CAR BOOKING PROJECT

Further Programming

Team members

S3748874 - Tran Son Phat
S3916151 - Cao Ngoc Son
S3802460 - Nguyen Minh Phu

RMIT UNIVERSITY

Introduction

A car booking agency is a company that rents automobiles to the public for a few hours or a few weeks at a time. These types of services are primarily located near an airport to provide support to customers who need a temporary vehicle when the travel of town or simply when their car broke down.

Car booking companies often have dedicated website through which online reservation can be made, and in this project, we aim to create a prototypical RESTful service as a backend of such websites. The team will emphasize on fulfilling all the business requirements of the project, while ensuring that the code base is clean, maintainable, and extensible using measure such as SOLID principles, common design patterns and rigorous testing.

Business requirement

Scope model

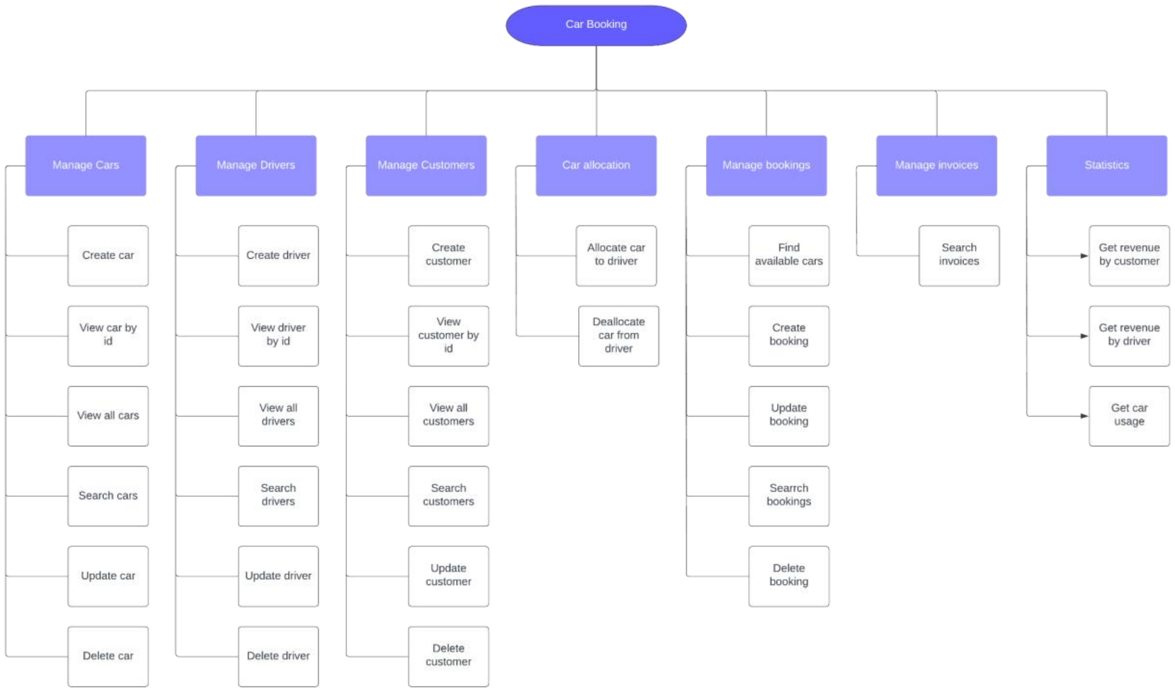

Figure 1: Scope model

In-scope items

Requirement	Description	Priority
Manager cars		
Create a car	Only the admin can create a car with various	High

	attributes, namely identification number, make, model, convertible, rating, license plate and rate per kilometer.	
View a car by id	Only the admin can view a car using its id.	High
View all cars	Only the admin can view all cars in the system.	High
Search cars by their attributes	Only the admin can search cars using its attributes	High
Update a car	Only the admin can make change to the car’s information	High
Delete a car	Only the admin can delete a car from the system	High
Manage customers		
Create a customer	Normal user and Admin can create a customer record on the system with attributes: first name, last name, address, and phone number	High
View a customer by id	Only the admin can view a customer using his/her id	High
View all customers	Only the admin can view all the customers in the system	High
Search customers by their attributes	Only the admin can search customers using their attributes	High
Update a customer	Normal user can update his/her customer information. Admin can make change to any customer’s information	High
Delete a customer	Normal user can delete their own record from the system. Admin can delete any records from the system	High
Manage driver		
Create a driver	Normal users can register as a user and create their driver record on the system. Admin can create drivers on the system	High
View a driver by id	Only the admin can view a driver by his/her id.	High

View all driver	Only the admin can view all the drivers on the system	High
Search drivers by their attributes	Only the admin can search drivers by their attributes	High
Update a driver	Driver can update their own information on the system. Admin can make change to any driver's information on the system	High
Delete a driver	Driver can delete their record on the system. Admin can delete any drivers from the system	High
Manage car allocation		
Allocate a car to a driver	Only the admin can allocate empty car to available driver. NOTE: ● Car must have no driver before allocation ● Driver must have no car before allocation	High
Deallocate a car from a driver	Only the admin can deallocate a car from a driver NOTE: ● A car must have a driver for it to be deallocated	High
Manage bookings		
Find available cars for booking	Customers and Admin can find all cars that are available to book within a period. NOTE: ● An available car is defined as a car with a driver and has no overlapped bookings within a period	High
Create booking	Customers and Admin can make bookings with the following information: starting location, destination, pick-up time, drop-off time, distance of the trip and an invoice. An invoice contains information such as the customer id, driver id and total charges NOTE: ● A car must have an id for it to be	High

	booked ● The customer must have no overlapped bookings within the pick-up and drop-off time of the booking ● The driver must have no overlapped bookings within the pick-up and drop-off time of the booking	
Update booking	Customers and Admin can make change to the booking information.	High
Delete booking	Customers can delete their own bookings. Admin can delete any bookings on the system	High
Find bookings within a start and end time	Only the admin can find all the bookings within a period NOTE: ● The search is INCLUSIVE, meaning that a booking's pick-up and drop-off time must be WITHIN the start and end time (of the search) for it to be in the result.	High
Invoice		
Find invoice within a start and end time	Only the admin can find all the invoices within a period NOTE: ● The search is INCLUSIVE, meaning that an invoice's pick-up and drop-off time must be WITHIN the start and end time (of the search) for it to be in the result.	High
Statistics		
Get revenue from a customer within a period	Get the sum of the revenue of all bookings made by a customer within a period. NOTE: ● The sum is INCLUSIVE, meaning that a customer's booking's pick-up and drop-off time must be WITHIN the start and end time (of the search) for it to be added to the sum.	High
Get revenue from a driver within a period	Get the sum of the revenue of all bookings made with a driver within a period.	High

	NOTE: ● The sum is INCLUSIVE, meaning that a driver's booking's pick-up and drop-off time must be WITHIN the start and end time (of the search) for it to be added to the sum.	
Find car usage in a month	Find the number of days each car is used within a month, for example, January of 2022.	High
Documentation		
Interactive documentation	An interactive documentation of the service endpoints where front-end developers can view the specification of the API and experiment with the endpoint calls.	Medium

Table 1: In-scope items

Out-of-scope items

Requirement	**Description**	**Priority**
Front-end	A full-fledge front-end of the system which makes use of this RESTful API will NOT be included in this project	Low
Authentication and Authorization	Although the system exhibits three distinct roles: Customer, Driver and Admin, there will NOT be any authentication or authorization implemented for these roles, since it will further complicate and lengthen the project.	Low

Table 2: Out-scope items

Use case diagram

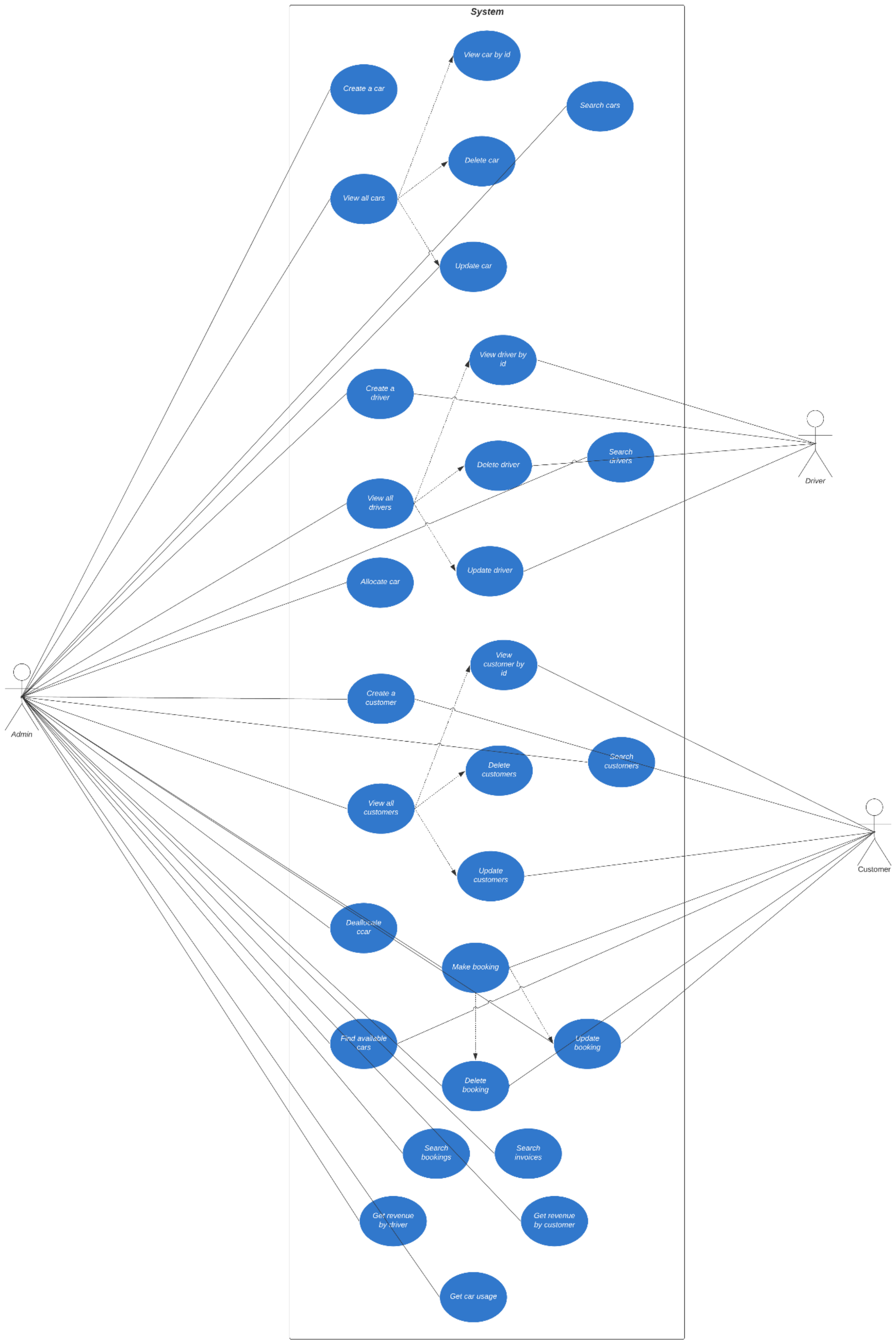

Figure 2: Use case diagram

Technology use

ID	Technology used	Description
Development		
1	Spring Boot	Spring Boot provides the team with a set of tools that can be used out-of-the-box to quickly create and test web services.
2	MySQL Database	MySQL is chosen to be the production and testing database for the car booking service. This is primarily because all members of the team are acquainted with this SQL database through the RMIT's Database course.
Testing		
4	JUnit	JUnit is the de-facto unit testing framework in Java and in this project, it will be our main tool for performing unit testing.
5	Mockito	The framework allows developers to create test double objects (mock) which will be a great addition to JUnit for unit testing. In this project, Mockito will be used to create mock version of DAOs (Data Access Object) to test higher layers such as services and controllers. This will greatly reduce the testing time, as we avoid making connections to the database.
6	AssertJ	AssertJ is yet another unit testing framework that will be utilized in tandem with our main testing framework, JUnit. We choose to use this framework due to a handful of useful testing utilities that JUnit does not provide, such as strong typed assertion and asserting if a collection contains items with specific attributes.
7	Postman	
Version control and Deployment		
8	AWS RDS	To run the code on a local machine, it is a requirement

		that the machine has a running MySQL server (for hosting the database) and client (for initializing the databases and users) instances. The team aims to eliminate this dependency by including a cloud MySQL database in our project, so the code can be run without the hassle of local database initialization.
9	Git and GitHub	As every team member is familiar with GitHub, we choose it as the main code version control tool.

Table 3: Tools and Technologies

Architecture

Overview

Throughout the design of the system, one recurrent theme can be observed: the system's business layer is broken up into small, distinct layers which are responsible for different tasks. This pattern is called Controller-Service-Repository which is quite common in the realm of web service development.

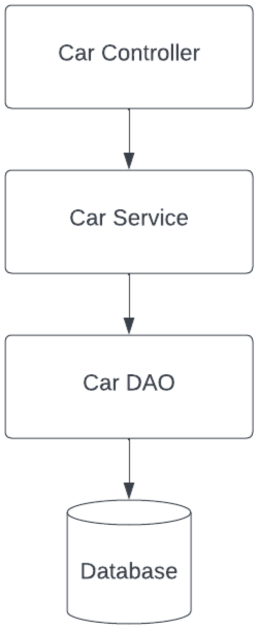

Figure 3: Architecture

In the design of the car booking system, the team chose to separate the business layer into three different components:

- **DAO (Data Access Object)**: This layer handles to reading and writing of information out of and into the databases. In more traditional designs, the layer is called Repository which is database-specific and directly communicates with the database below.

However, the team replaced it with DAOs, abstracting away the database logic and making the system modular. For instance, we can replace MySQL DAO with DAO of different database type, such as H2 or PostgreSQL without disrupting the system.

- **Service**: This is the intermediate layer to which data access objects will be injected. These classes are responsible for querying from different databases, using DAOs and combining these data to more complex business objects. They also introduce a layer of abstraction between controllers and DAOs so that these layers can be changed independently.

- **Controller**: The main entry of the web service. This layer is responsible for defining all the endpoints of the RESTful API, receiving inputs from clients through query parameters and request body and finally responding to the request, often with a body or a status code. It will do so by calling the methods defined in the service classes.

Another design choice the team is segregating the DAOs object into different interfaces, each reasonable for a different database query:

- **Crud DAO**: This DAO interface contains methods to perform CRUD (create, read, update, and delete) operations with the database below. Each of the methods has a one-to-one correspondence to a SQL query:
  - Find all entities: *SELECT * FROM TABLE*
  - Create entity: *INSERT INTO TABLE*
  - Update entity: *UPDATE TABLE*
  - Delete entity: *DELETE FROM TABLE*

- **Search DAO**: In the system, we often need to search for entities for a specific attribute, for example, a car with red paint or a customer with the name 'Adam'. To facilitate this common operation, the team has defined another interface that can help retrieve all the entities which fulfill a criterion. This corresponds with *SELECT * FROM TABLE WHERE {CRITERIA}* statement in the SQL language.

- **Exist DAO**: Like Search DAO, this interface is for checking if an entity with specific attribute exists in the database, and it returns a Boolean value to signify the entity's existence.

Other design Patterns

INVERSION-OF-CONTROL PATTERN

Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. Using an Inversion-of-Control framework such as Spring Boot allows our code based to benefit from several advantages:

- Ease of testing by isolating a component (like a service) and mocking its dependencies (like DAO objects in as service)
- Greater modularity of the system, as individual components can be switch with a different implementation without disrupting the system.
- Decoupling a task's execution from its implementation, as the framework will be responsible for calling the methods, not the developers.

BUILDER PATTERN

In the car booking system, there are complex models with many attributes that need to be initialized when created, such as Car or Booking Model. To simplify the object creation procedure, the team adopt the Builder pattern which helps to create these objects step-by-step.

DAO PATTERN

As previously mentioned, the DAO pattern is utilized to abstract away the implementation details of the database operations, so that different database flavors (MySQL or PostgreSQL), or different implementations of the same database can be used interchangeably, making our system more modular.

STRATEGY PATTERN

One key functionality of the system is the search and retrieval of entities which fulfill certain criteria. In fact, it was stated in the business requirement that an admin can search cars, customers, drivers, bookings, and invoices based on their attributes. Rather than hardcoding the search logic into the services, such as searchCustomerByName() or searchCarByColor(), the team applied the Strategy design pattern which enables us to extract and the search criteria into separate classes. The approach is beneficial for several reasons:

- It avoids cluttering the Service classes with many search methods.
- Developers can easily extend the search capability by defining more search criteria classes.

FACADE PATTERN

In the design of the application, the Service class plays the role of a Facade class which provides an abstracted and simplified interface to complex sets of functionalities of the data access objects. Adopting this design patterns into our system helps it gain increased modularity. For example, one team member can work on the service class, and another can work on the controller class separately, without knowing the details of each other. The latter just need to know the exposed interface of the service classes to complete his task, thereby reducing the development time.

Class diagram

To view the class diagram in more details, please refer to this URL: https://lucid.app/lucidchart/1fb7370e-065a-4011-a6e0-956de9eaf46f/edit?invitationId=inv_f76c0b76-b95a-4521-91db-ac77d3fc21af

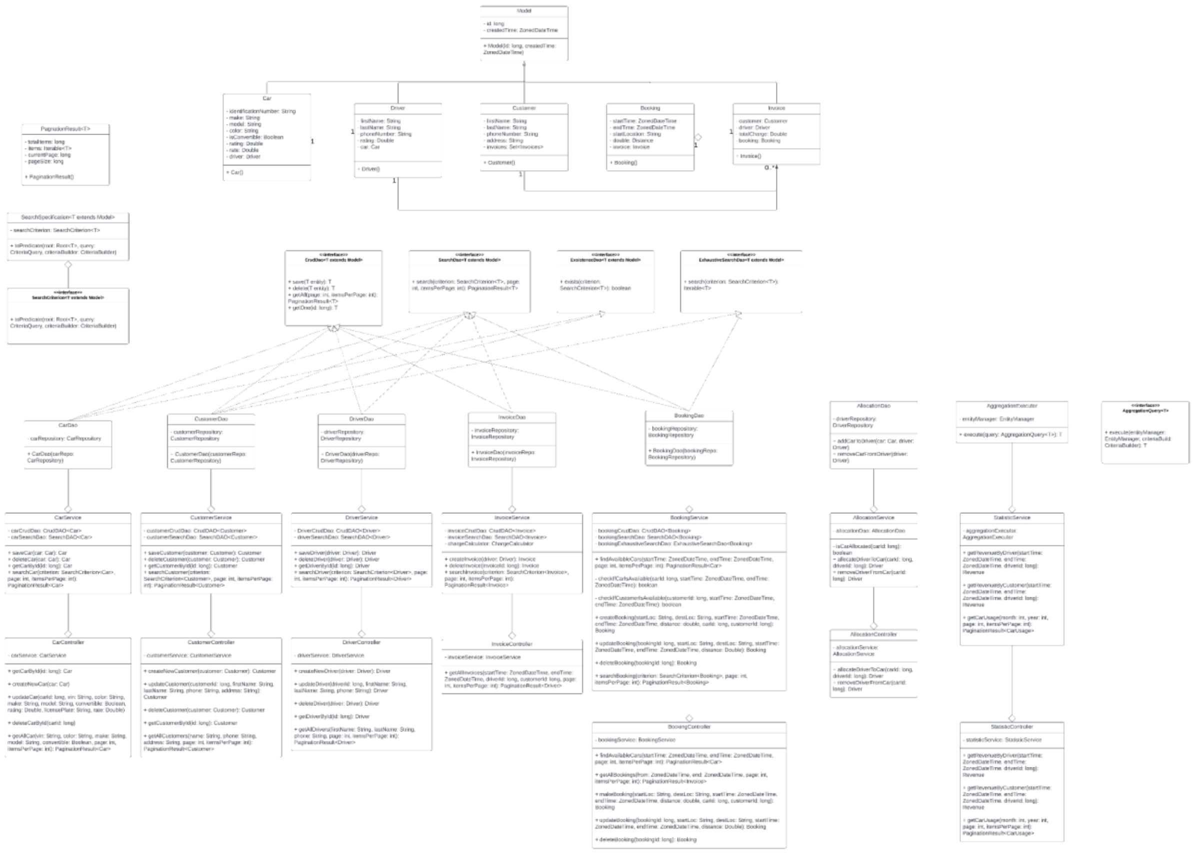

Figure 4: Class diagram

Implementation result

The result of this project is a RESTful service that fulfills all the business requirements stated at the beginning of this report. In this section, the team will provide the detailed specification of our car booking web service.

Endpoints

MANAGE CARS

Find all cars
Endpoint: **GET /cars**
Query parameters:

- size: The number of cars per page
- page: Current page to display
- identification: String

- color: String
- make: String
- model: String
- convertible: Boolean
- page: Integer, current page
- size: Integer, the number of cars per page

Note: If no search query parameter is provided, the endpoint will simply return all cars in the system.

View a car
Endpoint: **GET /cars/{carId}**
Path variables:

- carId: Integer, the id of the car

Delete a car
Endpoint: **DELETE /cars/{carId}**
Path variables:

- carId: Integer, the id of the car

Note: Deleting a car also deletes all associated bookings.

Create a car
Endpoint: **POST /cars**
Request body:

```
{
        "identificationNumber": String (required, unique),
        "make": String,
        "model": String,
        "color": String,
        "isConvertible": boolean,
        "rating": Double,
        "licensePlate": String (required, unique),
        "rate": Double (Per kilometer)
}
```

UPDATE A CAR

Endpoint: **PUT /cars/{carId}**
Path variables:

- carId: Integer, the id of the car

Query parameters:

- identification: String (required, unique)
- make: String
- model: String
- color: String

- convertible: boolean
- rating: Double
- licensePlate: String (required, unique)
- rate: Double (Per kilometer)

MANAGE DRIVERS

View all drivers
Endpoint: **GET /drivers**
Query parameters:
- name: String, The name of the driver
- phone: String, The phone of the customer
- size: Integer, The number of drivers per page
- page: Integer, Current page

Note: If no search query parameter is provided, the endpoint will simply return all drivers.

View a driver by id
Endpoint: **GET /drivers/{driverId}**
Path variable:
- driverId: Integer, The id of the driver

Create a driver
Endpoint: **POST /drivers**
Request body:
```
{
   "firstName": String (required),
   "lastName": String (required),
   "phoneNumber": String (required),
   "ratings": Double
}
```

Update a driver
Endpoint: **PUT /drivers/{driverId}**
Path variable:
- driverId: Integer, The id of the driver

Query parameters:
- firstName: String
- lastName: String
- phone: String
- ratings: Double

Delete a driver
Endpoint: **DELETE /drivers/{driverId}**
Path variable:
- driverId: Integer, The id of the driver

Note: Deleting a driver also deletes all associated bookings.

MANAGER CUSTOMERS

View/Search all customers
Endpoint: **GET /customers**
Query parameters:

- name: String
- phone: String
- address: String
- size: Integer, number of elements per page
- page: Integer, current page

Note: If search query parameters are missing, the endpoint will retrieve all customers.

View a customer by id
Endpoint: **GET /customers/{customerId}**
Path variables:

- customerId: Integer, the id of the customer

Create a customer
Endpoint: **POST /customers**
Request body:

```
{
   "firstName": String (required),
   "lastName": String (required),
   "address": String (required),
   "phoneNumber": String (required)
}
```

Update a customer
Endpoint: **PUT /customers/{customerId}**
Path variables:

- customerId: Integer, the id of the customer

Query parameters:

- firstName: String
- lastName: String
- phone: String
- address: String

Delete a customer
Endpoint: **DELETE /customers/{customerId}**
Path variables:

- customerId: Integer, the id of the customer

CAR ALLOCATION

Allocate car to driver
Endpoint: **POST /cars/{carId}/driver/{driverId}**
Path variables:

- carId: Integer, the id of the car.
- driverId: Integer, the id of the driver.

Note: To allocate successfully, the car must have no driver and the driver must have no car.

Deallocate car from driver
Endpoint: **DELETE /cars/{carId}/driver**
Path variables:

- carId: Integer, the id of the car

Note: Removing a car from a drive also remove any associated bookings.

MANAGE BOOKINGS

Find all available cars for booking
Find all cars that are available for booking during a period
Endpoint: **GET /bookings/cars**
Query parameters:

- start: String in ISO Date time format, the start of the period
- end: String in ISO Date time format, the end of the period
- page: Integer, current page
- size: Integer, the number of items in a page

Example:
GET /booking/cars?start=2020-01-06T00:00:00.000%2B07:00&end=2020-01-09T00:00:00.000%2B07:00

Note:

- Start time and end time must be in ISO Date Time string format for the endpoint to work correctly.

Make booking
Endpoint: **POST /customers/{customerId}/bookings**
Path variables:

- customerId: Integer, The id of the customer

Parameters:

- startingLocation: String
- endLocation: String
- startTime: String, in ISO Date time format
- endTime: String, in ISO Date time format
- distance: Double (kilometers)

- carId: Integer, the id of the car

Note:

- Start time and end time must be in ISO Date Time string format for the endpoint to work correctly.
- If car is not available, the call to the endpoint will result in a bad request response.

Update booking
Endpoint: **PUT /bookings/{bookingId}**
Path variables:

- bookingId: Integer, The id of the booking

Query parameters:

- startingLocation: String
- endLocation: String
- startTime: String, in ISO Date time format
- endTime: String, in ISO Date time format
- distance: Double (kilometers)

Delete booking
Endpoint: **DELETE /bookings/{bookingId}**
Path variables:

- bookingId: Integer, the id of the booking

Note: Deleting a booking also deletes its associated invoice.

Search bookings within a period
Endpoint: **GET /bookings**
Query parameters:

- from: String, in ISO DateTime format, the start of the period
- to: String, in ISO DateTime format, the end of the period
- page: Integer, the current page
- size: Integer, the number of bookings per page

Note:

- “from” and “to” parameters must both be present at the same time, or it will result in a bad request response.
- If no period is provided, the search will simply return all the bookings.

MANAGE INVOICES

View/Search all invoices
Endpoint: **GET /invoices**
Query parameters:

- from: String, in ISO DateTime format, the start of the period
- to: String, in ISO DateTime format, the start of the period
- customer: Integer, the id of the customer
- driver: Integer, the id of the driver

- page: Integer, the current page
- size: Integer, the number of items per page

Note:

- If no search query parameters are present, this endpoint will retrieve all invoices.
- “from” and “to” must be both present and it will result in a bad request response.

STATISTICS

Get revenue by a customer

Endpoint: **GET /statistics/revenue/customer**

Query parameters:

- from: String, in ISO DateTime format, the start of the period
- to: String, in ISO DateTime format, the start of the period
- customer: Integer, the id of the customer

Get revenue by a driver

Endpoint: **GET /statistics/revenue/driver**

Query parameters:

- from: String, in ISO DateTime format, the start of the period
- to: String, in ISO DateTime format, the start of the period
- customer: Integer, the id of the customer

Get car usage within a month

Endpoint: **GET /statistics/usage**

Query parameters:

- month: Integer
- year: Integer
- page: Integer, the current page
- size: Integer, the number of items per page

Testing

The car booking service is divided into three layers: DAO layer, Service layer and Controller layer, enabling the team to test each layer independently using unit tests and mock objects. Our approach for testing is as follows: firstly, we create a complete test suite for DAO classes for each of the entities; once we have confirmed that this layer behaves as expected, the team moved on to performing unit tests on a layer that is more high-level: service layer. Instead of using authentic DAO classes, we produced mock objects of these classes using a mocking library to avoid making connections to the test database. Finally, our method for testing controller layer is utilizing mock objects and another tool provided by Spring framework, MockMVC. Below are the detailed steps by which the web service is tested

PRODUCING TEST DATABASE

For testing the DAO classes, which comminutes directly with the database, we first create a test database with the same tables and structure as the production database, and populate it with several entities including cars, customers, drivers, bookings, and invoices. This is to ensure the integrity of the production database and reduce the testing time for DAO classes (since the test database only has some entities and therefore, it is more lightweight). Initially, the team intended to use an embedded database, such as H2, to further reduce the testing time but due some configuration errors, we finally settled with a test MySQL database.

TESTING DAO CLASSES

In the next step, we moved on to producing unit tests for DAO classes, including the CrudDAO, SearchableDAO, and ExistenceDAO for each of the entities. In each of the test suites, the team rigorously test all the methods for all the DAO interfaces, namely saving an entity, getting an entity by id, getting all entities, deleting an entity, searching for an entities and check if an entity with specific attribute exists. One important notion for this section is the use of Spring Profiles for specifying the production database and test database, as well as the @Transactional and @Rollback annotation to preserve the initial state of the test database.

TESTING SERVICE CLASSES

After the data layer has been confirmed to behave expected, the next step in the procedure was testing the business logic of the service classes. In this phase, the team made extensive use of Mockito - a mocking framework that allowed us to create mock versions of the DAO classes and avoid making connections to the database.

Firstly, we defined the mock DAO objects using the *@MockBean* annotation, which helps to initialize the mock object based on the classes and inject the mock directly into the test. In each of the unit test, the behaviors of the mock objects were defined manually using Mockito's *when().thenReturn()* syntax. With these behaviors defined, we tested the business logic of the service layer to verify if it produces expected results.

TESTING CONTROLLER CLASSES

Finally, in the controller tests, the team utilized another tool provided by Spring Mvc framework: MockMVC. This special object, which is already included in the Spring context when the dependency *spring-boot-starter-test* is added to the project's *pom.xml* file, helps us to simulate HTTP requests to endpoints in the controllers without having to rely on tools such as Postman.

With the MockMVC object and the mock Service objects injected to the controller tests, the endpoints were verified to return correct response and status code based on our given query parameters and request body.

Limitation and known bugs

LACK OF AUTHENTICATION AND AUTHORIZATION

The system is designed to consist of three types of users: drivers, customers, and admin, each with their own roles and authorization. For instance, the admin has total control over the system and may delete any cars, customers, drivers, and bookings at will, whereas the drivers and customers can modify their own information. Currently, there is no mechanism to authenticate these users and authorize them to execute operations based on their role (role-based authorization) and hence, the entire service's endpoints are exposed to an unauthorized user without any roles in the system.

If the team were to revise the design and implementation of this car booking system, we hope to complement it with functionalities for authentication (given username and password) and authorization (using roles).

LACK OF AUTOMATED CAR ALLOCATION

In its current state, the admin is responsible for manually allocating/deallocating cars to drivers. This will result in much inconvenience and confusion, especially when the system goes into production and potentially acquire millions of users. Hence, there is much need for an automated car allocation solution which can intelligently allocate cars to drivers based on their availability. However, developing this feature requires much time and research efforts, which goes beyond the current scope of the project.

LACK OF FAILSAFE MECHANISM WHEN DELETING ENTITIES

When an entity such as a car, a driver and a customer is deleted by the admin, all the associated bookings and invoices will also be removed from the system. This approach can be potentially problematic and cause much inconvenience for customers. Imagine a scenario where a driver's information is deleted from the database, his car will be driverless and as a result, all the bookings with this driver are also removed from the system. Therefore, customers with these bookings must make their bookings again to have a car for their upcoming trip.

In the team's opinion, it would be more convenient if there is a failsafe mechanism to automatically allocate free drivers to these cars and bookings so that bookings should not be made again. However, this may require a complex algorithm that is not included in the business requirement of the project.

Additional resources

The GitHub repository of the project is available at: PhatHuynhTranSon99/CarBooking: A simple RESTful API for booking car trips, written using Spring Boot (github.com)