RMIT Vietnam  
School of Science, Engineering & Technology (SSET)  
COSC2440 – Further Programming  
Lecturer: Mr. Minh Vu Thanh

### 

Assignment 2

Software Design Report

By Tri Truong

s3878145 – Tri Truong

# Introduction

In this project, I was tasked with creating a backend REST API for an enterprise application for a Taxi company. This API would be responsible for automating business processes, allowing the customers and drivers to facilitate their trips and bookings, as well as providing the company with valuable statistical information. This solution is built using Spring Framework in tandem with a number of dependencies, most notably Spring Web (Spring MVC), Hibernate, and Lombok and is designed to specifically work with MySQL databases.

# Business requirement

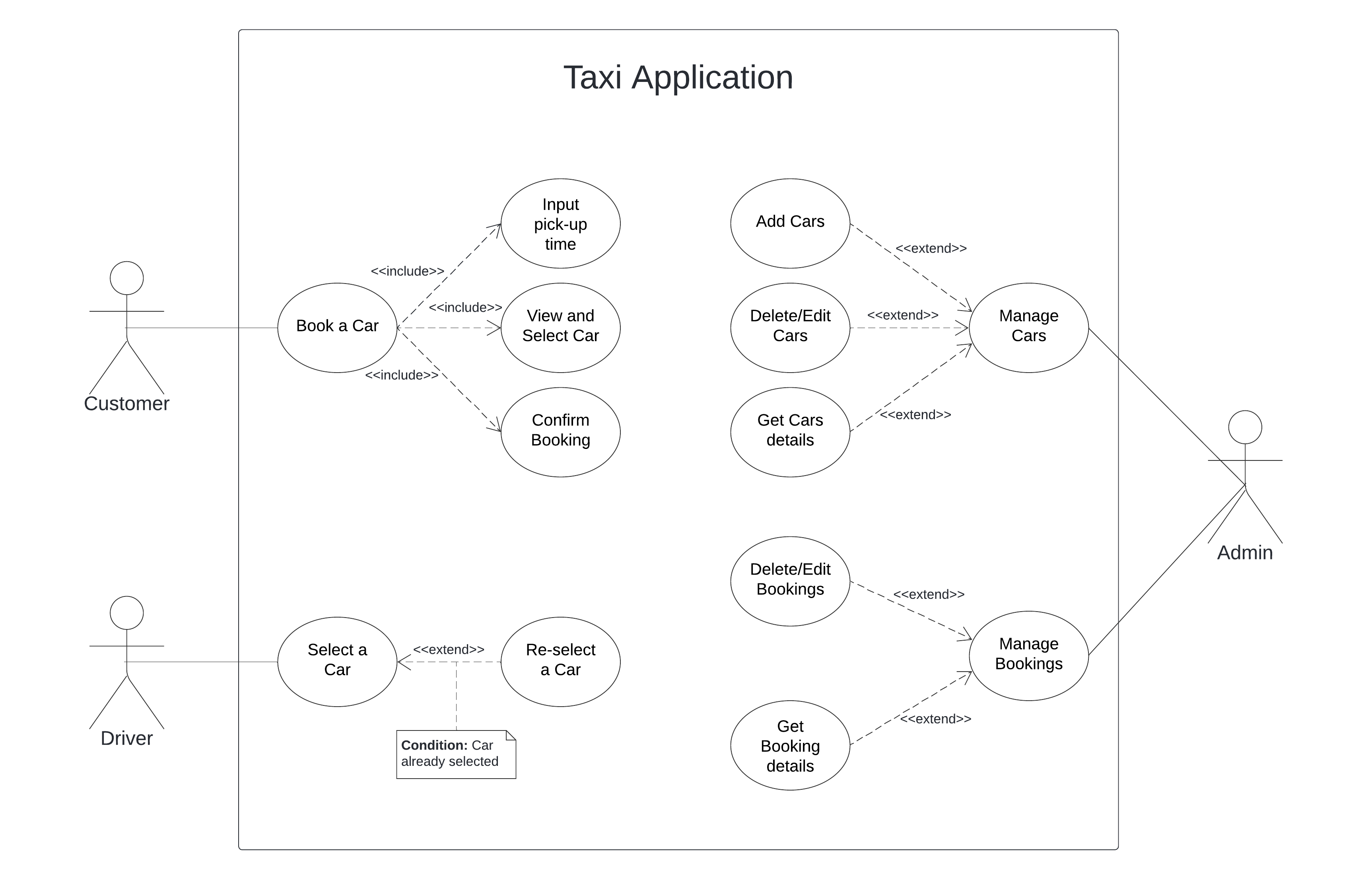
Several requirements were explicitly put forward for the implementation of this solution, including:

1. A car booking feature for customers. This consists of 3 steps – displaying all available cars, creating a booking, and finishing the booking (when the trip ends)
2. A car selecting feature for drivers that could check if the car has already been selected or not
3. All entities are to have an ID with type Long and a dateCreated with type ZonedDateTime
4. All APIs are to allow pagination
5. Basic CRUD operations for all entities, including adding, searching, deleting, and updating.
6. Customers can also be searched by name, address, and phone.
7. Bookings and invoices can be filtered by a specific date or by a date range
8. Invoices of a customer or driver can be returned filtered by a date range
9. Revenue of a specific date range can be returned. This includes revenue from all invoices, revenue from a specific customer, and revenue from a specific driver.
10. The number of days the cars were used during a specific month can be calculated and return.

All the above requirements have been successfully designed and implemented in the final solution. Please refer to the README.md file in the project directory for more information regarding how to test and use the implemented REST API to perform the requirements.

# Use case diagram

*Disclaimer: This use case diagram only represents the different use cases our actors would perform using a complete solution with a frontend. This does not reflect the design and mapping of the REST API.*



# Technology used

For this project, a number of technologies have been utilised, ranging from development software to project dependencies.

1. IntelliJ: an Integrated Development Environment (IDE) for Java development. This software is where the source code of the backend is created and edited. The license for this comes with the Student’s RMIT account.
2. Postman: an API development platform where developers could design, build, test, and iterate their APIs. This tool provides free cloud syncing for my workspace and is where the API Requests are saved and tested. The license for this is free for basic testing and collaborating.
3. MySQL: an open-source relational database management system for creating, managing, and developing databases. This suite is used to create a local database for developing the CRUD operations. The license for this is free for the Community release.
4. Spring Framework: an open-source application framework and inversion of control (IoC) container for the Java platform. This framework is used to majorly reduce the time and work needed to create an enterprise backend. As it is open source, it is free for any to use.
5. Dependencies:

* Spring Web: a dependency for creating RESTful APIs and applications using Spring MVC with Apache Tomcat as the default embedded container.
* Spring Data JPA: a dependency for managing relational databases using Object-Relation Mapping (ORM). This allows me to access and persist data without having to handle the SQL side of things.
* Lombok: an annotation library that reduces the need for boilerplate code such as getter and setter. This allows faster operation with entity classes.
* MySQL Driver: a dependency to work with MySQL JDBC and R2DBC
* JUnit: the main unit testing framework.
* Mockito: a mocking framework for unit testing services.

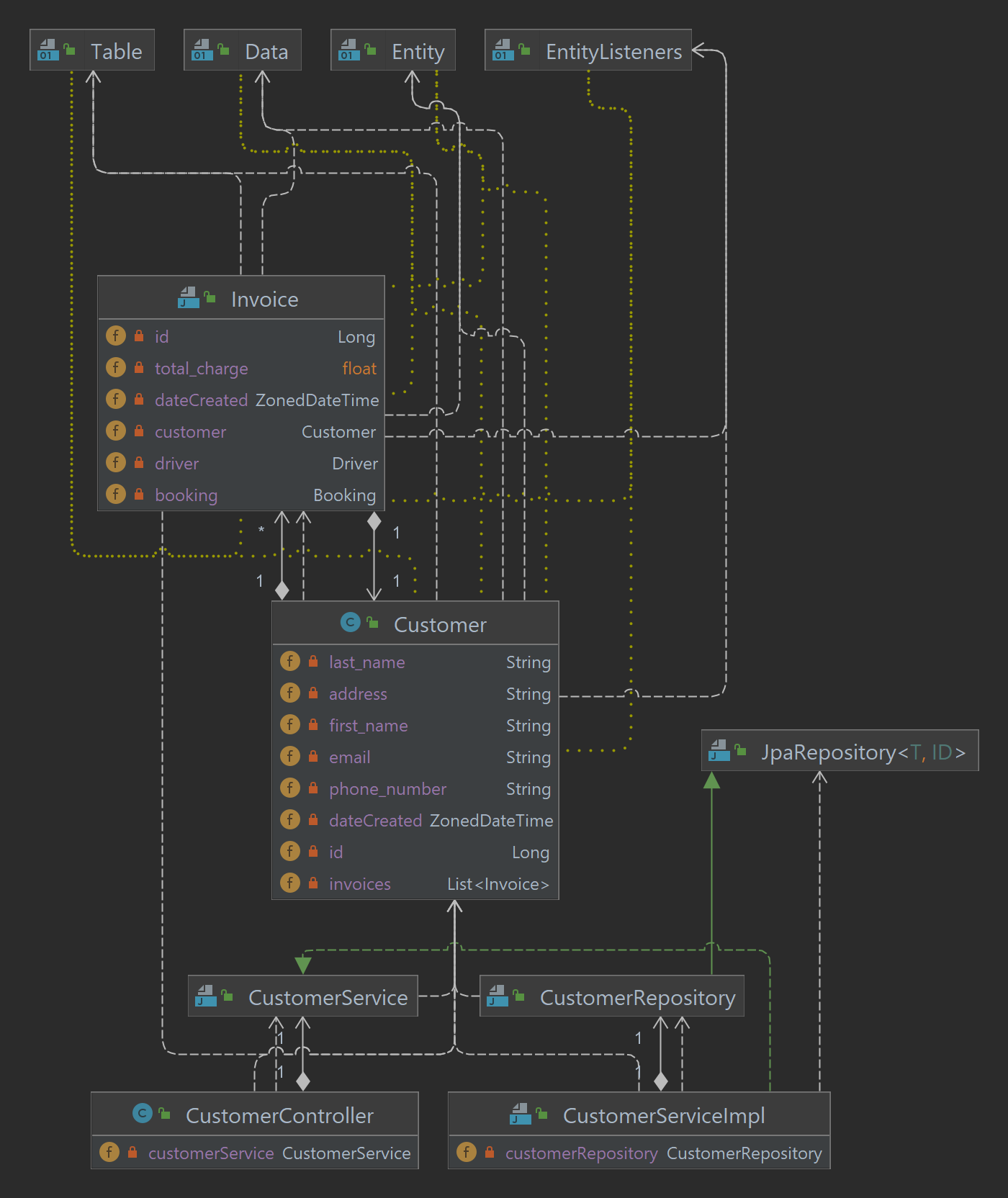
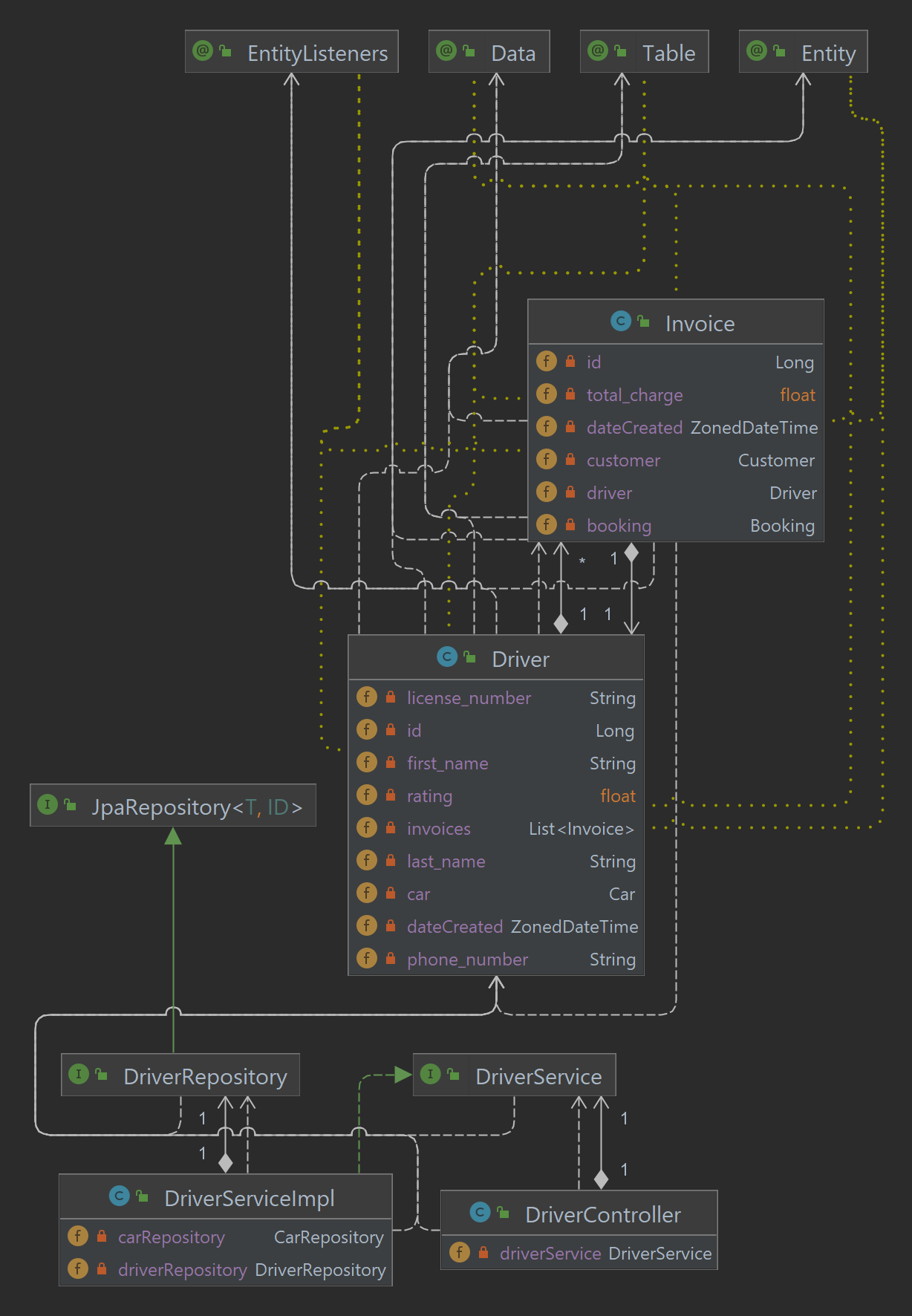
# Architecture

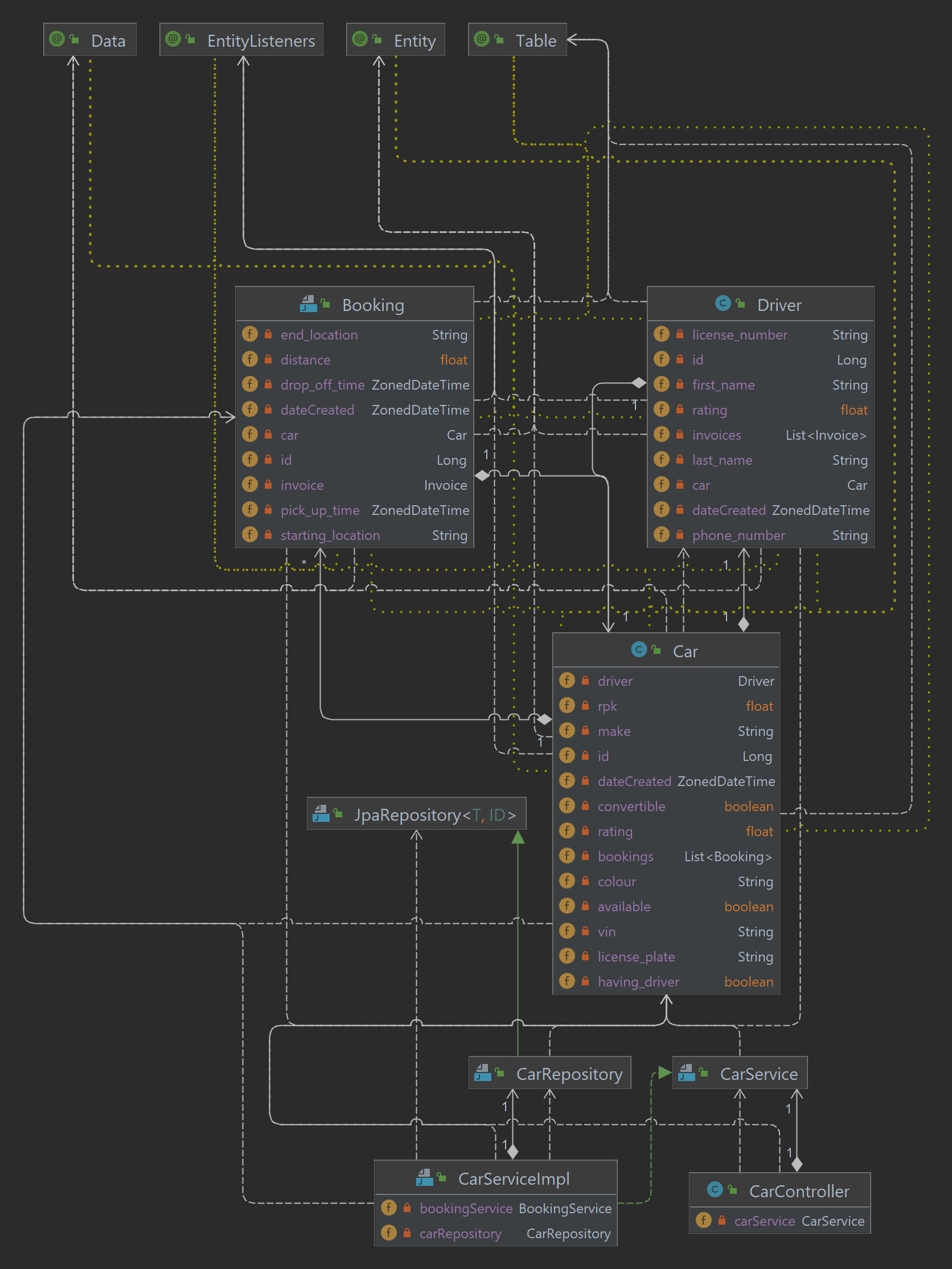
The operation of my implementation consists of 5 layers:

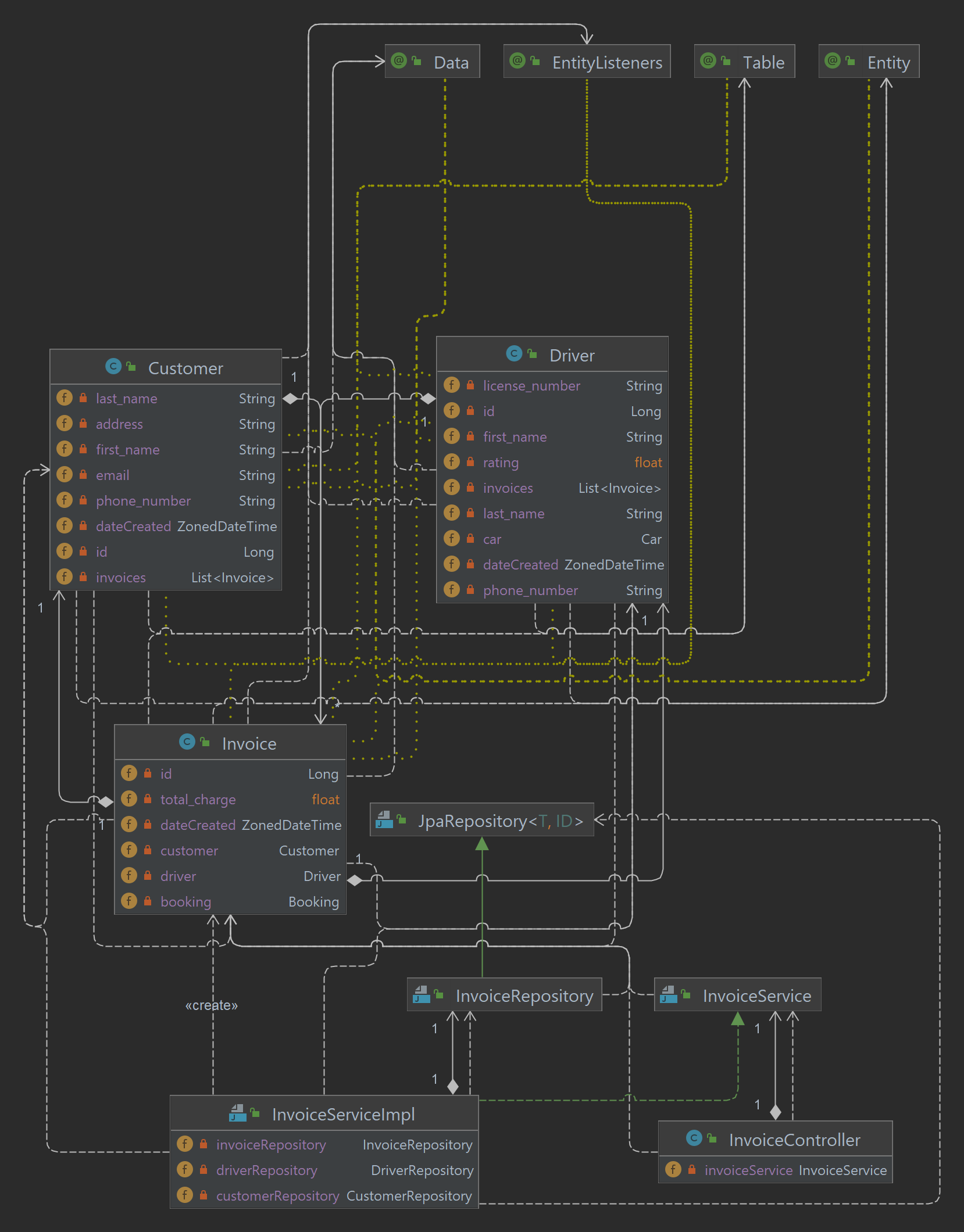
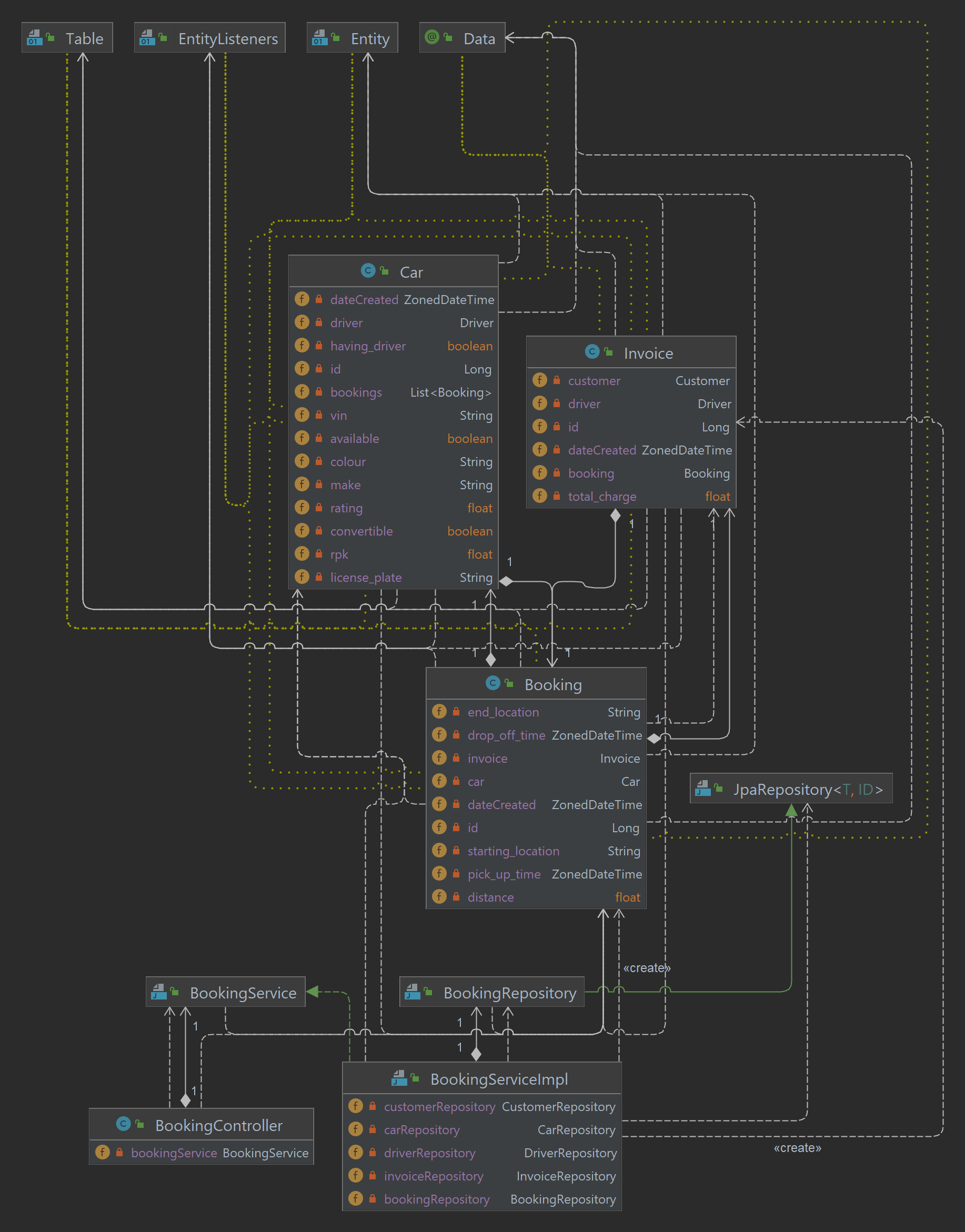
* Postman Client: This is where the REST API requests would be sent to the Controller and where we receive returns from the Controller layer. All of the required functionalities can be performed using Postman.
* Controller: This is the layer where API request mappings are created so the solution can be accessed using API requests through API clients. The Controller layer contains little logic aside from mapping.
* Service: This is the layer where all business operations and logic of the solution is declared and constructed. This layer will interact with the Hibernate Data Persistence layer to influence the MySQL Database
* Hibernate (DAO): This is the Persistence Logic Layer where complex underlying CRUD operations are turned into more intuitive Java methods, allowing easy interaction with the Database without the need for SQL Queries.
* MySQL Database: This is the endpoint of the solution, the Database Layer, where all of our records are stored.

# Class diagram

As the relationship between all classes would be too convoluted to draw in a single diagram, I have decided to create 5 different class diagrams for the 5 main entity packages. Furthermore, only directly related classes would be displayed per diagram. For example, on the Invoice diagram, as the Invoice entity does have a Driver attribute, the Driver class would also be included in the diagram. Inside the Driver class also exists a relationship with a Car class, but since there are no direct relationship between the main diagram entity (Invoice) and the Car class, it will not be displayed on the diagram.

1. **Customer Class Diagram**
2. **Driver Class Diagram**
3. **Car Class Diagram**



1. **Invoice Class Diagram**
2. **Booking Class Diagram**

# Implementation result

In summary, my implementation consists of 6 main packages, with 5 being the packages relating to our 5 main entities (booking, car, customer, driver, and invoice) and 1 being an exception package. Inside the exception package is an exception class called *ResourceNotFoundException.java* that will be thrown when the user requests for an object that does not exist in the database. In the 5 main packages, there exists 5 files:

* *Entity.java*: The main object class with all the basic attributes
* *EntityRepository.java*: This is essentially the class where we implement the persistence layer. This class simply extends the *JpaRepository* class.
* *EntityService.java*: This is an interface for all the services we would want to implement the logic for.
* *EntityServiceImpl*.*java:* This is the implementation of the services logic declared in the interface. (**NOTE:** The reason why there exists a separate file for the implementation is because it allows easier testing as well as letting me achieve loose coupling between the Controller and the Service layers)
* *EntityController.java:* This is the class that is responsible for API requests construction and mapping. All API requests are declared in these classes.

Regarding the functional requirements of the solution, the associated implementation as well the API Requests and URLs are as follows:

**Requirement a:**

Step 1: [GET REQUEST]: http://localhost:8080/api/cars/allAvailable

🡪 This gets all currently available cars

Step 2: [POST REQUEST]:   
http://localhost:8080/api/bookings/create?cus\_id=&car\_id=&start=&end=&distance=

Param meaning: **cus\_id**: the ID of the customer booking the car

**car\_id** being the ID of the car being booked

**start** being the starting location

**end** being the end location (destination)

**distance** being the distance of the trip

🡪 This creates a booking and an associated invoice based on the parameters above.

Step 3: [PUT REQUEST]: http://localhost:8080/api/bookings/finishBooking?booking\_id=

Param meaning: **booking\_id**: the ID of the booking being completed

🡪 This will set the drop off time to the time of the request execution and set the car availability to true

**Note:** While it was mentioned in the specification that the Customer would first input the pick-up datetime before all available cars are displayed, this step would be done in the frontend and not by interacting with the REST API. Therefore, no specific requests were created specifically for inputting datetime, instead the pick-up time would be automatically assigned when the booking is made (this could of course be changed to work with the frontend in the future should it be needed).

**Requirement b:**

[PUT REQUEST]: http://localhost:8080/api/drivers/{driver\_id}/bookCar?car\_id=1

Path variable meaning: **{driver\_id}**: the ID of the driver booking the car

Param meaning: **car\_id:** the ID of the car the driver would like to select

🡪 If the car’s *available* attribute is false, an exception would be thrown. If not, this PUT request will link the car to the driver and set its *having\_driver* and *driver* attributes to “*true*” and the driver with {driver\_id} respectively.

(Bonus) [PUT REQUEST]: http://localhost:8080/api/drivers/{driver\_id}/returnCar

Note: This put request takes in a Path Variable which is the ID of the driver booking the car

🡪 If the driver with ID {driver\_id} has a car, this method will unlinks the 2 objects

**Requirement c:** All entities have ID of type Long and dateCreated of type ZonedDateTime. These attributes will be generated automatically when the object is created.

**Requirement d:** Pagination is implemented for all entities

[GET REQUEST] http://localhost:8080/api/entities?page=

Param meaning: **page**: the page the user wants to get

🡪 This returns the page the user wants to get. Each page displays a maximum of 5 objects at the same time and the page index starts at 1.

**Note:** Replace*entities* by the plural of the entity. For example, for customers, the URL would be …/api/customers?page=

**Requirement e:** Basic CRUD operations is implemented for all entities

[POST REQUEST]: http://localhost:8080/api/entities (Adding into the DB)

🡪 This gets data from a .JSON input, converts it into the object we are posting, and adds it to the database

[PUT REQUEST]: http://localhost:8080/api/entities/{entity\_id} (Updating data to the DB)

Path variable meaning: **{entity\_id}**: the ID of the entity the user wants to update

🡪 This gets data from a .JSON input, converts it into an object, and updates the entity with ID *{entity\_id}* with data from this new object.

[GET REQUEST]: http://localhost:8080/api/entities/{entity\_id} (Getting data from the DB)

Path variable meaning: **{entity\_id}**: the ID of the entity the user wants to get

🡪 This gets the entity with *{entity\_id}* from the database

[DELETE REQUEST]: http://localhost:8080/api/entities/{entity\_id} (Deleting from the DB)

Path variable meaning: **{entity\_id}**: the ID of the entity the user wants to delete

🡪 This deletes the entity with *{entity\_id}* from the database

**Note:** Replace*entities* by the plural of the entity. For example, for customers, the URL would be …/api/customers…

**Requirement f:**

[GET REQUEST]: http://localhost:8080/api/customers/search?id=&name=&phone=&address=

Param meaning: **id:** the Customer’s ID

**name**: the Customer’s name

**phone:** the Customer’s phone number

**address:** the Customer’s address

🡪 This will return customers based on the parameters inputted by the user. If there is an id, the system will prioritise searching by ID (as it is unique, making all other parameters unnecessary). If there is no ID provided, the system will use any inputted parameters to search for the customer.

**Requirement g:**

[GET REQUEST]: http://localhost:8080/api/bookings/searchAll?date=&start\_date=&end\_date=

(or) http://localhost:8080/api/invoices/searchAll?date=&start\_date=&end\_date=

Param meaning: **date**: the specific date to return bookings/invoices from

**start\_date:** the starting date of the date range to return bookings/invoices from

**end\_date:** the ending date of the date range to return bookings/invoices from

🡪 This will return invoices/bookings based on the parameters inputted by the user. Using only the *date* param, the user will get results from a specific date. Using *start\_date* and *end\_date*, all records between that period will be returned. If all 3 params are inputted, only results by *date* is returned.

**Note:** Date input must be under the following format “dd+MM+yyyy”. For example 01/01/2022 would be 01+01+2022

**Requirement h:**

[GET REQUEST]:

http://localhost:8080/api/invoices/searchByID?customer\_id=&start\_date=&end\_date=

(or) http://localhost:8080/api/invoices/searchByID?driver\_id=&start\_date=&end\_date=

Param meaning: **customer\_id**: the ID of the customer from whom revenue is calculated

**driver\_id:** the ID of the driver from whom revenue is calculated

**starting\_date:** the starting date of the date range to return the invoices from

**end\_date:** the ending date of the date range to return the invoices from

🡪 This will return invoices based on the parameters inputted by the user. Depending on whether the *customer\_id* or the *driver\_id* param is inputted, the invoices will be searched accordingly and filtered through the date range. If both params are used, an exception will be thrown.

**Requirement i:**

[GET REQUEST]:

or http://localhost:8080/api/invoices/searchByID/revenue?start\_date=&end\_date=

or http://localhost:8080/api/invoices/searchByID/revenue?driver\_id=&start\_date=&end\_date=

or http://localhost:8080/api/invoices/searchByID/revenue?customer\_id=&start\_date=&end\_date=

Param meaning: **customer\_id**: the ID of the customer whose invoices to be returned

**driver\_id:** the ID of the customer whose invoices to be returned

**starting\_date:** the starting date of the date range to return the invoices from

**end\_date:** the ending date of the date range to return the invoices from

🡪 This will return invoices based on the parameters inputted by the user. Depending on whether the *customer\_id* or the *driver\_id* param is inputted, the invoices will be searched accordingly and filtered through the date range. If both params are used, an exception will be thrown.

**Requirement j:**

[GET REQUEST]: http://localhost:8080/api/cars/getMonthlyStatistics?month=

Param meaning: **month:** the month we want to get car usage statistics from

🡪 This will return all cars used within a month

**Note:** Date input must be under the following format “MM+yyyy”. For example 01/2022 would be 01+2022

# Unit Testing

For this project, JUnit testing classes have also been created. Running all tests from DriverServiceImplTests.java, BookingServiceImplTests.java, CustomerServiceImplTests.java, CarServiceImplTests.java, and InvoiceServiceImplTests.java, a 100% code coverage is achieved, save for the Controller classes. This is due to the fact that in order to test the controller classes, a full Integration Test would need to be designed. Since only Unit Testing were required, no tests for the Controller Classes were created.

# Limitation, known bugs

As all known bugs have been eliminated during the development process, no other bugs are known to be reported. Furthermore, all functional requirements were met, therefore it is believed that the solution’s capability is within the expected standard, and is only limited in aspects that were not required or specified.