CSCI 5308- Shipogle - Team 9 (Dev) and Team 27 (Client)

This document provides all the information related to the development of the application, from dependencies, use cases, smells detected and refactored to the references used to complete this project. Steps to deploy or run the application are also added and touched upon. Jira board link along with test cases status has also been added.

1. Dependencies

Backend specific dependencies: These dependencies are needed in the Shipogle app for database management, building web applications, implementing security, sending emails, user authentication, live reloading during development, connecting to a MySQL database, performing unit testing, mocking objects for testing, supporting real-time communication, and using an in-memory database for unit testing.

Dependency	**Why We Need It**
spring-boot-starter-data-jpa	To use Java Persistence API (JPA) for database management
spring-boot-starter-web	To build web applications with Spring MVC framework
spring-boot-starter-security	To implement security features in the application
spring-boot-starter-mail	To send emails from the application
jjwt-api	To use JSON Web Tokens (JWT) for user authentication

jjwt-impl	To implement the jjwt-api
jjwt-jackson	To serialize and deserialize JSON with jjwt
jjwt	To create and parse JWTs
spring-boot-devtools	To support development with live reload
mysql-connector-j	To connect to a MySQL database
spring-boot-starter-test	To perform unit testing for the application
junit	To perform unit testing for the application
mockito-all	To use Mockito framework for unit testing
spring-boot-starter-websocket	To support real-time communication between server and client
h2	To use an in-memory database for unit testing

Frontend specific dependencies:These dependencies are used for component building in React, styles in CSS, for integration with the backend APIs, creation of Cookies for login sessions,

routing of react components and pages, form validation and for web socket connections and functionalities.

Dependency	Description
@date-io/date-fns	Date abstraction layer for date-fns, a date utility library
@date-io/dayjs	Date abstraction layer for Day.js, a minimalist JavaScript library for date parsing and formatting
@emotion/react	A library for styling components with CSS-in-JS using Emotion in React
@emotion/styled	A library for creating styled React components with Emotion
@mui/icons-material	Material Design icons for use with Material-UI
@mui/material	A popular React UI framework implementing Material Design
@mui/x-date-pickers	Date and time pickers for Material-UI
@testing-library/jest-dom	Custom jest matchers for testing DOM nodes with jest
@testing-library/react	A lightweight library for testing React components
@testing-library/user-event	A library for simulating user events in React testing
axios	A promise-based HTTP client for JavaScript

date-fns	A modern JavaScript date utility library
dayjs	A minimalist JavaScript library for date parsing and formatting
js-cookie	A simple, lightweight JavaScript API for handling browser cookies
react	A JavaScript library for building user interfaces
react-dom	React package for working with the DOM
react-hook-form	A performant, flexible and extensible forms library for React
react-router-dom	A collection of navigational components for web applications built with React
react-scripts	Scripts and configurations for Create React App
socket.io-client	Client-side library for real-time bidirectional communication using WebSocket
web-vitals	A library for measuring and reporting on web app performance
websocket	A library for WebSocket, a protocol providing full-duplex communication channels over a single TCP

2. Commands

The commands to install/configure these dependencies would depend on the build tool being used for the project, such as Maven.

To include these dependencies in a Maven project, you can add them to the pom.xml file as follows:

Note: The pom.xml is already present in the code base

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-mail</artifactId>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-api</artifactId>
        <version>0.11.2</version>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-impl</artifactId>
        <version>0.11.2</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
```

```
  <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.2</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.11.2</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.10.19</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
```

```
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

To install the dependencies in the project, you can run the following command:

```
mvn clean install
```

Detailed instructions to build and deploy the app from scratch:

1. Install the required tools:
    - Java Development Kit (JDK) version 11 or higher
    - Apache Maven build tool version 3.6 or higher
    - MySQL database server version 5.7 or higher

2. Clone the Shipogle app repository from GitHub using the following command in your terminal or command prompt:
    - git clone https://git.cs.dal.ca/courses/2023-winter/csci-5308/group09.git

3. Set up the MySQL database by creating a new database and a user with privileges to access it. You can use a tool like phpMyAdmin or the MySQL command line tool to do this.

4. Configure the Shipogle app to use the MySQL database by editing the application.properties file located in the src/main/resources directory. Replace the placeholders <database-name>, <username>, and <password> with your database name, username, and password respectively:
        i. spring.datasource.url=jdbc:mysql://db-5308.cs.dal.ca/CSCI5308_9_DEVINT

ii. spring.datasource.username=CSCI5308_9_DEVINT_USER
iii. spring.datasource.password=ahJiR0boh5

5. Build the Shipogle app using Maven by running the following command in your terminal or command prompt from the root directory of the project:
    i. `mvn clean package`
    ii. This will generate a JAR file in the target directory.

6. Run the Shipogle app by executing the following command in your terminal or command prompt from the root directory of the project:
    i. java -jar target/Shipogle.jar

7. Access the Shipogle app by opening a web browser and navigating to http://localhost:8080.

Instructions to deploy app on server(Commands as well)

1. The server where you want to deploy the app, you need to install JDK/JRE version 11 or higher there.

2. You have to use the JAR file generated in previous steps; you have to move that JAR file to the web server. To move a file to a web server, you will need authentication credentials for that particular user on that server.

3. If you are using Linux or MacOS, you can transfer the file to the server using the scp command. After entering the command, it will ask for credentials for the user you are using to connect to the web server. Here, “csci5308vm9” is the username and “csci5308vm9.research.cs.dal.ca” is host_url and “backend/” is path where we want to transfer the file.
    i. scp ./Shipogle.jar csci5308vm9@csci5308vm9.research.cs.dal.ca:backend/

4. After transferring the file to the server, you can run the project JAR file on the server with the ssh command. Here, “csci5308vm9” is the username and “csci5308vm9.research.cs.dal.ca” is host_url and “java -jar ~/backend/Shipogle.jar” is command to run the application on the server.
    i. ssh csci5308vm9@csci5308vm9.research.cs.dal.ca "java -jar ~/backend/Shipogle.jar"

5. After running these commands successfully, you can check the running application by going to: http://csci5308vm9.research.cs.dal.ca:8080.
    i. Here, replace csci5308vm9.research.cs.dal.ca with your server's url.

3. ALL USAGE SCENARIOS

Note: Please refer to the GIFs folder to get a better understanding about the use cases mentioned below and follow the UI for the application feature.

3.1. Registration with Shipogle

Use Case: Registration of a Driver/Sender on Shipogle App

Pre-conditions:

1. Users have downloaded and installed the Shipogle app on their device.
2. User has a valid email address and Government ID number.
3. User has allowed location permission on the app.

Flow of events:

1. User launches the Shipogle app and clicks on the 'Register' or ‘Sign up’ button.
2. Users are directed to the registration page where they are prompted to enter their basic details such as name, address, location, and Government ID number.
3. User enters the required details and clicks on the 'Verify' button.
4. The Verification System checks the entered details and shows verifies(dummy implementation)
5. Then the user is prompted to enter their desired username and password.
6. User enters their desired username and password and clicks on the 'Submit' button.
7. Shipogle App registers the User as a Driver/Sender and sends a verification email to the registered email address.
8. User receives the verification email and clicks on the verification link to complete the verification process.
9. Users can now login to the Shipogle app using their registered username and password.

Post-conditions:

1. User is successfully registered as a Driver/Sender on the Shipogle app.
2. Users receive a verification email to their registered email address.

Users can now login to the Shipogle app using their registered username and password.

3.2. Login of a driver/sender on Shipogle app

Pre-conditions:

- User has a verified email address
- User has valid login credentials (username and password)

Flow of events:

1. User opens the Shipogle app on their device
2. User is directed to the login page
3. User enters their registered username and password
4. User clicks on the "Submit" button to initiate the login process
5. The system verifies the entered credentials
6. If the entered credentials are correct, the system authenticates the user and directs them to the search page
7. If the entered credentials are incorrect, the system displays an error message and prompts the user to re-enter their credentials
8. Once the user is successfully authenticated, they can access the features available to them in the app

Post-conditions:

- User is successfully logged in to the app
- User is directed to the search page
- User can access the features available to them in the app

3.3. Search Dashboard on Shipogle App

Pre-conditions: The user has opened the Shipogle app and is on the search dashboard page.

Workflow:

1. The user selects the source city from the dropdown list.
2. The user selects the destination city from the dropdown list.
3. The user enters the maximum package weight that can be carried.
4. The user enters the radius within which they want to search for rides.
5. The user enters the pickup date and drop-off date for the ride.
6. The user enters the maximum price they are willing to pay for the ride.

7. The user clicks on the search button to find the available rides matching the search criteria.
8. The app shows a list of rides available for the selected route, along with the details of the driver, the vehicle type, and the price for the ride.
9. The user can click on the map view button to see the rides available on a map view. Note that this option may not be available for all cities due to the security issues with Google Maps API places API.

Note: The Map view feature, which uses the Google Maps Places API to display interactive maps and geolocation features, cannot be accessed on a virtual machine (VM) due to a requirement for a secure connection using HTTPS protocol. In the absence of an SSL certificate, the HTTP protocol is used for communication between the client and server, which is not a secure method of data transfer. Therefore, the Google Maps API requests are blocked by the browser when made from a VM with an HTTP protocol and no SSL certificate. However, the Map view feature works smoothly and seamlessly when accessed from a local machine where security issues are not a concern.

10. The user can click on a ride to see more details about the ride, such as the exact pickup and drop-off locations, the time of departure and arrival, and any special instructions from the driver.
11. The app shows the booking details to the user, including the pickup and drop-off locations, the time of departure and arrival, the driver's details, and the total fare for the ride.

Post-conditions: The user has searched for a ride and booked a ride through the Shipogle app.

3.4. Booking a ride with a user (Accepted Ride) on Shipogle app

Prerequisite:

- User has searched and selected a driver to book a ride with

Workflow:

1. User selects the driver to book a ride with from the search results.
2. User is directed to the driver's ride details page where he can view the route between the source and destination cities, the driver's ride details, and pricing details.
3. User confirms the ride by clicking on the 'Book Ride' button and selects a suitable pickup and drop-off location.
4. The driver receives a notification on his app regarding the ride request.

5. Driver accepts the ride request and the sender receives a notification that the ride has been accepted.
6. Sender receives the driver's contact information and can contact the driver through message chat.
7. The sender has the option to cancel the order if he decides to not go through with the ride.

Post-Conditions:

1. The sender's ride is confirmed and the driver is notified.
2. The sender can pay for the ride and the order is created.

3.5. Canceling a Ride with a User on Shipogle App

Prerequisite:

- The user has requested a ride with the driver
- The driver is logged in to the Shipogle app and has the ride request in their delivery requests

Workflow:

1. Driver opens the Shipogle app and navigates to the "Delivery Requests" section
2. Driver finds the ride request they want to cancel and selects it
3. Driver clicks on the "Cancel" button.
4. Shipogle app sends a notification to the user that their ride request has been cancelled

Post-conditions:

- The ride request is cancelled and removed from the driver's delivery requests
- The user receives a notification that their ride request has been cancelled.

3.6. Negotiation for a Price (Using Chat-bot) on Shipogle App

Prerequisite:

The driver has accepted the ride for the sender.

Workflow:

1. Sender receives a notification that the driver has accepted the ride and he can now negotiate the price.
2. Sender types in a message and sends it to the driver.
3. The driver receives the message and responds to the sender's message with a proposed price.
4. This process continues until both parties agree on a price.
5. Once both parties agree on the price, they can choose to pay at delivery or online.

Post Conditions:

1. Sender and driver can now proceed with the agreed-upon price.
2. Sender can pay for the ride using the payment link in Orders.

3.7. Starting a Ride with OTP on Shipogle App

Prerequisite:

- The sender has booked a ride with a driver.
- The driver has accepted the ride request.
- The sender has shared the pick-up code with the driver.

Workflow:

1. The driver arrives at the pick-up location.
2. The sender shares the pick-up code with the driver.
3. The driver enters the pick-up code in the Shipogle app.
4. The app verifies the pick-up code and provides an OTP (One Time Password) to the driver.
5. The driver enters the OTP in the app to confirm the start of the ride.
6. The app records the start time of the ride and notifies the sender that the ride has started.
7. The driver picks up the packages from the sender's location.(offline)

Post Conditions:

- The sender is notified that the ride has started and can track the driver's location and delivery progress.

- The sender can track the driver's location and delivery progress in real-time using the app.

3.8. Ending a ride (with OTP) on Shipogle App

Prerequisite:

- The driver has started the ride and the sender has provided the drop off code to the driver.

Workflow:

1. The driver enters the drop off code provided by the sender.
2. The app verifies the code and prompts the driver to confirm the end of delivery.
3. The driver confirms the end of delivery by clicking on the "End Delivery" button.
4. The app verifies the OTP and marks the delivery as completed.
5. The app prompts the sender to rate and provide feedback on the driver's service.

Post Conditions:

- The completed delivery is shown in the sender's order history.
- The sender can rate the driver and provide feedback on their service.
- The driver's performance metrics are updated based on the sender's feedback.
- The sender can raise any issues related to the delivery through the app's support feature.

3.9. Paying for a Ride and Receiving Receipt on Shipogle App

Prerequisite:

- The sender has requested a ride and the driver has accepted it.

Workflow:

1. The sender goes to his orders page on the Shipogle app.
2. The sender selects the order he wants to pay for and clicks on the "Pay" button.
3. The app takes the sender to the payments page where he enters his card details and clicks on "Submit".

4. The app processes the payment.
5. The sender's payment is confirmed and he receives a digital receipt in html format.
6. The sender can view and download the receipt for his records.

Post-conditions:

- The payment is received by the driver and the sender is not able to see the "Pay" button anymore for that order.
- The sender's PDF receipt is downloaded and he can view or save it for his records.

3.10. Canceling the ride and Refund on Shipogle app:

Prerequisite: The user has paid for the ride.

Workflow:

1. The sender goes to his orders and selects the order he wants to cancel.
2. He clicks on the cancel button and a message pops up, asking him to confirm the cancellation.
3. The sender confirms the cancellation and selects the reason for cancellation.
4. The app calculates the refund amount and displays it to the sender.
5. The sender receives a message that the refund is processed.

Post Conditions:

1. The ride is canceled.
2. The payment for the ride is refunded to the sender.
3. The sender is able to see the canceled order in his orders..

3.11. Tracking an ongoing ride on the Shipogle app:

Prerequisite:

- The sender has placed an order and it has been accepted by the driver.
- The ride has been started by the driver.

Workflow:

1. The sender opens the Shipogle app on their device.
2. The sender navigates to the "Orders" section of the app.
3. The sender selects the "In Progress" tab to view their ongoing orders.
4. The sender selects the specific order they want to track by tapping on it.
5. The sender sees the driver's current location and the estimated time of arrival (ETA) on the map.
6. The sender can also view details about the driver, such as their name, contact information, and vehicle information.
7. The sender can track the ride in real-time from the shown location.
8. The sender can communicate with the driver through the in-app chat if needed.

Post-conditions:

1. The sender is able to track the progress of their ongoing ride on the Shipogle app.
2. The sender is able to communicate with the driver through the in-app chat if needed.

3.12. Giving Feedback to the driver on Shipogle app

Prerequisite: The order is completed by the driver.

Workflow:

1. The sender opens the Shipogle app and navigates to the Completed Order section.
2. He selects the specific completed order for which he wants to provide feedback.
3. The sender clicks on the Give Feedback button on the order details page.
4. The app redirects the sender to the feedback form.
5. The feedback form allows the sender to provide a rating (on a scale of 1-5 stars) for the driver's performance during the ride.
6. The feedback form also provides a text box where the sender can add additional comments or suggestions related to the ride.
7. The sender fills out the feedback form and submits it.
8. After submitting the feedback, the sender receives a confirmation message that the feedback has been submitted successfully.

Post Conditions:

1. The app calculates the driver's average rating based on the feedback provided by all the senders.

2. The sender can view the driver's average rating on the Search dashboard while searching for rides.
3. The feedback provided by the sender helps Shipogle app to improve the quality of its services and make necessary changes.

3.13. Raising an Issue for the Ride on Shipogle App

Pre-requisite: The user has completed the ride with the driver.

Workflow:

1. The user goes to the "Completed Orders" section of the app.
2. The user selects the specific order for which they want to raise an issue.
3. The user clicks on the "Raise an Issue" button.
4. The user is presented with a form where they can describe the issue they faced during the ride.
5. The user enters the details of the issue in the form.
6. The user clicks on the "Submit" button.

Post Conditions:

1. The issue is submitted and recorded in the app's database.
2. The user is shown a message acknowledging the issue submission.
3. The issue gets shown in the Current issues section.

4. SMELLS

4.1. Architecture Smells

Architecture smells after left we have removed the initial ones:

ArchitectureSmells

Project Name	Package Name	Architecture Smell	Cause of the Smell
backend	com.shipogle.app.controller	Feature Concentration	The tool detected the smell in this component because the component realizes more than one architectural concern/feature. Independent sets of related classes within this component are: [ChatControllerIntegrationTest]; [No
backend	com.shipogle.app.controller	Scattered Functionality	The tool detected the smell in this component because a set of two or more components realizes the same high-level architectural concern. Following components realize the same concern: com.shipogle.app.model; com.sh
backend	com.shipogle.app.controller	Scattered Functionality	The tool detected the smell in this component because a set of two or more components realizes the same high-level architectural concern. Following components realize the same concern: com.shipogle.app.model; com.sh
backend	com.shipogle.app.service	God Component	The tool detected the smell in this component because the component contains high number of classes. Number of classes in the component are: 37
backend	com.shipogle.app.service	Scattered Functionality	The tool detected the smell in this component because a set of two or more components realizes the same high-level architectural concern. Following components realize the same concern: com.shipogle.app.model; com.sh
backend	com.shipogle.app.repository	Feature Concentration	The tool detected the smell in this component because the component realizes more than one architectural concern/feature. Independent sets of related classes within this component are: [NotificationRepository]; [ForgotPas
backend	com.shipogle.app.model	Feature Concentration	The tool detected the smell in this component because the component realizes more than one architectural concern/feature. Independent sets of related classes within this component are: [Message; User; PackageOrder; Pa

The Feature Concentration smell is appearing because we have followed the MVC architecture pattern in our Spring Boot application, and as a result, all the controllers are kept in the controllers package, services in the service package, and so on. Hence, this smell is expected in our codebase.

The Scattered Functionality smell is occurring because we have kept different functionality for repositories and models at a high level. This decision was made to keep the codebase organized and easy to maintain.

The God Component smell is due to the fact that we have implemented the core business logic of the application in the service layer. This is intentional, as it helps us keep the business logic in one place and makes it easier to maintain.

Overall, while these smells may indicate potential issues in other projects, in our case, we have taken a deliberate approach to our architecture and organization, and these smells are expected and acceptable in our codebase. We will continue to monitor and address any issues that may arise as we develop and maintain the application.

4.2. Design smells

DesignSmells

Project Name	Package Name	Type Name	Design Smell	Cause of the Smell
backend	com.shipogle.app.config	AuthConfig	Unutilized Abstraction	The tool detected the smell in this class because this class is potentially unused. (Please ignore the smell if the reported class is auto-generated and/or used to serve a specific known purpose.)
backend	com.shipogle.app	TestConstants	Unnecessary Abstraction	The tool detected the smell in this class because the class contains only a few data members without any method implementation
backend	com.shipogle.app	TestConstants	Deficient Encapsulation	The tool detected the smell in this class because the class exposes fields belonging to it with public accessibility. Following fields are declared with public accessiblity: TEST_TOKEN
backend	com.shipogle.app	ShipogleApplicationTests	Unnecessary Abstraction	The tool detected the smell in this class because the class contains only a few data members without any method implementation
backend	com.shipogle.app	ShipogleApplicationTests	Unutilized Abstraction	The tool detected the smell in this class because this class is potentially unused. (Please ignore the smell if the reported class is auto-generated and/or used to serve a specific known purpose.)
backend	com.shipogle.app.controller	ShipoglePaymentController	Unutilized Abstraction	The tool detected the smell in this class because this class is potentially unused. (Please ignore the smell if the reported class is auto-generated and/or used to serve a specific known purpose.)
backend	com.shipogle.app.socket_handlers	NotificationSocketHandlerTests	Feature Envy	The tool detected a instance of this smell because testAfterConnectionClosed is more interested in members of the type: NotificationSocketHandler
backend	com.shipogle.app.socket_handlers	ChatSocketHandler	Deficient Encapsulation	The tool detected the smell in this class because the class exposes fields belonging to it with public accessibility. Following fields are declared with public accessiblity: instance; sessions; ID_SPLITTER
backend	com.shipogle.app.model	DriverRoute	Insufficient Modularization	The tool detected the smell in this class becuase the class has bloated interface (large number of public methods). Total public methods in the class: 43 public methods
backend	com.shipogle.app.model	User	Insufficient Modularization	The tool detected the smell in this class becuase the class has bloated interface (large number of public methods). Total public methods in the class: 48 public methods
backend	com.shipogle.app.model	PackageOrder	Insufficient Modularization	The tool detected the smell in this class becuase the class has bloated interface (large number of public methods). Total public methods in the class: 26 public methods
backend	com.shipogle.app.utility	Const	Unutilized Abstraction	The tool detected the smell in this class because this class is potentially unused. (Please ignore the smell if the reported class is auto-generated and/or used to serve a specific known purpose.)
backend	com.shipogle.app.utility	Const	Deficient Encapsulation	The tool detected the smell in this class because the class exposes fields belonging to it with public accessibility. Following fields are declared with public accessiblity: UNAUTHORIZED_ERROR_CODE; SECRET_KEY;
backend	com.shipogle.app.utility	Const	Broken Modularization	The tool detected the smell in this class because it contains only data members without any method implementation. Following fields are declared in this class: UNAUTHORIZED_ERROR_CODE; SECRET_KEY; URL_FR

Unutilized Abstraction: This smell is coming in the controller, config and constant files because they are not directly called in the current implementation. However, they are called through external APIs.

Deficient Encapsulation: This smell is present in the Test constants to handle the integration test, as some data members need to be accessed from the test cases for the integration testing purposes.

Unnecessary Abstraction: This smell is coming in the test classes that have just a few methods to be tested upon, but creating a separate class or interface for them is not beneficial in terms of code organization or maintenance.

Insufficient Modularization: This smell is coming in three of the core classes of the Project that are in itself are big features. So, modularizing them was making the business logic a bit complex. However, we have removed this smell present at other places where modularization was possible without increasing complexity.

Broken Modularization: This smell is coming in the constant file we are using to run the integration test as it contains only data members without any method implementation. However, it is working as intended in the current implementation and modularization is not needed for this particular file.

Feature Envy: Feature Envy is coming in a test class because the test method testAfterConnectionClosed is interacting heavily with the NotificationSocketHandler class to test its functionality, which is expected in this case. Therefore, this smell can be ignored in this specific scenario.

4.3. Implementation smells

Here are some elaborations:

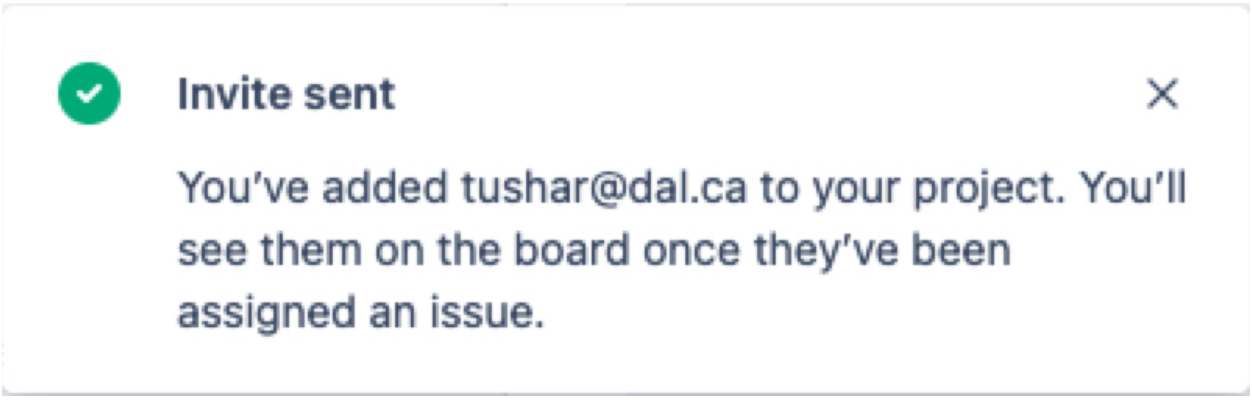
Long statement: This smell is showing up in an integration test class that requires a complex setup and multiple API calls, so it's expected to have long statements to set up the test scenario.

Long parameter list: The DriverRouteRepository needs to receive several filters to perform a specific search on the dashboard, which requires passing these parameters to the SQL query, making a long parameter list necessary for this particular case.

5. Jira Board

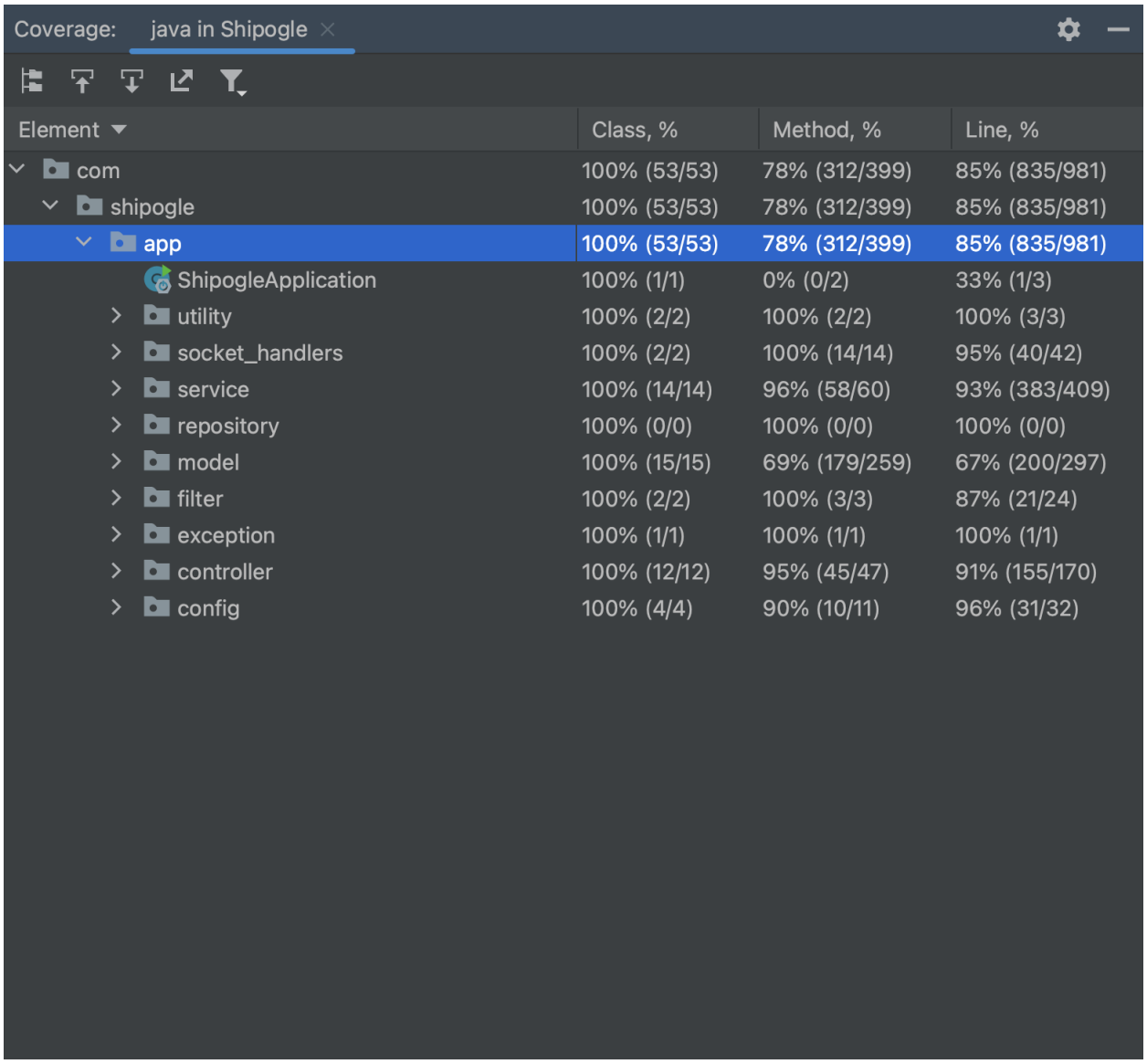
Link : https://shipogle.atlassian.net/jira/software/projects/SHAD/boards/3

We utilized the Jira board to execute our feature implementation and successfully finished all of them. Each team member completed their assigned tasks, documented their contributions, and adhered to the agile methodology by implementing sprints for efficient progress. Additionally, we extended an invitation to our professor Tushar for access to the Jira Board.



6. Unit Testing and Code Coverage for Application Optimization.

To ensure optimal functionality of our application, we have created unit tests for all testable classes, including **unit and integration tests**, while also addressing boundary cases. This approach has allowed us to achieve a code coverage of **approximately 85%**, which guarantees the smooth operation of the application.



7. REFERENCES

1. https://www.baeldung.com/java-generating-random-numbers-in-range
2. https://stackoverflow.com/questions/37620694/how-to-scroll-to-bottom-in-react
3. https://stackoverflow.com/questions/65764360/material-ui-remove-menu-padding
4. https://www.digitalocean.com/community/tutorials/
5. https://linuxize.com/post/how-to-unzip-files-in-linux/
6. https://www.cyberciti.biz/faq/unix-linux-execute-command-using-ssh/
7. https://blog.devgenius.io/using-nginx-to-serve-react-application-static-vs-proxy-69b85f368e6c?gi=49ae7fb70188
8. https://www.w3schools.com/css/css_boxmodel.asp
9. https://www.digitalocean.com/community/tutorials/react-axios-react
10. https://stackoverflow.com/questions/54807454/what-is-prevstate-in-reactjs
11. https://www.baeldung.com/spring-boot-testing
12. https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/
13. https://nexocode.com/blog/posts/understanding-principles-of-gitlab-ci-cd-pipelines/
14. https://docs.gitlab.com/ee/ci/quick_start/
15. https://www.viralpatel.net/java-create-validate-jwt-token/
16. https://github.com/ali-bouali/spring-boot-3-jwt-security/blob/dfe46421a934aaa57e5d321a6f0f89e83b383acc/src/main/java/com/alibou/security/config/JwtService.java#L30
17. https://jwt.io/introduction
18. https://www.marcobehler.com/guides/spring-security
19. https://spring.io/guides/gs/securing-web/
20. https://www.toptal.com/spring/spring-security-tutorial
21. https://www.baeldung.com/spring-email
22. https://stackoverflow.com/questions/33015900/no-mimemessage-content-exception-when-sending-simplemailmessage
23. https://www.baeldung.com/spring-security-custom-logout-handler
24. https://spring.io/guides/gs/messaging-stomp-websocket/
25. https://blog.logrocket.com/websocket-tutorial-real-time-node-react/
26. https://www.baeldung.com/spring-websockets-send-message-to-user