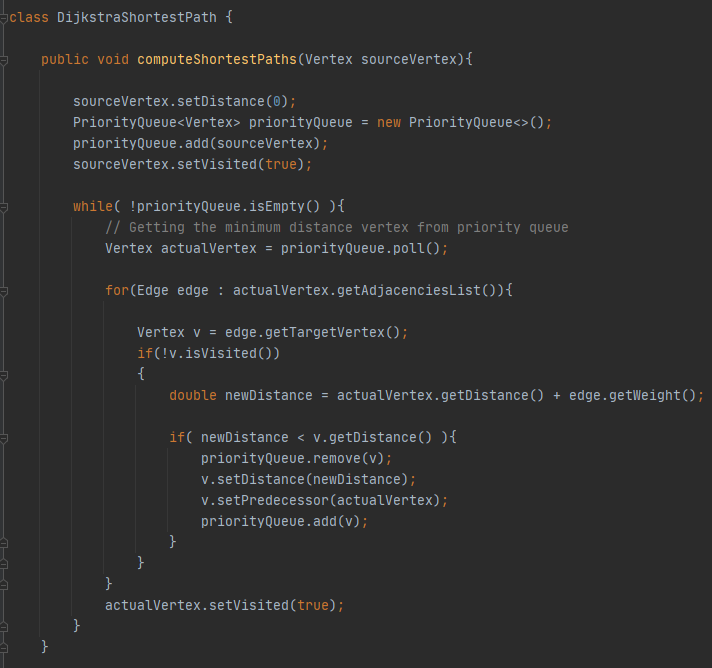


Figure : Path Finding Algorithm ComputeShortestPaths method

To compute the shortest path to all nodes.

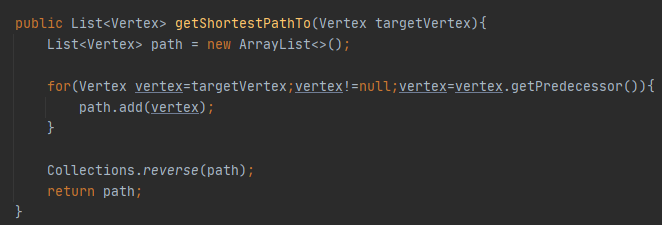


Figure : Path Finding Algorithm getShortTestPathTo method

Returns the specific shortest path, start to given end node.

## 5.3 Stripe (Payment Sandbox)

For this project we will be using stripe to process our payments when booking a ticket.

Stripe is a full stack payment processor which mean it can be a third-party payment processor as well as a payment gate. With this it means that Stripe can be used to process credit and debit cards.

Stripe also has a very high level of security for payments. They have been audited by a PCI-certified auditor and is certified to PCI Service Provider Level 1.

For payments in this project we used the bridge pattern and one of the class in this pattern is the StripePaymentSystem.java, in this class we have two major methods, the first one is to initialize the payment. The second method is to confirm the payment.



Figure : StripePayment.java

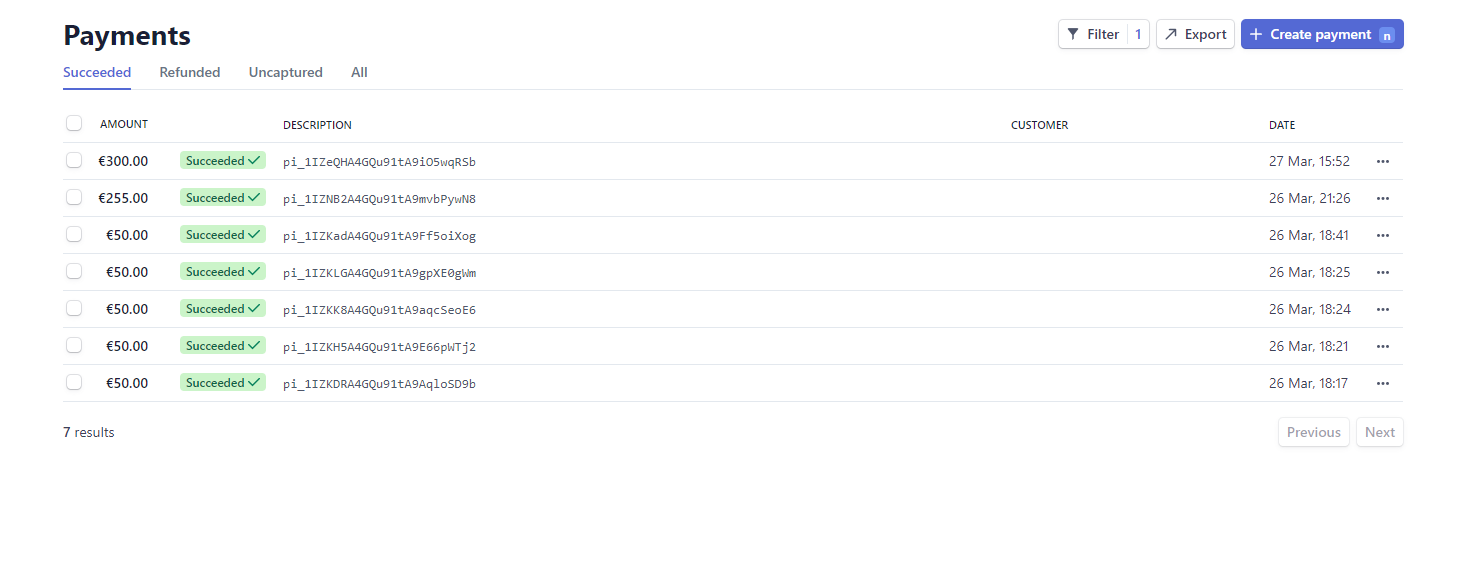


Figure : Payments in Stripe Dashboard

One a customer makes a payment we can this these reflected in the Stripe dashboard. Here we can view all of the transaction that have taken place. You can see what payments have succeeded, in progress or have failed. Here we can manually refund payments and send receipts.

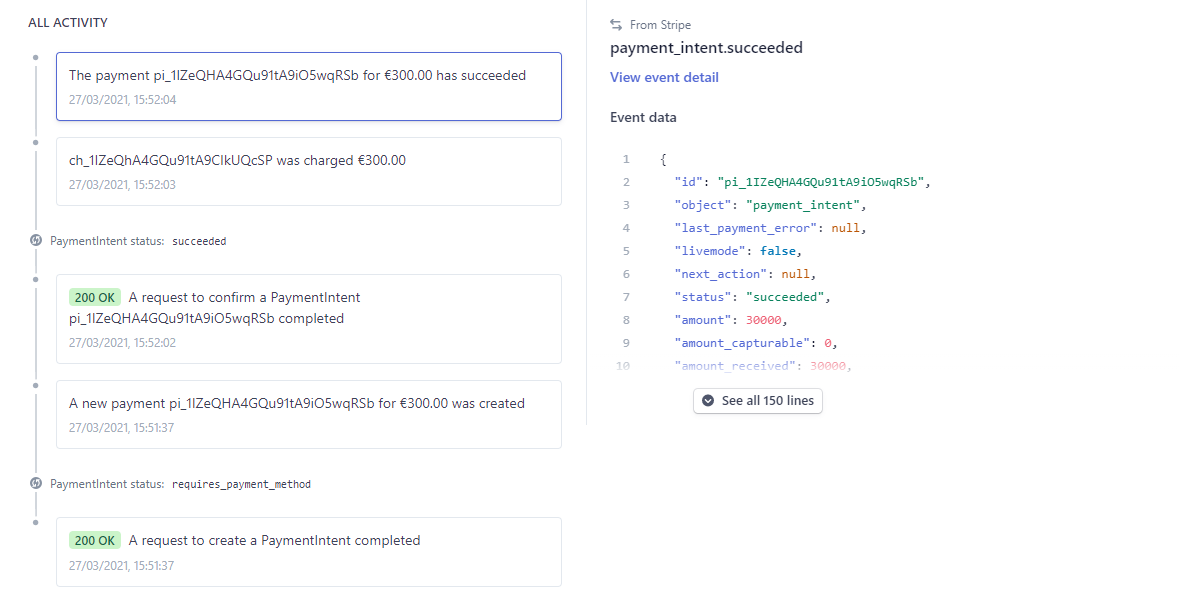


Figure : Stripe Webhooks

Stripe uses webhooks to be able to notify the app when an event happens in our account. These are very usefully for asynchronous events such as when a customer disputes a charge, recurring payments or when a customer confirms a payment.

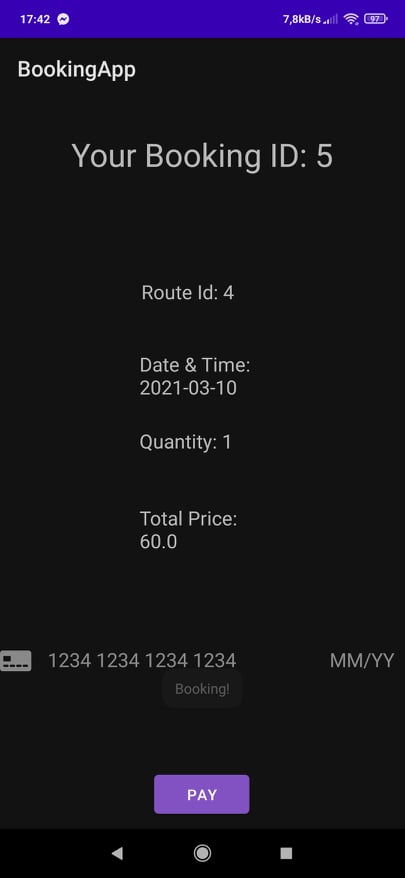
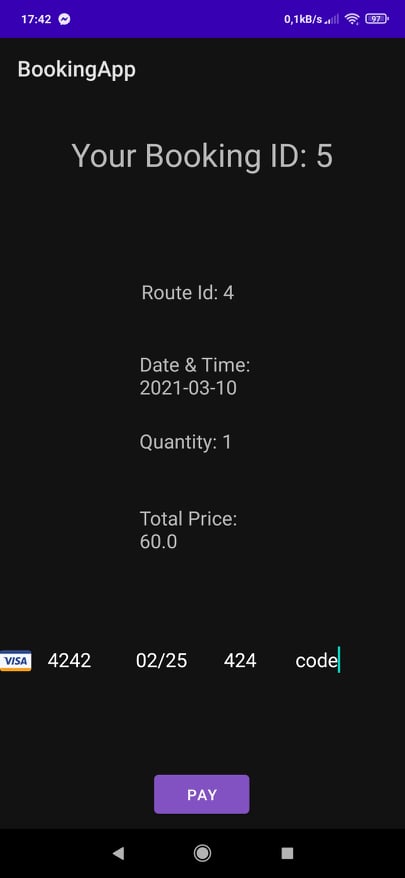


Figure : Card Details

Figure : Android CardWidget



Figure : BookResultFragment

On our android, we have a cardInputWidget on our booking\_result\_fragment that takes our inputted card information. We then start the payment process method once the booking screen is opened. We then have a method to observe the payment intent.

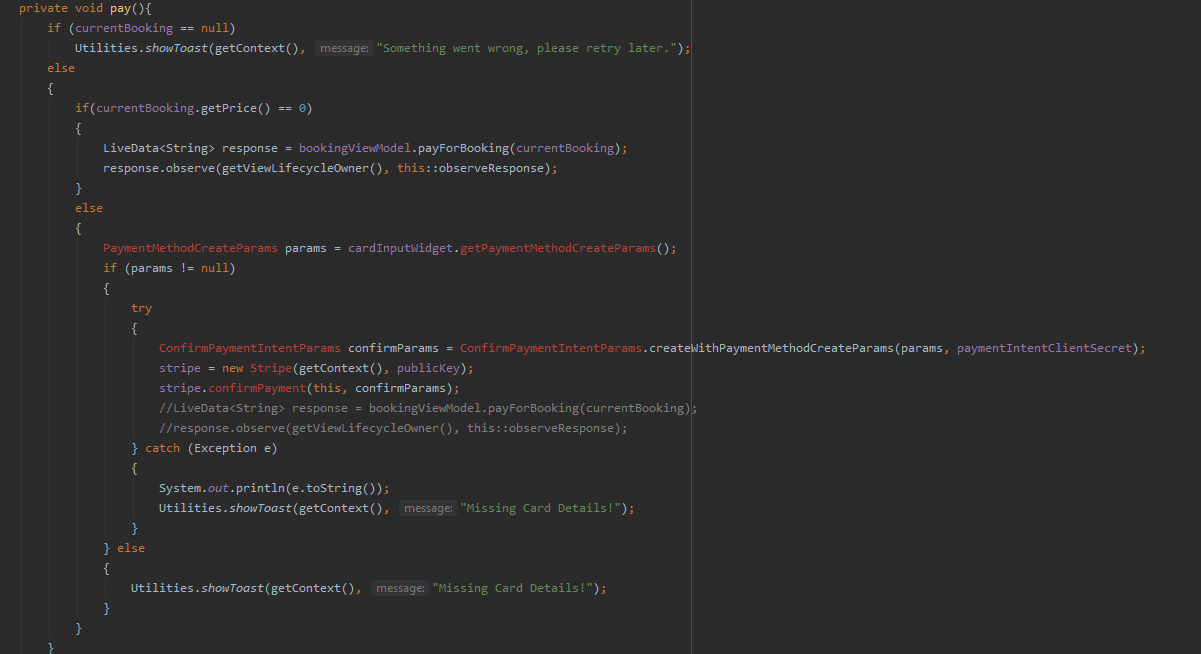


Figure : BookingResultFragment.java

With our pay method, once we have all the card information inputted into the card widget and press the button to pay, this method will then be able to finalize the payment.

## 5.4 CS4125 vs. CS4227

These Class Diagrams are quite large and might be best viewed in their respective .png files.

## Diagram, schematic Description automatically generated

Figure : UML Class Diagram CS4125

## 

Figure : Android Class Diagram CS4227

## 

Figure : Spring Boot Class Diagram CS4227

Current version of the project i.e. the CS4227 version differs from the old CS4125 version in many ways, the main difference is that for the old version routes were pre-defined, the user had to know the exact details for the route they were planning to book which not user friendly. In addition, the payment system was not fully implemented and was planned to be completely handled by our backend i.e. not relying on other payment processing providers. Both of these were changed, firstly the routes are no longer pre-defined and instead our system keeps track of different nodes or stations and connections that specify paths between these nodes, the distance and the available transport types, users now request a route to be generated between their desired starting and ending points. It also allows them to select filters i.e. what transport type they want to use. This approach is more user friendly and also allows for better journey planning for our users as they no longer require to know the full details of a journey, only their start point and their destination. Also payment is now done through an external payment processing service, to provide greater security for transactions and also for easier method of paying. Currently only Stripe is supported but can be easily expanded thanks to the use of Bridge pattern.

Another major addition is the use of Interceptor pattern which allows the backend to perform extra processing on a request which can be used to expand the system to perform more complex logic. Currently this is used to log requests which helps with analysing how the app is being used and what the most frequent requests are.

The discount system has also gone through changes, by changing it to use the Command pattern we were able to rework the discount system to support applying multiple discounts and also ability to expand the system to support more discount types in the future if required.

The GUI of the project has gone through small rework to improve the user experience, it is now easier to book a journey as instead of having to search for a route and then go back to book the route id, the app now allows to click on a result of a search to go straight into booking it.

The package structure remained very similar to previous iteration of the project from CS4125, but new packages were created for some of the new services created to support additional use cases and functionality.

# 6. Testing

## 6.1 Junit Testing

Junit is a unit testing framework for Java. It was used for the testing of our project. Junit allowed a Continuous Integration (CI) approach. This informed us whether a build or test failed, meaning something newly added has broken the project and needs to be investigated. Unit tests were created for the BookingController, DiscountController, PaymentController, UserController.

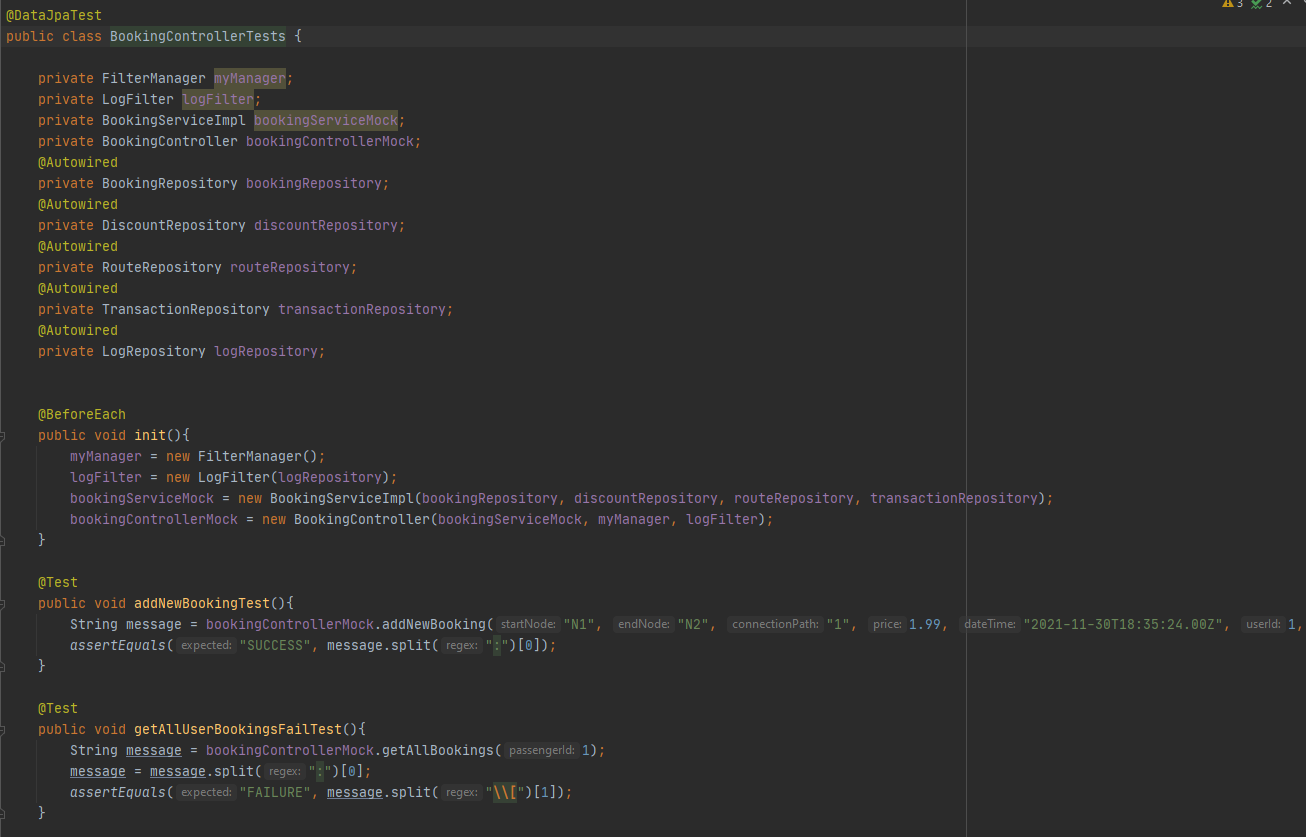
The @Autowired creates databases in memory, so the tests don’t affect the real database. It essentially creates an empty database, that we can perform different requests and test the results.

Figure : BookingController Sample of Tests

All tests setup to run on Jenkins automatically (c.f. 5.1 Jenkins) but are also ran when a maven clean install is executed.

Unit tests were implemented only for the UserController in the CS4125 project, before being expanded upon in this project.

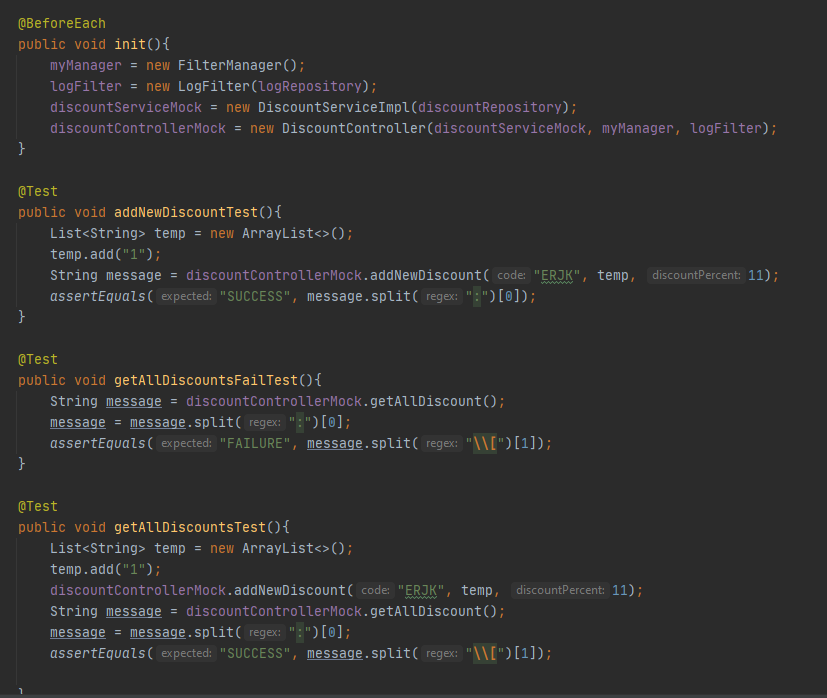


Figure : DiscountController Unit Tests

Some of our DiscountController test, to add a new discount, a positive and negative test to retrieve all discounts.

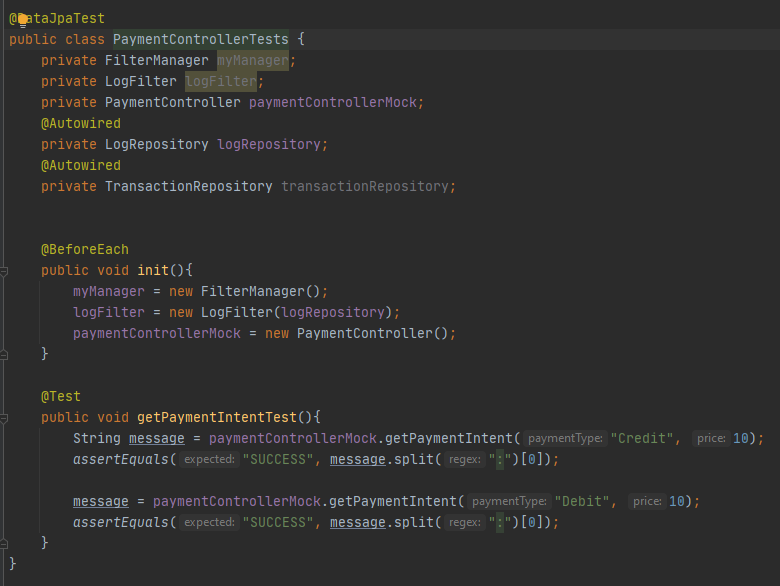


Figure : PaymentController Unit Tests

Some of out PaymentController tests, this is to test the Intent of the payments, if they’re valid or not.

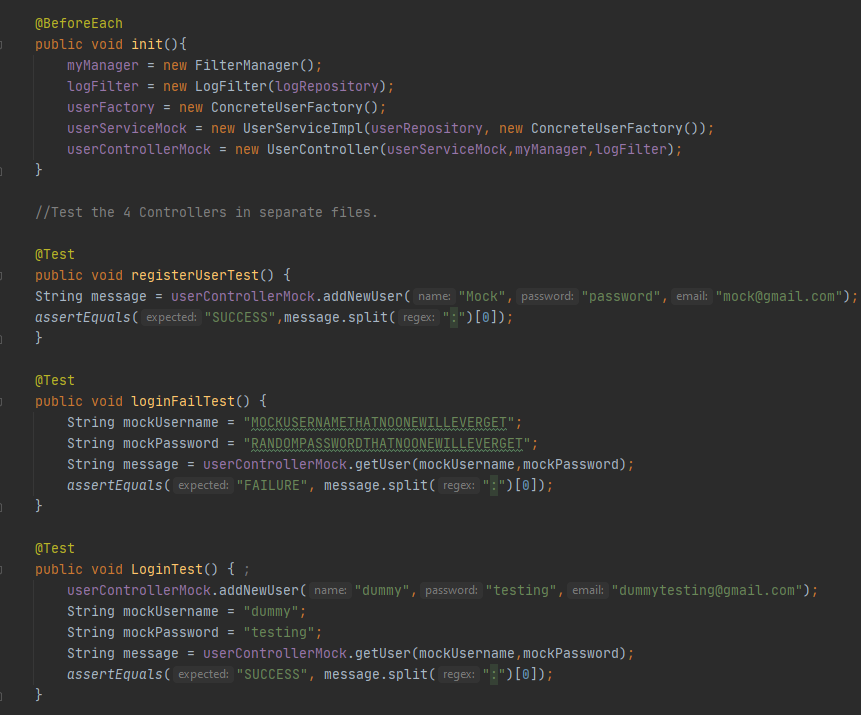


Figure : UserController Unit Tests

Some of our UserController tests, To register a new user, a positive and negative login test.

## 6.2 REST Testing

Representational state transfer (REST) is an architectural style used to interact with web server applications. It sends and receives responses directly to and from the web server allowing for direct access and speed especially in the testing scenario where no User Interface (UI) is required. The REST Testing was created as a separate maven project that uses Junit and REST Assured libraries to allow for test cases creation on the REST API. The tests included one of each basic commands being GET, POST and DELETE. Where each command is doing operation to get response, send a request to the web service or delete an entry from the service respectively. Examples of GET and POST commands are shown below.

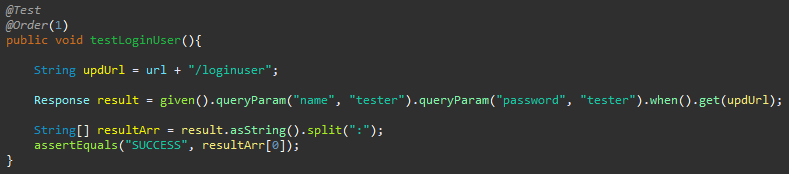


Figure : Test to Login User using GET command

A request gets built and is sent using Json. Essentially sends a CURL command to test the live client. This test is to login a user.

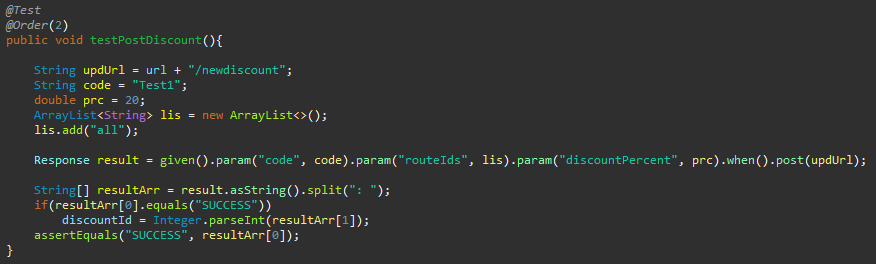


Figure : Test to create a new Discount using POST command

This test is used to create a new discount using the POST command.

# 7. Problems Encountered

## 7.1 Dependencies

The project was in total split into 3 separate projects. The android side, the spring boot side and the testing side. All the projects required multiple settings, dependencies and libraries imported and to work for all members of the group. That caused issues few times where for example initial spring boot dependencies had to have the correct versions to work with each other but that wasn’t clearly stated in any documentations. Similar problem arose when creating the testing project which used REST for example and Junit versions where both had to be correctly assigned in the pom file. Further that caused an issue where two people were working on two sets of test individually, one on the spring boot side and the other on the REST side and when combined in Jenkins the JDK version and Junit versions didn’t match and one had to be downgraded in the pom file without breaking any of the code.

## 7.2 Other path finding algorithms

There was a plan in motion to implement different path finding algorithms for example to search for all the paths. The issue with the path finding algorithms was that the database implementation wasn’t initially created to fit nodes and edges for destinations and routs in that sense. All that had to be implemented separately as initially the algorithm couldn’t be created from the raw data. That lead to the issue where applying different algorithms for the path finding and strategy pattern with the combination of the filters to be applied would be too expensive, considering the changes to be made in each class for the algorithm, to add with respect to the value it would contribute.

# 8. Evaluation and Critique

## 8.1 Critique of Admin User

Initially in this project there was an idea to implement an admin user. The said user would be able to handle administrator level features such as adding and deleting routes, adding and deleting discounts and some other features requiring this access if necessary in the further development. That idea never became a thing as it would require not only new functionality but also new fragments on the android side of the project to have a graphical interface for the created functionality. Currently our system operates in such a way that all new features have to be added or deleted using a curl command, that directly connects to the live service, or through manual addition to the database.

## 8.2 Critique of Graphical User Interface

The system is split between Spring Boot and Android applications. The later provides the necessary functionality to implement Graphical User Interface (GUI). This project includes a functional, fully tested and working GUI but some more work could be done on it. The work could include making the interface more neat, more appealing and add some new functionality like the one for admin user from section 8.1. To make the visual aspects of the project neater some rearranging would be required, e.g. on the search fragment which has a lot of options and buttons. To make the interface more appealing some more functional colour theme would be required alongside with greater button designs and option selection designs.

## 8.3 Critique of Command Pattern

Using the command pattern for applying the discounts has worked and functions efficiently, but it did come with one issue that was not resolved in the scope of this project. One of potential features that we wanted to implement was a loyalty point system, this system would allow to provide users to receive points when they use our app i.e. when they purchase tickets. These points then could be used to apply discounts for any future purchases. We have found that using the command pattern for such a system was not going to work or at least would require a lot of separate commands to handle the different parts of the system i.e. check current points, apply discount based on points, add new points etc. From our experience it seems this system would be best to be implemented as a separate service that would handle all of its logic itself, at current time we did not find which pattern would be best suited for such as system. If the project were to be continued this would definitely be one area of high priority to do research and find a solution suited for such a system.

## 8.4 Critique of Memento Pattern

The memento pattern was added on the android side to have the values of each input box saved on the search screen of the application and allow the user to go back to previous search option. This was done by introducing a undo button in a case a user would like to change search parameters ultimately looking for example for a different day etc. This use didn’t add much value to the project and the android APK already has the functionality to go back to the previous screen and have most of the inputs there saved while doing so. During further development of the project there would be a chance for a better location to implement that pattern but currently it isn’t of a great use.

## 8.5 Evaluation of Interceptor Pattern

The interceptor pattern implementation is one of the highlights of the application. It provides functionality to apply filters onto all services of the application. The filters come in the form of expandable list which can hold from one to as many as needed filters that would be ran in order. In each Service Implementation class the interceptor is handling the execution of the application based on the requests provided by the controller. The entire section is made of a switch statement where each case handles desired functionality based on the request issued by the controller. The use of the switch statement along with the request creation results in a great extensibility of this pattern where new functionality can be easily implemented by just making proper request and handling it in a new case of switch statement for respective Service Implementation.

## 8.6 Evaluation of Bridge Pattern

Implementation of the Bridge Pattern turned out to be a very effective way of supporting multiple different types of payment services(Stripe) and different payment types(Credit and Debit). In the current version of application user can pay both using a credit or debit card through Stripe. Stripe was implemented using the available service of Stripe Sandbox, therefore we can test our application without having to create real payment transaction and instead use the testing sandbox to test different type of transactions. The creation of the Bridge pattern also allows to add new payment options such as PayPal in the future without having to re-write any code, as we are able to add new payment services by creating new payment system class in our backend which then has common methods with other services. Another great benefit is that we can also very easily add any new payment types such as bank transfers in the future if required. Overall this pattern was implemented successfully and allows our project to be more extensible.

## 8.7 Evaluation of Jenkins

The addition of Jenkins pipeline enabled the application to perform automated test execution through a remote server. This provides functionality to create and execute Junit tests at any given time. The tests currently implemented introduce proof of concept for test automation of both spring boot and live services. Even with limited amount of test cases they allowed us to find and resolve some minor issues included in the development process. The way this service is implemented allows for easy extensibility of the tests that with a relatively little amount of work could cover the entire application both on the spring boot side and the live services.

# 9. Contributions of each team member

## 9.1 Spring Boot

|  |  |  |  |
| --- | --- | --- | --- |
| **Package** | **Class** | **Author** | **Lines of Code** |
| com.cs4125.bookingapp.controllers | BookingController | Eoghan/Pawel | 100 |
| com.cs4125.bookingapp.controllers | DiscountController | Eoghan/Pawel | 86 |
| com.cs4125.bookingapp.controllers | PaymentController | Eoghan/Pawel | 122 |
| com.cs4125.bookingapp.controllers | RouteController | Eoghan/Pawel | 114 |
| com.cs4125.bookingapp.controllers | UserController | Eoghan/Pawel | 48 |
| com.cs4125.bookingapp.model.entities | Booking | Eoghan/Damian | 98 |
| com.cs4125.bookingapp.model.entities | Connection | Damian | 75 |
| com.cs4125.bookingapp.model.entities | Discount | Eoghan/Damian | 67 |
| com.cs4125.bookingapp.model.entities | Log | Eoghan | 59 |
| com.cs4125.bookingapp.model.entities | Node | Damian | 53 |
| com.cs4125.bookingapp.model.entities | Route | Eoghan/Damian | 67 |
| com.cs4125.bookingapp.model.entities | TransactionRecord | Eoghan/Damian | 65 |
| com.cs4125.bookingapp.model.entities | User | Eoghan/Damian | 75 |
| com.cs4125.bookingapp.model.repositories | BookingRepository | Eoghan/Damian | 14 |
| com.cs4125.bookingapp.model.repositories | ConnectionRepository | Eoghan/Damian | 15 |
| com.cs4125.bookingapp.model.repositories | DiscountRepository | Eoghan/Damian | 12 |
| com.cs4125.bookingapp.model.repositories | LogRepository | Eoghan/Damian | 16 |
| com.cs4125.bookingapp.model.repositories | NodeRepository | Eoghan/Damian | 17 |
| com.cs4125.bookingapp.model.repositories | RouteRepository | Eoghan/Damian | 17 |
| com.cs4125.bookingapp.model.repositories | TransactionRepository | Eoghan/Damian | 12 |
| com.cs4125.bookingapp.model.repositories | UserRepository | Eoghan/Damian | 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.abstractFactory | AbstractCriteriaFactory | Damian | 8 |
| com.cs4125.bookingapp.services.abstractFactory | ConnectionCriteriaFactory | Damian | 44 |
| com.cs4125.bookingapp.services.abstractFactory | CriteriaFactoryProducer | Damian | 23 |
| com.cs4125.bookingapp.services.abstractFactory | RouteCriteriaFactory | Damian | 23 |
| com.cs4125.bookingapp.services.commandDiscount | ApplyAllDiscounts | Damian | 28 |
| com.cs4125.bookingapp.services.commandDiscount | ApplyDiscount | Damian | 20 |
| com.cs4125.bookingapp.services.commandDiscount | DiscountContext | Damian | 36 |
| com.cs4125.bookingapp.services.commandDiscount | DiscountInvoker | Damian | 17 |
| com.cs4125.bookingapp.services.commandDiscount | DiscountOperation | Damian | 6 |
| com.cs4125.bookingapp.services.criteria | AndCriteria | Darragh | 24 |
| com.cs4125.bookingapp.services.criteria | BikeCriteria | Darragh | 23 |
| com.cs4125.bookingapp.services.criteria | BusCriteria | Darragh | 25 |
| com.cs4125.bookingapp.services.criteria | CarCriteria | Darragh | 23 |
| com.cs4125.bookingapp.services.criteria | Criteria | Darragh | 8 |
| com.cs4125.bookingapp.services.criteria | OrCriteria | Darragh | 29 |
| com.cs4125.bookingapp.services.criteria | PlaneCriteria | Darragh | 23 |
| com.cs4125.bookingapp.services.criteria | TaxiCriteria | Darragh | 24 |
| com.cs4125.bookingapp.services.criteria | TrainCriteria | Darragh | 25 |
| com.cs4125.bookingapp.services.criteria | UberCriteria | Darragh | 25 |
| com.cs4125.bookingapp.services.criteria | WalkCriteria | Darragh | 25 |
| com.cs4125.bookingapp.services.interceptor | Filter | Eoghan/Pawel | 12 |
| com.cs4125.bookingapp.services.interceptor | FilterChain | Eoghan/Pawel | 34 |
| com.cs4125.bookingapp.services.interceptor | FilterManager | Eoghan/Pawel | 28 |
| com.cs4125.bookingapp.services.interceptor | LogFilter | Eoghan/Pawel | 34 |
| com.cs4125.bookingapp.services.interceptor | Target | Eoghan/Pawel | 12 |
| com.cs4125.bookingapp.services.pathFinding | PathFindingContext | Eoghan | 30 |
| com.cs4125.bookingapp.services.pathFinding | ShortestPathStrategy | Eoghan | 87 |
| com.cs4125.bookingapp.services.pathFinding | DijkstraShortestPath | Eoghan | 45 |
| com.cs4125.bookingapp.services.pathFinding | Edge | Eoghan | 36 |
| com.cs4125.bookingapp.services.pathFinding | Vertex | Eoghan | 67 |
| com.cs4125.bookingapp.services.pathFinding | Strategy | Eoghan | 13 |
| com.cs4125.bookingapp.services.payment | CreditCardPayment | Darragh | 17 |
| com.cs4125.bookingapp.services.payment | DebitCardPayment | Darragh | 17 |
| com.cs4125.bookingapp.services.payment | Payment | Darragh | 14 |
| com.cs4125.bookingapp.services.payment | PaymentSystem | Darragh | 7 |
| com.cs4125.bookingapp.services.payment | PaypalPaymentSystem | Darragh | 16 |
| com.cs4125.bookingapp.services.payment | StripePaymentSystem | Darragh/Damian | 85 |
| com.cs4125.bookingapp.services | BookingService | Damian | 16 |
| com.cs4125.bookingapp.services | BookingServiceImpl | ALL | 274 |
| com.cs4125.bookingapp.services | DiscountService | Damian | 16 |
| com.cs4125.bookingapp.services | DiscountServiceImpl | ALL | 179 |
| com.cs4125.bookingapp.services | EncryptionService | Damian | 9 |
| com.cs4125.bookingapp.services | EncryptionServiceImpl | Damian | 16 |
| com.cs4125.bookingapp.services | PriceCalculation | Damian | 132 |
| com.cs4125.bookingapp.services | RouteService | Damian | 21 |
| com.cs4125.bookingapp.services | RouteServiceImpl | ALL | 295 |
| com.cs4125.bookingapp.services | TransactionContext | Damian | 58 |
| com.cs4125.bookingapp.services | TransactionRecordCancelledState | Damian | 23 |
| com.cs4125.bookingapp.services | TransactionRecordCompletedState | Damian | 31 |
| com.cs4125.bookingapp.services | TransactionRecordInitialState | Damian | 32 |
| com.cs4125.bookingapp.services | TransactionRecordInProgressState | Damian | 36 |
| com.cs4125.bookingapp.services | TransactionRecordState | Damian | 12 |
| com.cs4125.bookingapp.services | UserService | Damian | 10 |
| com.cs4125.bookingapp.services | UserServiceImpl | ALL | 89 |
| com.cs4125.bookingapp | BasicController | Eoghan | 154 |
| com.cs4125.bookingapp | BookingApplication | Eoghan | 17 |
| com.cs4125.bookingapp.controller | BookingControllerTests | Eoghan | 64 |
| com.cs4125.bookingapp.controller | DiscountControllerTests | Eoghan | 67 |
| com.cs4125.bookingapp.controller | PaymentControllerTests | Eoghan | 43 |
| com.cs4125.bookingapp.controller | UserControllerTests | Eoghan | 68 |
| com.cs4125.bookingapp | BookingApplicationTests | Eoghan | 14 |

## 9.2 Android Application

|  |  |  |  |
| --- | --- | --- | --- |
| **Package** | **Class** | **Author** | **Lines of Code** |
| com.cs4125.bookingapp.entities | Booking | Damian | 124 |
| com.cs4125.bookingapp.entities | Discount | Damian | 87 |
| com.cs4125.bookingapp.entities | Route | Damian | 124 |
| com.cs4125.bookingapp.entities | TransactionRecord | Damian | 67 |
| com.cs4125.bookingapp.entities | TransactionStatus | Damian | 7 |
| com.cs4125.bookingapp.entities | User | Damian | 107 |
| com.cs4125.bookingapp.entities | UserType | Damian | 7 |
| com.cs4125.bookingapp.memento | CareTaker | Pawel | 22 |
| com.cs4125.bookingapp.memento | Memento | Pawel | 18 |
| com.cs4125.bookingapp.memento | Originator | Pawel | 24 |
| com.cs4125.bookingapp.memento | State | Eoghan/Pawel | 101 |
| com.cs4125.bookingapp.repositories | BookingRepository | Darragh/Damian | 16 |
| com.cs4125.bookingapp.repositories | BookingRepositoryCacheProxy | Damian | 117 |
| com.cs4125.bookingapp.repositories | BookingRepositoryImpl | Darragh/Damian | 168 |
| com.cs4125.bookingapp.repositories | DiscountRepository | Darragh/Damian | 14 |
| com.cs4125.bookingapp.repositories | DiscountRepositoryCacheProxy | Damian | 138 |
| com.cs4125.bookingapp.repositories | DiscountRepositoryImpl | Darragh/Damian | 194 |
| com.cs4125.bookingapp.repositories | PaymentRepository | Darragh/Damian | 7 |
| com.cs4125.bookingapp.repositories | PaymentRepositoryImpl | Darragh/Damian | 48 |
| com.cs4125.bookingapp.repositories | ResultCallback | Damian | 8 |
| com.cs4125.bookingapp.repositories | RouteRepository | Darragh/Damian | 17 |
| com.cs4125.bookingapp.repositories | RouteRepositoryCacheProxy | Damian | 233 |
| com.cs4125.bookingapp.repositories | RouteRepositoryImpl | Darragh/Damian | 249 |
| com.cs4125.bookingapp.repositories | UserRepository | Darragh/Damian | 13 |
| com.cs4125.bookingapp.repositories | UserRepositoryImpl | Darragh/Damian | 80 |
| com.cs4125.bookingapp.ui.main | AdminFragment | Pawel/Damian | 40 |
| com.cs4125.bookingapp.ui.main | AdminViewModel | Pawel/Damian | 250 |
| com.cs4125.bookingapp.ui.main | BookingFragment | Pawel/Damian | 153 |
| com.cs4125.bookingapp.ui.main | BookingResultFragment | Pawel/Damian | 218 |
| com.cs4125.bookingapp.ui.main | BookingViewModel | Pawel/Damian | 146 |
| com.cs4125.bookingapp.ui.main | DatePickerFragment | Pawel/Damian | 33 |
| com.cs4125.bookingapp.ui.main | LoginFragment | Pawel/Damian | 101 |
| com.cs4125.bookingapp.ui.main | LoginViewModel | Pawel/Damian | 57 |
| com.cs4125.bookingapp.ui.main | MainActivity | Pawel/Damian | 11 |
| com.cs4125.bookingapp.ui.main | MainFragment | Pawel/Damian | 71 |
| com.cs4125.bookingapp.ui.main | MainViewModel | Pawel/Damian | 8 |
| com.cs4125.bookingapp.ui.main | RegisterFragment | Pawel/Damian | 105 |
| com.cs4125.bookingapp.ui.main | RegisterViewModel | Pawel/Damian | 59 |
| com.cs4125.bookingapp.ui.main | RouteAdapter | Pawel/Damian | 111 |
| com.cs4125.bookingapp.ui.main | SearchFragment | Eoghan/Pawel/Damian | 285 |
| com.cs4125.bookingapp.ui.main | SearchResultFragment | Pawel/Damian | 105 |
| com.cs4125.bookingapp.ui.main | SearchViewModel | Pawel/Damian | 138 |
| com.cs4125.bookingapp.ui.main | TimePickerFragment | Pawel/Damian | 37 |
| com.cs4125.bookingapp.ui.main | Utilities | Pawel/Damian | 12 |
| com.cs4125.bookingapp.web | RetrofitClientInstance | Damian | 43 |
| com.cs4125.bookingapp.web | SpringRetrofitService | Damian | 106 |

## 9.3 Rest Assured Testing

|  |  |  |  |
| --- | --- | --- | --- |
| **Package** | **Class** | **Author** | **Lines of Code** |
| unitTesting | Testing | Pawel | 90 |

## 9.4 Team Member Contribution

|  |  |
| --- | --- |
| **Team Member** | **Lines Contributed** |
| Eoghan Russell | 1,439 |
| Damian Skrzypek | 3,199 |
| Darragh Kelly | 730 |
| Pawel Ostach | 594 |

## 9.5 Total Code Developed

A total of ~8070 lines of code were written for this project. ~4,585 were from CS4125, ~4,000 new lines were committed in CS4227.

# References

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].

*Design Patterns in Java Tutorial - Tutorialspoint*, available: <https://www.tutorialspoint.com/design_pattern/index.htm> [accessed 27/03].

*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].

*Security at Stripe*, available: <https://stripe.com/docs/security/stripe> [accessed 04/04].

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

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