

# Parcel Delivery System

## Enterprise Software Engineering

Student ID - 20030816

Ignacio Vidal

V030816k@students.staffs.ac.uk

May 2022

# Table of Contents

Introduction .....	4
System Specifications.....	4
1. Feature: Driver Registration.....	4
2. Feature: Customer Registration.....	4
3. Feature: User login.....	6
4. Feature: Find Parcels by Status.....	7
5. Feature: Find Parcel by ID .....	8
6. Feature: Customer register a parcel delivery .....	8
7. Feature: Customer drop a parcel for delivery.....	9
8. Feature: Recipient book a parcel collection.....	9
9. Feature: Driver selects a parcel for delivery .....	9
10. Feature: Driver collects a parcel for delivery .....	9
11. Feature: Driver delivers a parcel to customer .....	9
12. Feature: Driver returns a parcel to warehouse .....	10
13. Feature: Recipient collects a parcel from warehouse .....	10
14. Feature: Recipient reject a parcel from driver.....	10
System Design .....	11
Architecture .....	11
Domain Model .....	12
UML Diagrams.....	13
Parcel Lifecycle.....	13
Design Patterns .....	14
Database .....	15
SQL Script to Create Tables.....	16
SQL Script to Seed Data.....	17
Test Strategy .....	18
Future Development .....	18
Bibliography .....	19

## Table of Figures

Figure 1. Process flow to register users .....	5
Figure 2. Process to login .....	6
Figure 3. Process flow to find parcels by status.....	7
Figure 4. Process flow to find parcel by id .....	8
Figure 5. Process flow to register new parcel.....	8
Figure 6. Process flow to update parcel state.....	10
Figure 7. Backend application architecture .....	11
Figure 8. Class hierarchy of the application .....	12
Figure 9. UML representation of main diagrams.....	13
Figure 10. Parcel lifecycle.....	14
Figure 11. Database model .....	15
Figure 12. Result of test cases.....	18

## Introduction

London Warehouse LTD is a fictitious company that operates a warehouse in London and acts as a middleman between parcel senders and receivers. The company offers a web application where people can book parcel deliveries and drivers can select packages for delivery. Senders must drop the packages in the warehouse where they are collected by self-employed drivers.

The report includes the functional requirements, system architecture, UML diagrams, database model and testing strategy of the Parcel Delivery application. The application is developed as web service that exposes HTTP RESTful APIs, uses JSON to encode the payloads and is built on top of:

- Java 11: This is the open-source implementation of the Java SE Platform (Oracle, 2022)
- Spring: It is a framework to build web services and provides inversion control, dependency injection and an ecosystem of libraries to extend its core functionality (VMware, 2022a)
- Spring Boot: It is an opinionated framework that configures Spring modules to build stand-alone applications (VMware, 2022b)
- JUnit 5: It is a testing running and assertions library for java applications (JUnit, 2022)

## System Specifications

### 1. Feature: Driver Registration

As a self-employed DRIVER I want to register with my name, email, password.

This feature enables any user to register as a Driver. Constraints:

- A driver is uniquely identified based on their email address
- The same email address cannot be used to register more than one account
- The plaintext password must not be stored in the database
- The plaintext password must be hashed and stored in the database

### 2. Feature: Customer Registration

As a CUSTOMER I want to register with my name, email, password.

This feature enables users to register as Customers so that they can book parcel deliveries. Constraints:

- A customer is uniquely identified based on their email address
- The same email address cannot be used to register more than one account
- The plaintext password must not be stored in the database
- The plaintext password must be hashed and stored in the database

This is the process flow to register users with the role CUSTOMER and DRIVER:

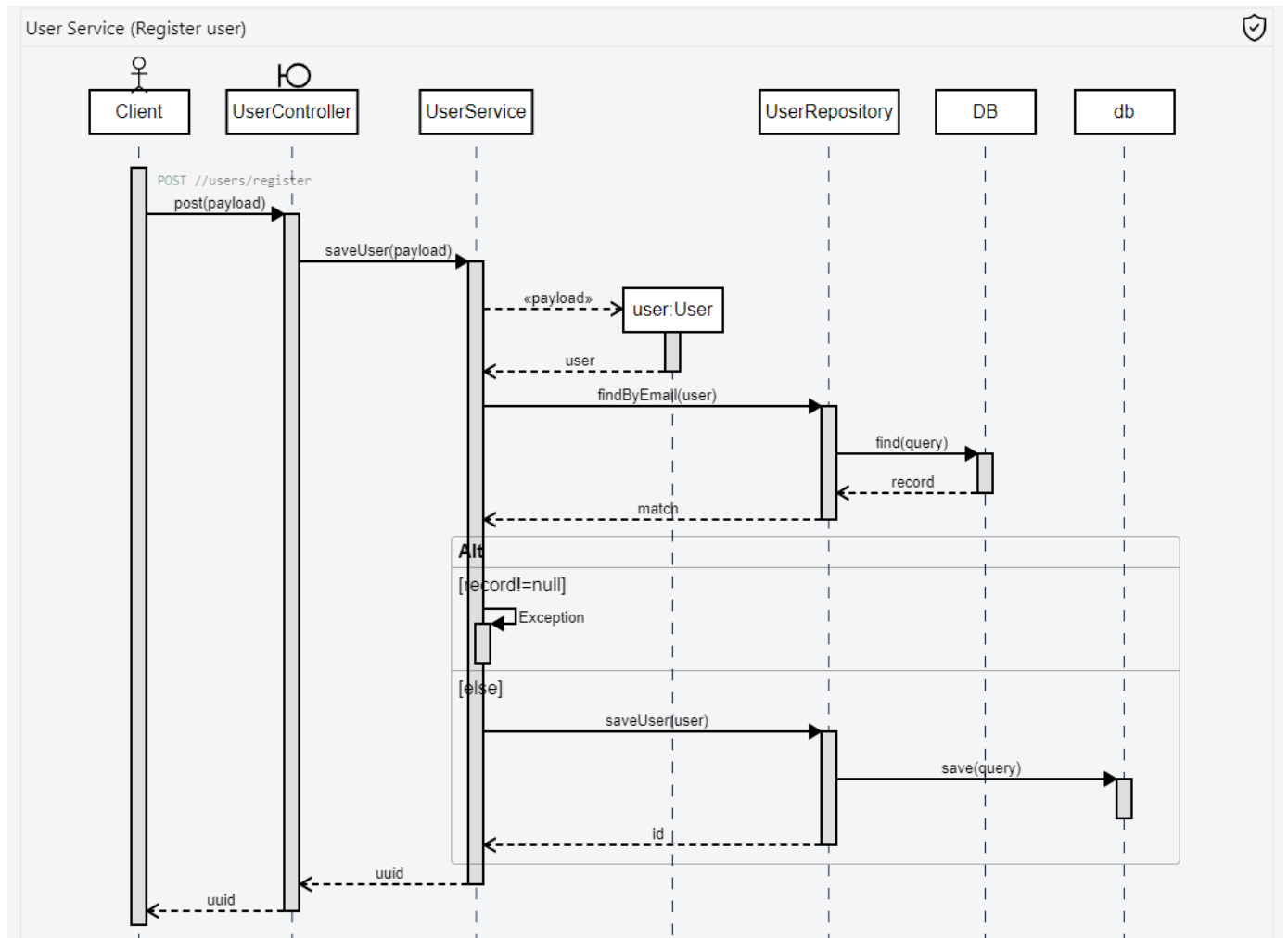


Figure 1. Process flow to register users

### 3. Feature: User Login

As a CUSTOMER or DRIVER, I want to log into my account using my email and password

This feature enables users to login and returns a JSON Web Token (JWT) with the user information and permissions in the Authorization header. Constrains:

- The password must be hashed and then compared against the hash stored in the database

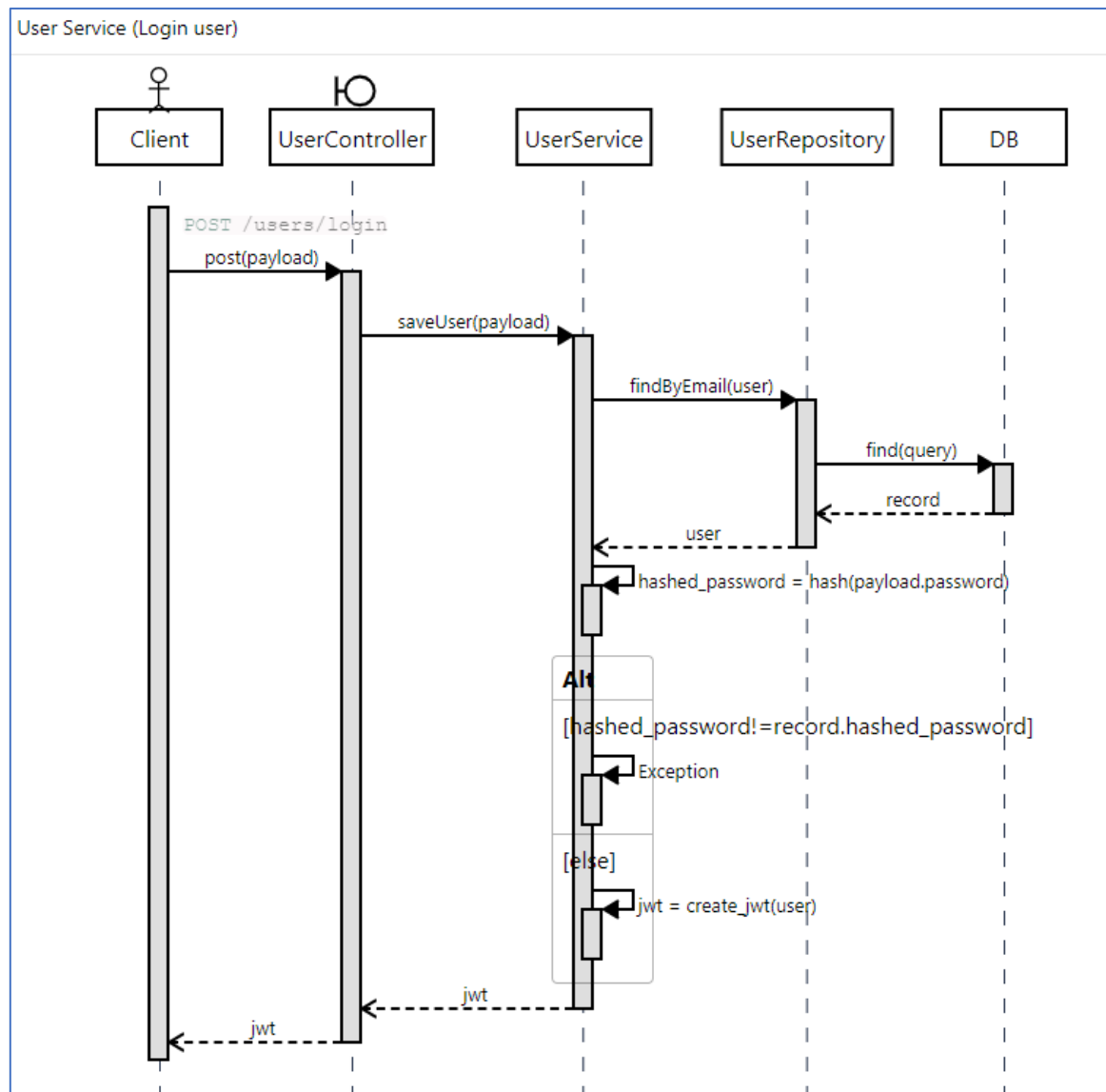


Figure 2. Process to login

#### 4. Feature: Find Parcels by Status

As a DRIVER I want to identify all the parcels by a given status.

This feature helps drivers identify which parcels are ready for delivery in the warehouse. There is a finite set of valid status. Constraints:

- Authentication: User must be authenticated with a valid JWT token
- Authorization: User must have the DRIVER role assigned
- The status in the request is subject to input validations

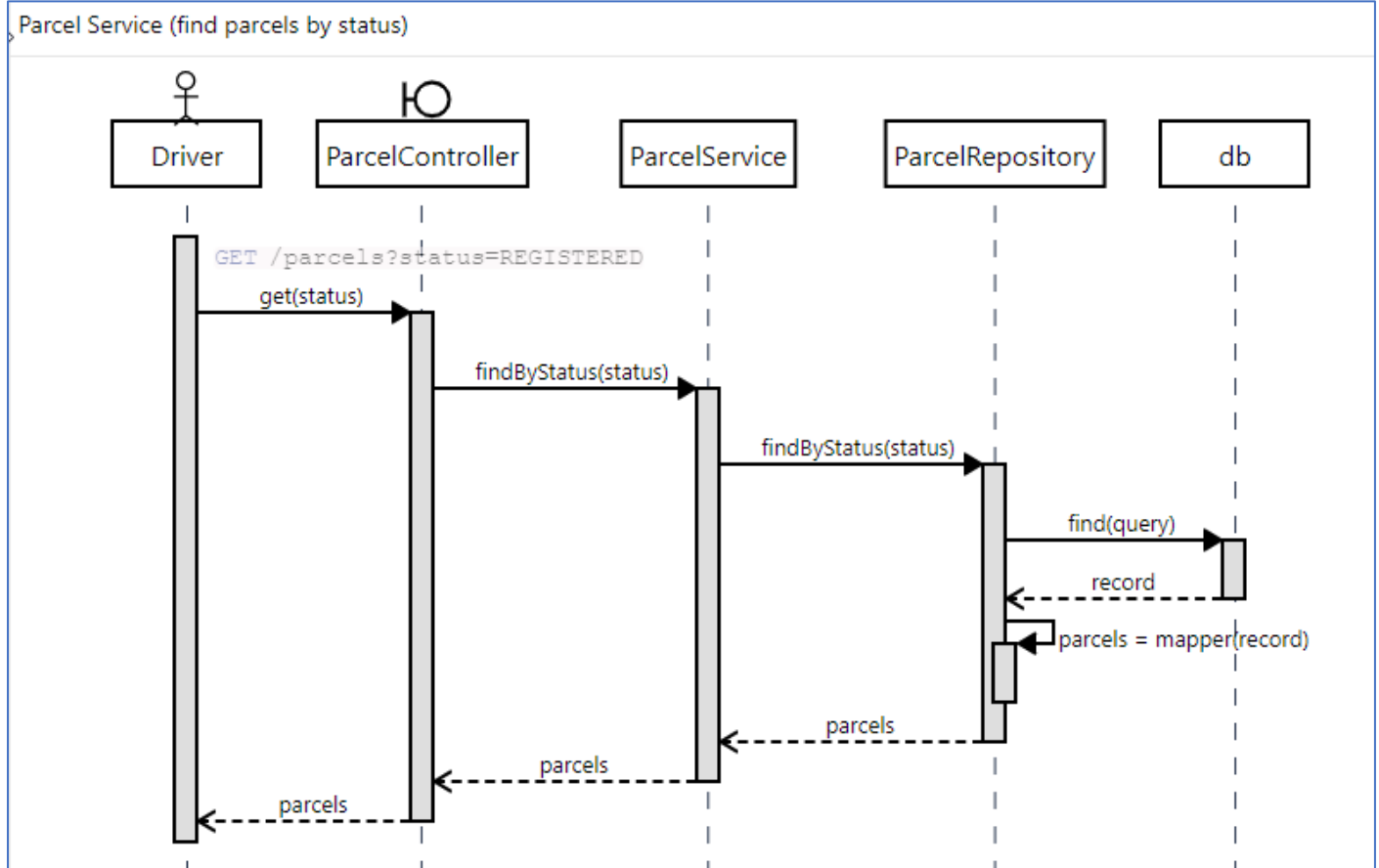


Figure 3. Process flow to find parcels by status

## 5. Feature: Find Parcel by ID

As a user I want to find a Parcel based on the universal unique identifier UUID and see its details.

This feature helps customers check the details of a given parcel. Constrains:

- Authentication: User must be authenticated with a valid JWT token
- Authorization: User must have CUSTOMER or DRIVER role

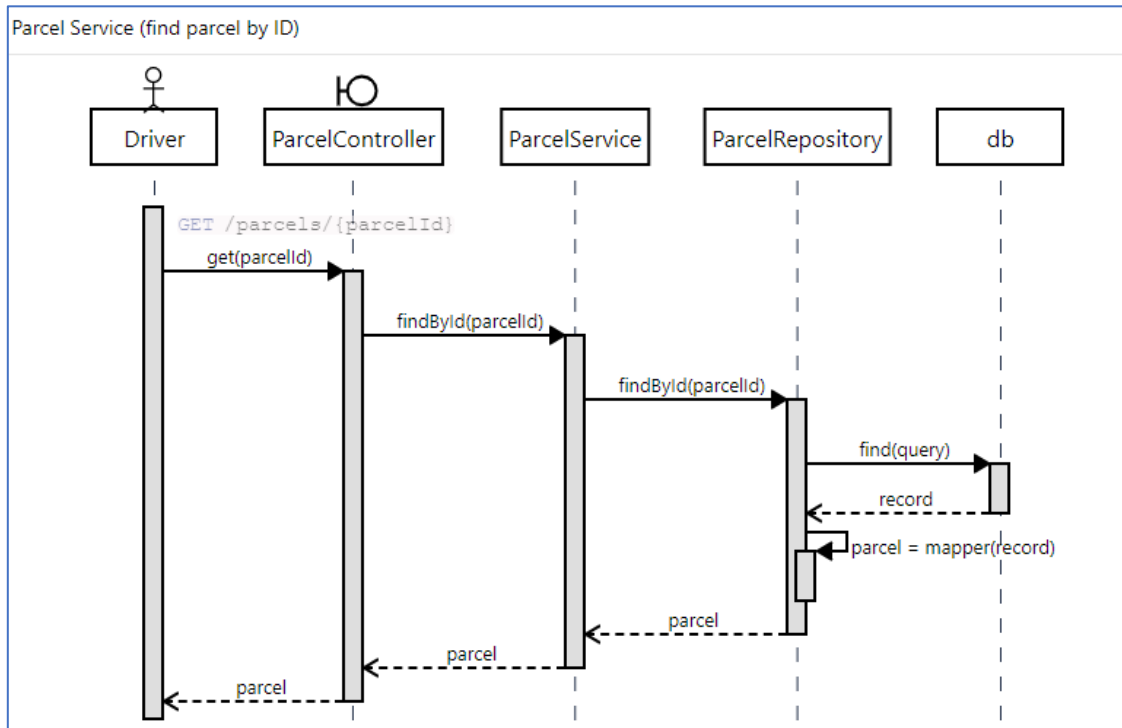


Figure 4. Process flow to find parcel by id

## 6. Feature: Customer registers a parcel delivery

As a CUSTOMER I want to register a new parcel for delivery.

It enables customers to book a parcel for delivery online before they drop it in the warehouse. Constraints:

- STATUS: A new parcel will be created with the status "REGISTERED"
- Authentication: User must be authenticated with a valid JWT token
- Authorization: User must have CUSTOMER or DRIVER role
- A UUID of the parcel will be generated and provided to the user

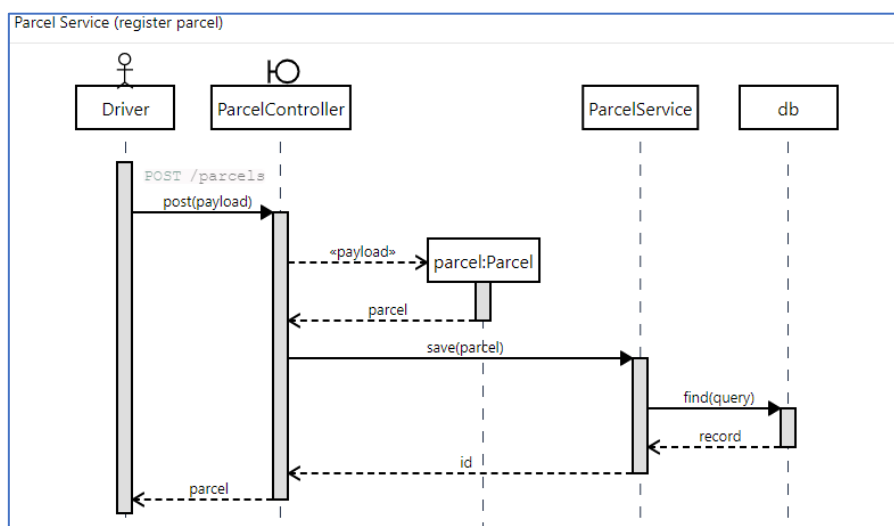


Figure 5. Process flow to register new parcel



## 7. Feature: Customer drops a parcel for delivery

As a CUSTOMER I want to drop a parcel in the warehouse.

Customers can drop pre-booked parcels in the warehouse and the operator will update the parcel status so it can be selected by drivers for delivery. Constraints:

- STATUS: The parcel must be "REGISTERED"
- STATUS: The parcel will be transitioned to "READY\_FOR\_ALLOCATION"
- Authentication: User must be authenticated with a valid JWT token
- Authorization: User must have DRIVER role

## 8. Feature: Recipient books a parcel collection

As a customer I want to book a parcel for collection in the warehouse.

It enables customers to book a parcel for local collection at the warehouse. Constraints:

- STATUS: The parcel must be "READY\_FOR\_ALLOCATION"
- STATUS: The parcel will be transitioned to "BOOKED\_FOR\_COLLECTION"
- Authentication: User must be authenticated with a valid JWT token
- Authorization: User must have CUSTOMER or DRIVER role

## 9. Feature: Driver selects a parcel for delivery

As a DRIVER I want to see the parcels pending delivery in the warehouse so I can book their delivery.

It enables drivers to see the parcels that are ready in the warehouse. Constraints:

- A parcel can only be assigned to one driver
- STATUS: The parcel must be "READY\_FOR\_ALLOCATION"
- STATUS: The parcel will be transitioned to "DELIVERY\_ASSIGNED"
- Authentication: User must be authenticated with a valid JWT token
- Authorization: User must have DRIVER role

## 10. Feature: Driver collects a parcel for delivery

As a DRIVER I want to collect the parcel I booked for delivery online

This feature enables drivers to collect the parcels they have booked for delivery. The status update is performed by a warehouse employee that holds the DRIVER role. Constraints:

- STATUS: The parcel must be "DELIVERY\_ASSIGNED"
- STATUS: The parcel will be transitioned to "OUT\_FOR\_DELIVERY"
- Authentication: User must be authenticated with a valid JWT token
- Authorization: User must have DRIVER role

## 11. Feature: Driver delivers a parcel to the customer

As a driver I want to update the parcel as delivered once it has been handed to the customer.

Constraints:

- STATUS: The parcel must be "OUT\_FOR\_DELIVERY"
- STATUS: The parcel will be transitioned to "DELIVERED"
- Authentication: User must be authenticated with a valid JWT token
- Authorization: User must have DRIVER role

## 12. Feature: Driver returns a parcel to the warehouse

As a driver I want to be able to return a parcel to the warehouse when the delivery was not successful so that the parcel may be booked for delivery again.

Constraints:

- STATUS: The parcel must be "OUT\_FOR\_DELIVERY"
- STATUS: The parcel will be transitioned to "READY\_FOR\_ALLOCATION"
- Authentication: User must be authenticated with a valid JWT token
- Authorization: User must have DRIVER role

## 13. Feature: Recipient collects a parcel from the warehouse

As a customer I want to collect a parcel locally in the warehouse

It enables customers to pick up the parcels booked for local collection in the warehouse to avoid waiting for their delivery. The status update is performed by a warehouse operator that holds the DRIVER role. Constraints:

- STATUS: The parcel must be "BOOKED\_FOR\_COLLECTION"
- STATUS: The parcel will be transitioned to "DELIVERED"
- Authentication: User must be authenticated with a valid JWT token
- Authorization: User must have DRIVER role

## 14. Feature: Recipient reject a parcel from a driver

As a CUSTOMER I want to reject a parcel delivery from a driver.

It enables drivers to reflect that a customer rejected the delivery of a parcel. The driver can return the item to the warehouse and the operator can update the parcel state. Constraints:

- STATUS: The parcel must be "OUT\_FOR\_DELIVERY"
- STATUS: The parcel will be transitioned to "REJECTED\_BY\_CUSTOMER"
- Authentication: User must be authenticated with a valid JWT token
- Authorization: User must have DRIVER role

This is the process flow of all the parcel updates covered in features 7-14:

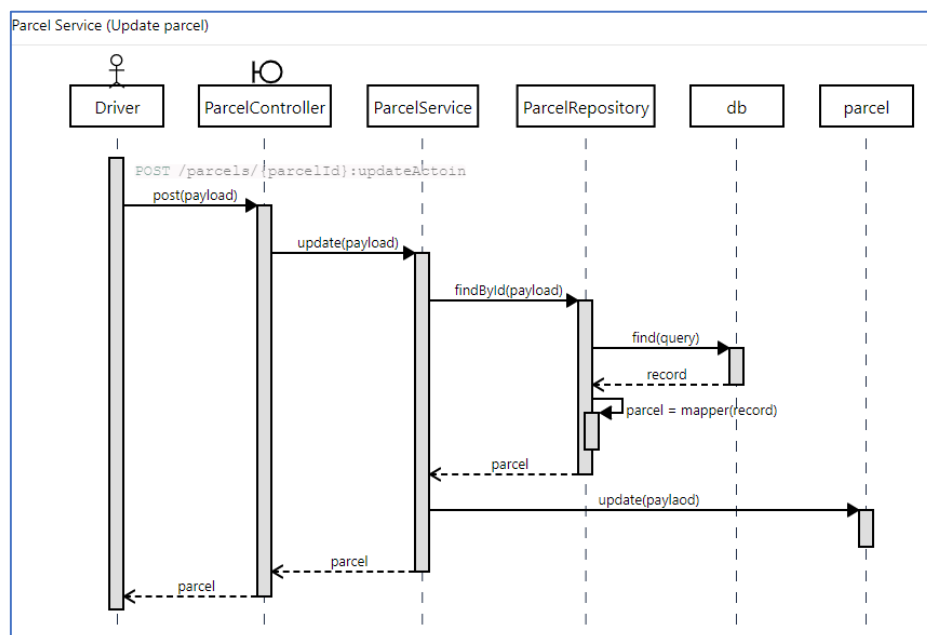


Figure 6. Process flow to update parcel state

# System Design

## Architecture

The backend application is divided in 4 layers to following the Onion Architecture pattern (Tapas, 2018):

- **Controller:** Acts as an adapter for HTTP requests and performs input validation, authentication and authorization validation
- **Service:** Defines single entry points for each user journey and is currently invoked by the controller
- **Repository:** It is the adapter for the database and currently uses PostgreSQL
- **Domain:** It includes the entities representing the domain and they encapsulate the business constraints to provide rich domain

This architecture pattern was selected because:

- **Enables multiple entry points to the application:** A new adapter to consume messages from a message bus such as Kafka could be introduced, and the web layer would not be affected
- **Prioritises rich domain objects:** The Parcel entity encapsulate business constraints to transition the parcel state providing a clean interface to the service layer
- **Encapsulate features:** The service layer only depends on the domain objects and its only responsible for orchestrating the steps required to deliver a functionality feature

It is worth mentioning this approach is suited for large enterprise application and might be considered overkill for a small application. A simpler solution would have been to remove the repository layer at the cost of coupling the application to PostgreSQL DB and move the domain constraints to the service layer to have an anaemic domain model. However, this project has been used to practise developing enterprise grade applications and therefore, the entire onion architecture pattern was implemented.

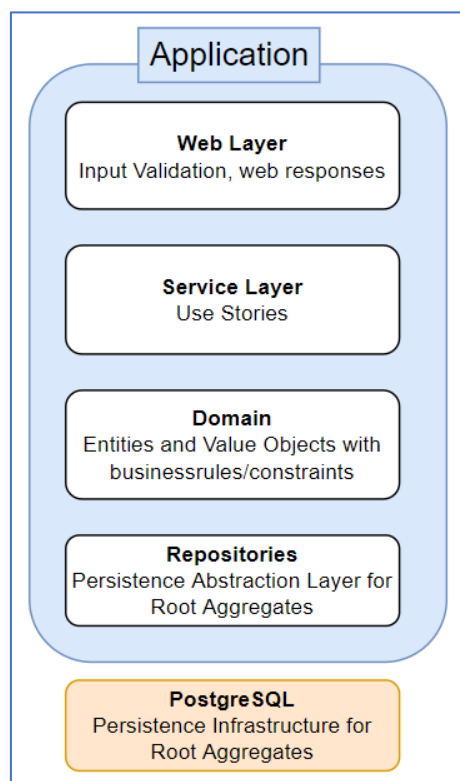


Figure 7. Backend application architecture

## Domain Model

This graph shows the domain model and class hierarchy of the application. The picture can be read top to bottom and each column is roughly the domain objects required for a user journey.

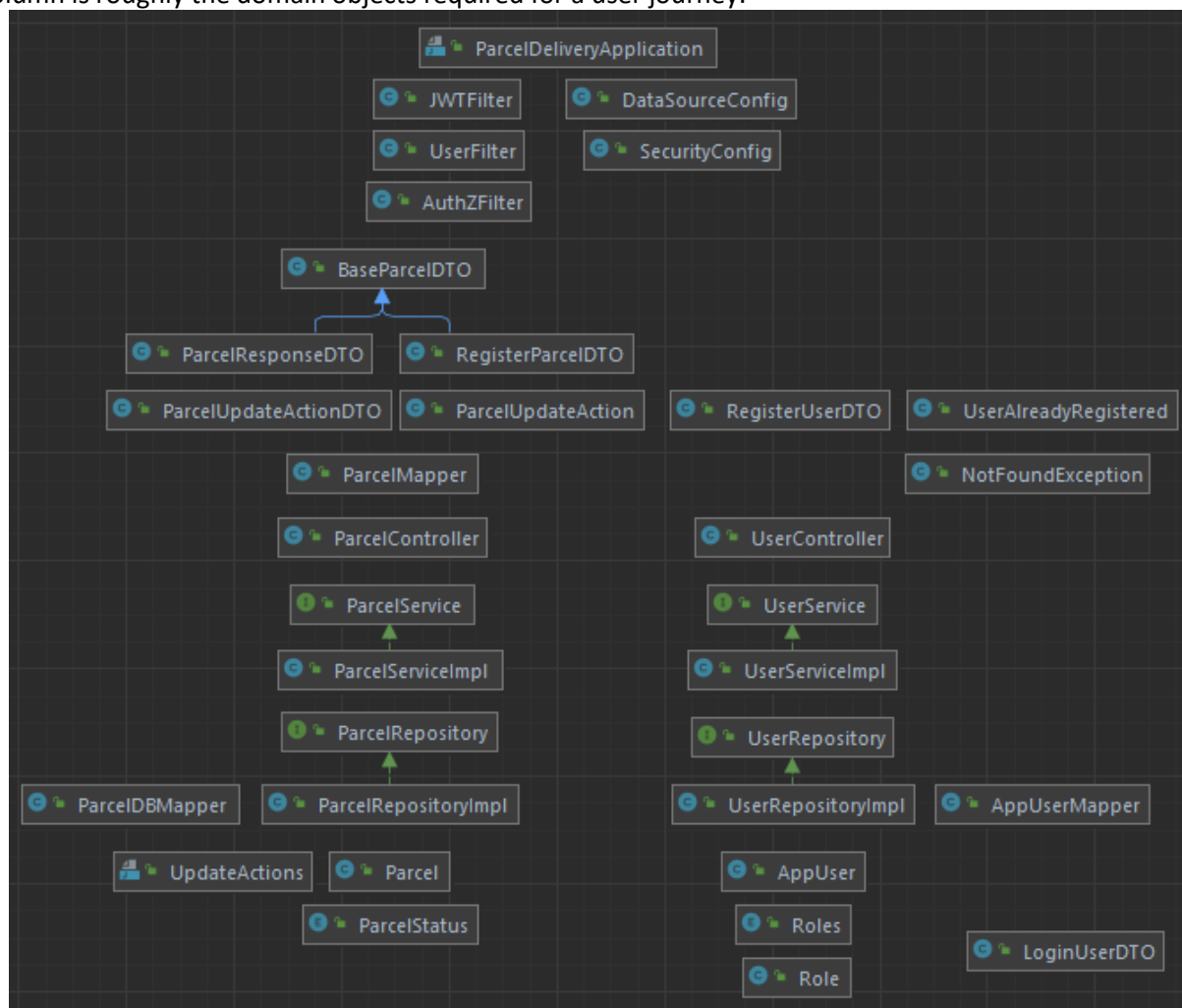


Figure 8. Class hierarchy of the application

## UML Diagrams

These are the UML representation of the Parcel and AppUser entities and any associated Enums.

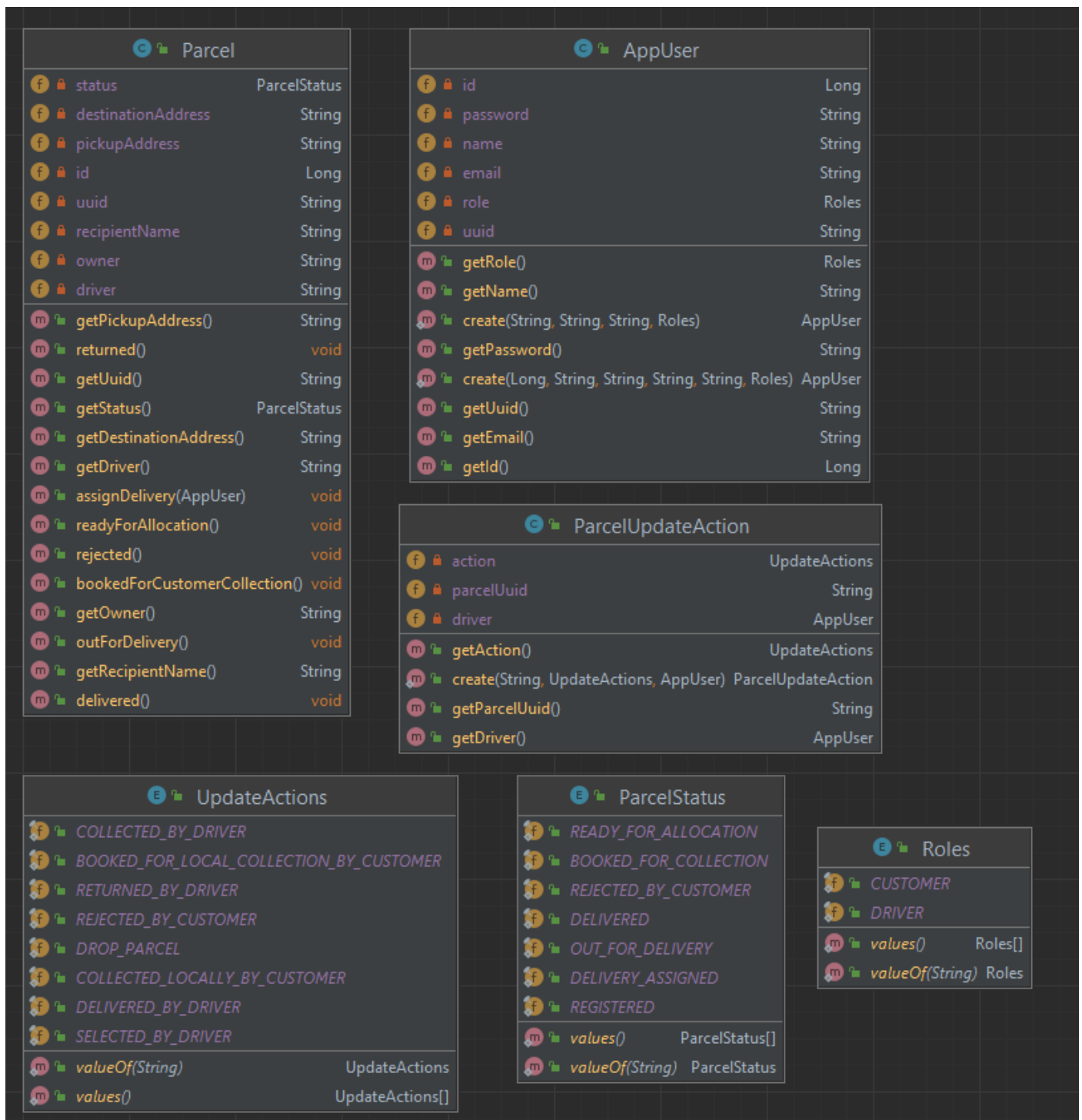


Figure 9. UML representation of main diagrams

## Parcel Lifecycle

The Parcel entity has been modelled as finite state machine where a finite set of states is available and only one state is possible at a given time. This provides the benefit of encapsulating the business constraints within each state. This graph depicts the lifecycle of a Parcel entity:

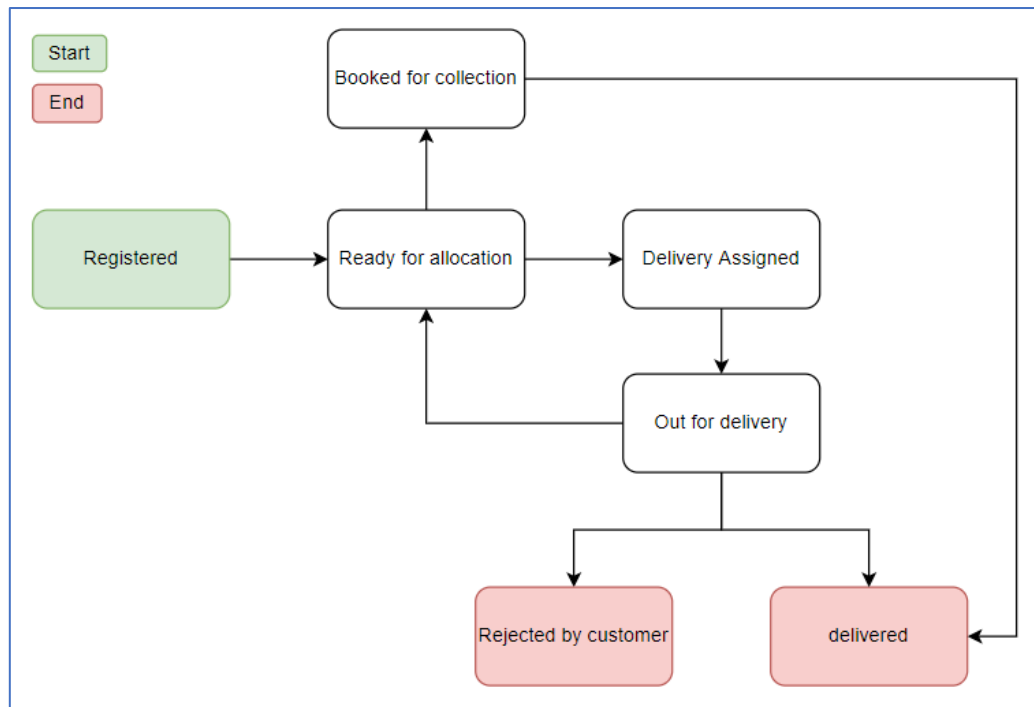


Figure 10. Parcel lifecycle

## Design Patterns

This application has implemented various design patterns to maintain a clean architecture and encapsulation of responsibilities:

- **Façade:** The service layer acts as a façade that hides the complexity of each user case under a single-entry point. For example, to update the state of parcel, there is an Action object that represent the update to be applied and service layer decides which update should be applied to a Parcel
- **Repository:** This creates an abstraction layer of the database based on an interface which provides the flexibility to swap the database infrastructure seamlessly
- **Factory and Builder:** The entity objects are immutable by design and can only be instantiated via static factory methods or static builder classes. Factory methods provide an interface to create objects and builders provide a fluent and clean interface to instantiate complex objects. Given objects are immutable, they do not expose any setters which makes them thread safe.
- **Dependency Inversion:** The services and repositories depend on their interfaces, which makes them open to extension and closed to modification (Open/Closed Principle). In addition, the follow the Liskov Substitution principle, because any repository implementing the interface can be exchanged without breaking the application. As a result, the dependency inversion is achieved.

Other design patterns such as Strategy was considered, however, the state updates of the Parcel entity are not complex enough to make worth creating an entire classes hierarchy to apply state updates. In summary, adding more design patterns would have introduced a significant amount of boiler plate with limited benefits for a small application.

## Database

This application uses a PostgreSQL, a relation database, because it is open source, lightweight and efficient. The database runs as docker container and exposes port 5432 to the application. The database migrations are managed through the application with the Flyway package that reads the .sql files located at resources/db.migration. This a visualization of the database model in the application.

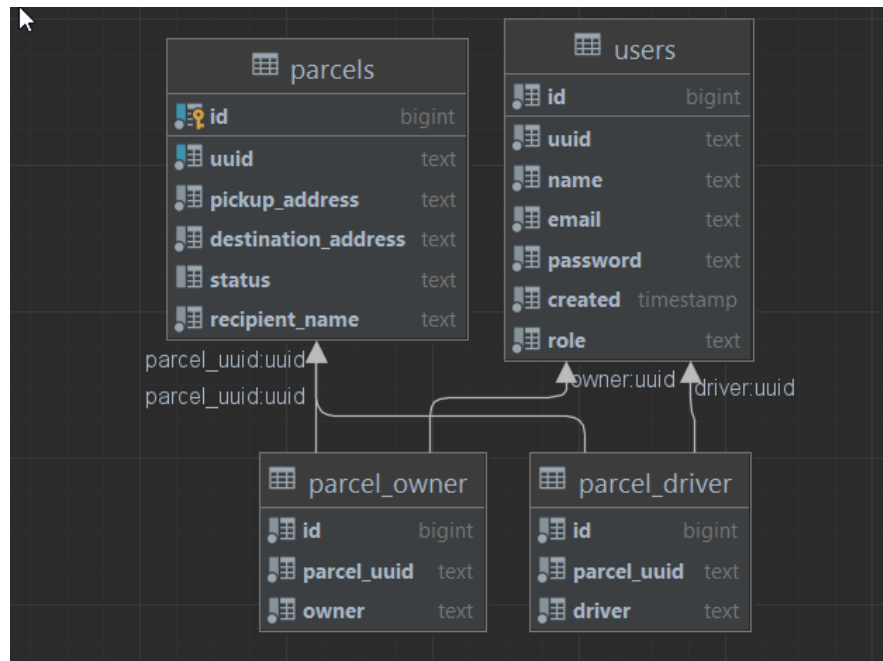


Figure 11. Database model

## SQL Script to Create Tables

This is the first migration applied to the database that created the tables:

```
CREATE TABLE users
(
  id          BIGSERIAL NOT NULL PRIMARY KEY,
  uuid        TEXT      NOT NULL UNIQUE,
  name        TEXT      NOT NULL,
  email       TEXT      NOT NULL,
  password    TEXT      NOT NULL,
  created     TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  role        TEXT      NOT NULL
);

CREATE TABLE parcels
(
  id          BIGSERIAL NOT NULL PRIMARY KEY,
  uuid        TEXT      NOT NULL UNIQUE,
  pickup_address TEXT    NOT NULL,
  destination_address TEXT NOT NULL,
  status      TEXT      NOT NULL,
  recipient_name TEXT    NOT NULL
);

CREATE TABLE parcel_owner
(
  id          BIGSERIAL NOT NULL,
  parcel_uuid TEXT      NOT NULL,
  owner       TEXT      NOT NULL,
  FOREIGN KEY (parcel_uuid) REFERENCES parcels (uuid),
  FOREIGN KEY (owner) REFERENCES users (uuid)
);

CREATE TABLE parcel_driver
(
  id          BIGSERIAL NOT NULL,
  parcel_uuid TEXT      NOT NULL,
  driver      TEXT      NOT NULL,
  FOREIGN KEY (parcel_uuid) REFERENCES parcels (uuid),
  FOREIGN KEY (driver) REFERENCES users (uuid)
);
```



## SQL Script to Seed Data

This is the script used to populate the database:

```
INSERT INTO users (uuid, name, email, password, created, role)
VALUES ('4c9c2a9f-ce78-40f7-8591-df60421d3d82', 'Ignacio', 'test@gmail.com', '123456',
now(), 'DRIVER');
INSERT INTO users (uuid, name, email, password, created, role)
VALUES ('666c63ef-ecel-4abf-aea4-40fe3430920c', 'Nacho', 'test@gmail.com', '123456',
now(), 'CUSTOMER');
INSERT INTO users (uuid, name, email, password, created, role)
VALUES ('9d719427-8119-4a94-9a74-5f5e4a73cdac', 'Inaki', 'test@gmail.com', '123456',
now(), 'DRIVER');
INSERT INTO users (uuid, name, email, password, created, role)
VALUES ('9e4b1400-ab27-446f-a6c7-b6ad41148acc', 'Inazio', 'test@gmail.com', '123456',
now(), 'CUSTOMER');

INSERT INTO parcels(uuid, pickup_address, destination_address, status, recipient_name)
VALUES ('f6833235-28ae-4e53-8dc3-db5ccad38dd6', 'Calle 1', 'Calle 2', 'REGISTERED',
'Ignacio');
INSERT INTO parcels(uuid, pickup_address, destination_address, status, recipient_name)
VALUES ('07e09b43-0b05-464d-b831-4c48fcc8e874', 'Calle 3', 'Calle 4', 'REGISTERED',
'Nacho');
INSERT INTO parcels(uuid, pickup_address, destination_address, status, recipient_name)
VALUES ('d6b0025c-b0b9-4f70-95bd-30b9cb462fd1', 'Calle 5', 'Calle 6', 'REGISTERED',
'Inaki');
INSERT INTO parcels(uuid, pickup_address, destination_address, status, recipient_name)
VALUES ('61b55ce3-b749-4a1f-b18d-fcaa7ae9bd73', 'Calle 7', 'Calle 8', 'REGISTERED',
'Inazio');

INSERT INTO parcel_owner(parcel_uuid, owner)
VALUES ('f6833235-28ae-4e53-8dc3-db5ccad38dd6', '9d719427-8119-4a94-9a74-
5f5e4a73cdac');
INSERT INTO parcel_owner(parcel_uuid, owner)
VALUES ('07e09b43-0b05-464d-b831-4c48fcc8e874', '666c63ef-ecel-4abf-aea4-
40fe3430920c');
INSERT INTO parcel_owner(parcel_uuid, owner)
VALUES ('d6b0025c-b0b9-4f70-95bd-30b9cb462fd1', '666c63ef-ecel-4abf-aea4-
40fe3430920c');
INSERT INTO parcel_owner(parcel_uuid, owner)
VALUES ('61b55ce3-b749-4a1f-b18d-fcaa7ae9bd73', '9d719427-8119-4a94-9a74-
5f5e4a73cdac');

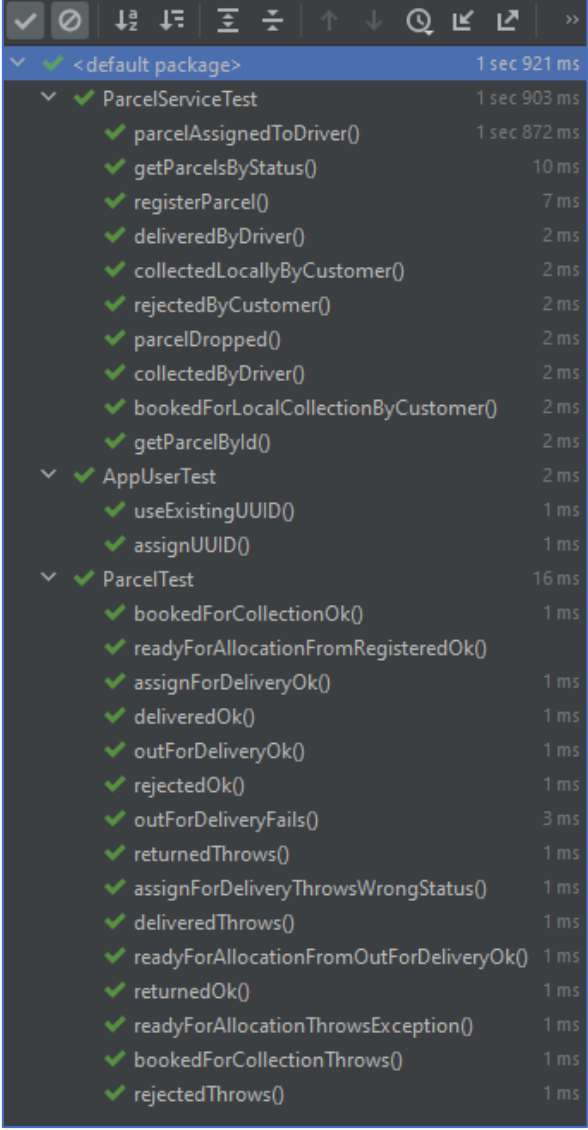
INSERT INTO parcel_driver(parcel_uuid, driver)
VALUES ('f6833235-28ae-4e53-8dc3-db5ccad38dd6', '9d719427-8119-4a94-9a74-
5f5e4a73cdac');
INSERT INTO parcel_driver(parcel_uuid, driver)
VALUES ('07e09b43-0b05-464d-b831-4c48fcc8e874', '666c63ef-ecel-4abf-aea4-
40fe3430920c');
INSERT INTO parcel_driver(parcel_uuid, driver)
VALUES ('d6b0025c-b0b9-4f70-95bd-30b9cb462fd1', '666c63ef-ecel-4abf-aea4-
40fe3430920c');
INSERT INTO parcel_driver(parcel_uuid, driver)
VALUES ('61b55ce3-b749-4a1f-b18d-fcaa7ae9bd73', '9d719427-8119-4a94-9a74-
5f5e4a73cdac');
```

## Test Strategy

The test strategy prioritises unit and integration test cases:

- Unit test cases are small, efficient and mock external dependencies. They have been applied to the entity objects of the application that enforce the domain constraints
- Integration test cases cover the Service layer and mock the database for an in-memory object. These test cases cover the functionality features described above

The test cases can be executed via “mvn test” command or directly in IntelliJ. This is a screenshot of the test results executed in IntelliJ:



✓	<default package>	1 sec 921 ms
✓	ParcelServiceTest	1 sec 903 ms
✓	parcelAssignedToDriver()	1 sec 872 ms
✓	getParcelsByStatus()	10 ms
✓	registerParcel()	7 ms
✓	deliveredByDriver()	2 ms
✓	collectedLocallyByCustomer()	2 ms
✓	rejectedByCustomer()	2 ms
✓	parcelDropped()	2 ms
✓	collectedByDriver()	2 ms
✓	bookedForLocalCollectionByCustomer()	2 ms
✓	getParcelById()	2 ms
✓	AppUserTest	2 ms
✓	useExistingUUID()	1 ms
✓	assignUUID()	1 ms
✓	ParcelTest	16 ms
✓	bookedForCollectionOk()	1 ms
✓	readyForAllocationFromRegisteredOk()	
✓	assignForDeliveryOk()	1 ms
✓	deliveredOk()	1 ms
✓	outForDeliveryOk()	1 ms
✓	rejectedOk()	1 ms
✓	outForDeliveryFails()	3 ms
✓	returnedThrows()	1 ms
✓	assignForDeliveryThrowsWrongStatus()	1 ms
✓	deliveredThrows()	1 ms
✓	readyForAllocationFromOutForDeliveryOk()	1 ms
✓	returnedOk()	1 ms
✓	readyForAllocationThrowsException()	1 ms
✓	bookedForCollectionThrows()	1 ms
✓	rejectedThrows()	1 ms

Figure 12. Result of test cases

The testing strategy could be expanded to include end-to-end testing where the application is tested via the HTTP RESTful APIs, including the database. This type of testing is computationally expensive and slow due to the network calls, therefore should be implemented in small amounts.

## Future Development

This application could be enhanced to enable real time notifications of the state changes to Parcels. This could be enabled with asynchronous email notifications triggered via message bus that could keep customers engaged with the business. In addition, a mobile application could be developed and integrated via the REST APIs to expand the customer outreach. Finally, the testing strategy could be expanded to include end-to-end test cases that reflect the functionality features.

## Bibliography

- JUnit, team (2022). *JUnit 5 User Guide*. [Online]. 2022. Available from: <https://junit.org/junit5/docs/current/user-guide/>. [Accessed: 15 May 2022].
- Oracle, C. (2022). *JDK 11*. [Online]. 2022. Available from: <https://openjdk.java.net/projects/jdk/11/>. [Accessed: 15 May 2022].
- Tapas, P. (2018). *Onion Architecture: Definition, Principles & Benefits*. [Online]. 12 February 2018. CodeGuru. Available from: <https://www.codeguru.com/csharp/understanding-onion-architecture/>. [Accessed: 15 May 2022].
- VMware, I. (2022a). *Spring*. [Online]. 2022. Spring. Available from: <https://spring.io/>. [Accessed: 15 May 2022].
- VMware, I. (2022b). *Spring Boot*. [Online]. 2022. Available from: <https://spring.io/projects/spring-boot>. [Accessed: 15 May 2022].