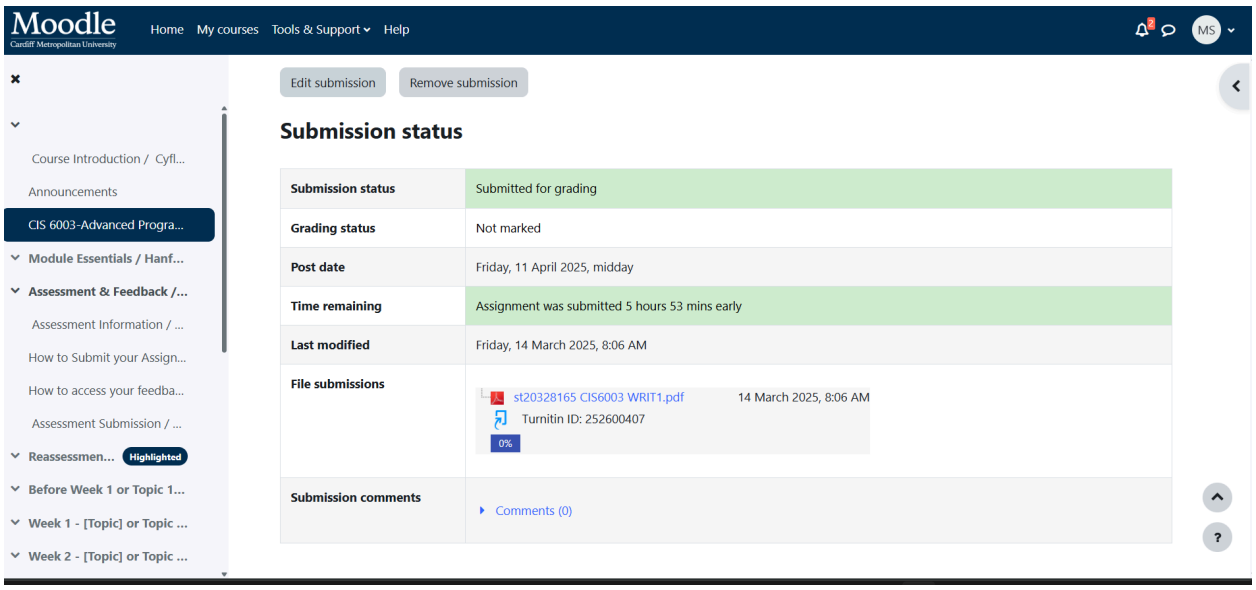
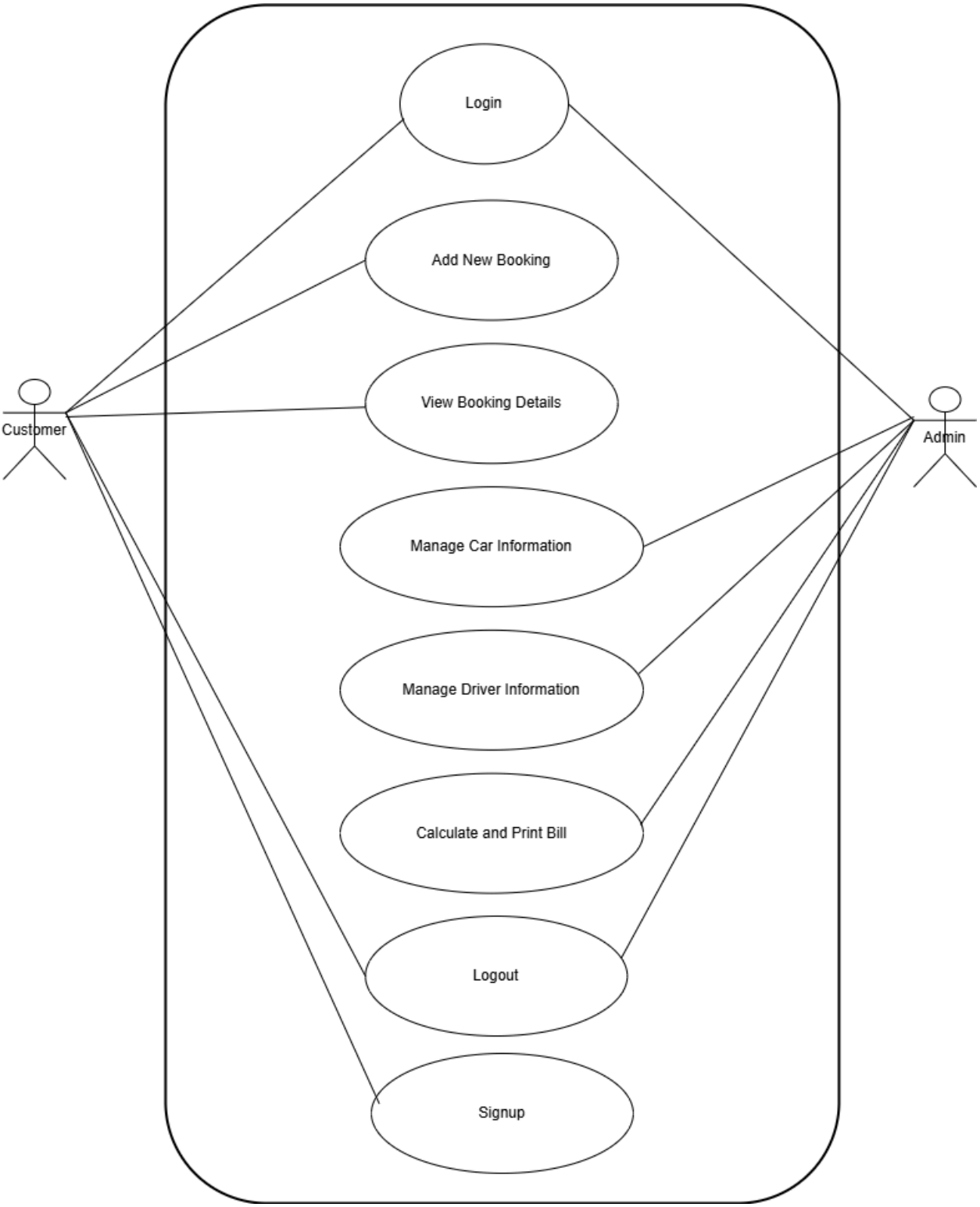
Moodle Turnitin

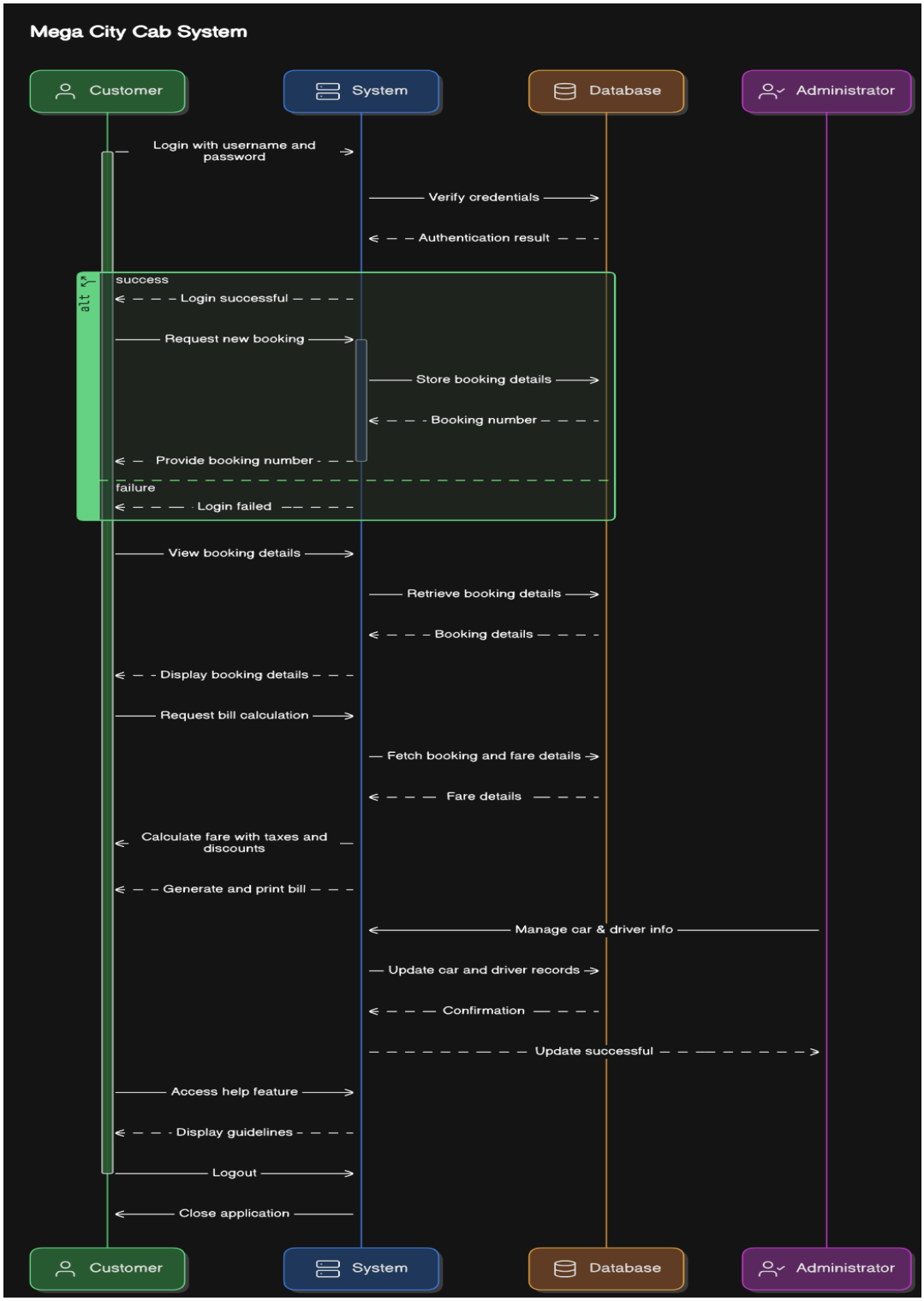


Github Link: https://github.com/SachinSilva99/megaCab

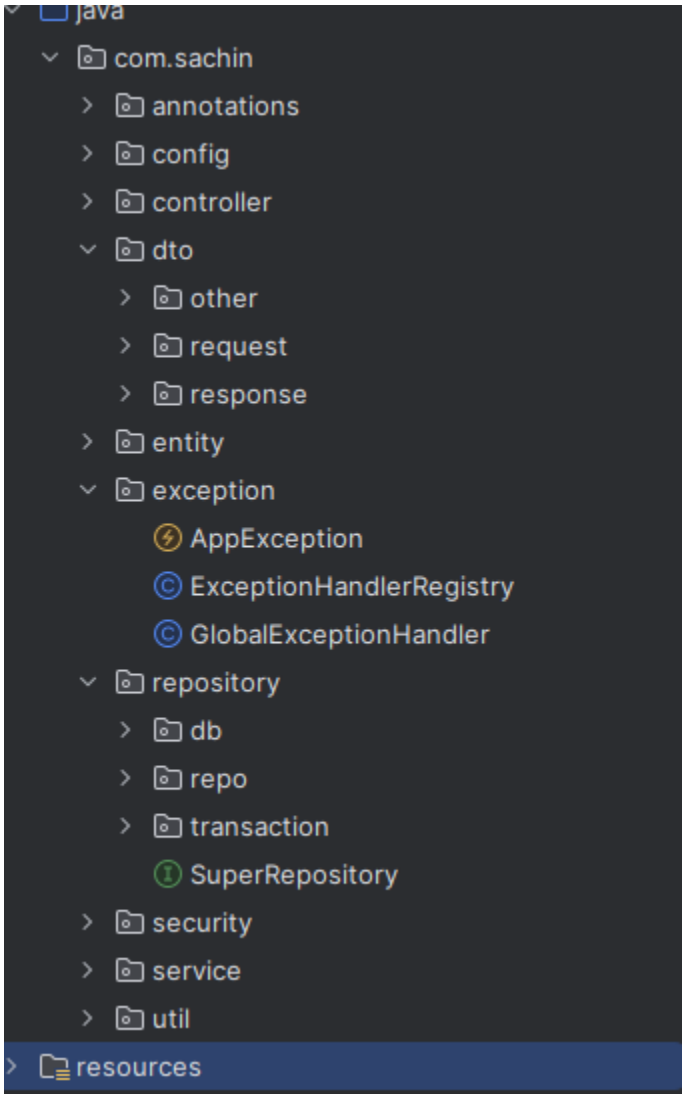
Use Case Diagram


Login
Add New Booking
View Booking Details
Manage Car Information
Manage Driver Information
Calculate and Print Bill
Logout
Signup
Customer
Admin

Sequence Diagram



Introduction Architecture



Basically, the project is made up of several layers and has been designed for easy effectuation of a single task in the preceding layers, considered layers as listed below:

1. Controller Layer

- Located in the controller package.
- Handles incoming HTTP requests and maps them to appropriate service methods.

- Acts as an interface between the frontend and backend logic.

2. Data Transfer Object (DTO) Layer

- Located under the dto package, further splitting up into request, response, and other sub-packages.
- Used to transfer data between layers whereby, of course, nothing is lost in any layer, ensuring abstraction.

3. Entity Layer

- Very much finds its way in the entity package.
- Contains database model classes that represent database tables.
- Uses JPA/Hibernate for the operation of object-relational mapping (ORM).

4. Exception Handling Layer

- It's in the exception package.
- Includes custom exception classes (AppException), wherein global handlers (GlobalExceptionHandler and ExceptionHandlerRegistry) are herein included.
- Effectively ensures centralized error handling and is a great aid in debugging.

5. Repository Layer

- Located in the repository package; further subpackage arrangement consists of db, repo, and transaction.
- Provides abstraction to database transactions.
- SuperRepository is a common interface for repository implementations.

6. Service Layer

- Located within the confines of the service package.
- It executes the business logic, orchestrating data transfers between repositories and controllers.

7. Security Layer

- It finds itself in the confines of the security package.
- Manages Authentication, Authorization, and Security configurations.

8. Utility Layer

- It resides within the util package.
- Contains Helper methods and Common reusable utilities.

9. Configuration Layer

- Defined in the config package-utilization.
- Contains application-specific configurations, ranging from database to cloud security and third-party integrations applications.

I have used only one servlet , this servlet acts as the central request handler, much like Spring's DispatcherServlet.

Key features of this implementation include:

- **Annotation-Based Request Mapping**: Registering controllers and their methods using custom @RequestMapping annotations.
- **CDI-Based Dependency Injection**: Controllers are retrieved via Contexts and Dependency Injection (CDI), ensuring decoupled component management.
- **Exception Handling Mechanism**: A global exception handler (GlobalExceptionHandler) and a registry (ExceptionHandlerRegistry) manage centralized error handling.
- **Dynamic Request Handling**: Incoming HTTP requests are matched to registered handlers, extracting request bodies when needed.
- **Header-Based Token Handling**: Extracts authentication tokens from headers for further processing.
- **JSON Serialization**: Uses ObjectMapper for JSON request/response conversion.

Dispatcher Servlet

I have created a single Servlet class for all the controllers, which is front controller design pattern.
It is designed to handle HTTP requests dynamically by mapping them to controller methods using @RequestMapping annotations. This servlet leverages CDI for dependency injection and provides structured request handling.

During initialization, the servlet registers multiple controllers such as UserController, CarController, and others using CDI. It also registers a global exception handler (GlobalExceptionHandler) and scans for @RequestMapping annotations to store request mappings in a HashMap for efficient request lookup. When a request is received, the servlet extracts the request path and HTTP method, checks if a matching method exists in the mapping, retrieves the appropriate controller instance, and invokes the corresponding method dynamically using reflection. If the method requires a request body, it is parsed using ObjectMapper before

invocation. The servlet also manages authorization by extracting tokens from the Authorization header and storing them in HeaderHolder.

The servlet includes centralized error handling via the ExceptionHandlerRegistry, which maps exception types to their respective handler methods. If an exception occurs during request processing, the registry identifies and invokes the corresponding exception handler method, ensuring that a structured error response is returned. The sendSuccessResponse() method is responsible for converting successful responses into JSON format, while the sendErrorResponse() method handles structured error messaging with an HTTP status of 500.

Controllers

Introduction

The provided controllers in the system are responsible for handling various business operations related to **Bookings, Drivers, and other domain entities**. These controllers serve as the entry point for API requests, ensuring smooth interaction between the frontend and backend.

Common Controller Structure & Functionality

1. Controller Annotations & Scope

- Controllers are annotated with @ApplicationScoped, ensuring they are managed within the application's lifecycle.
- Each controller is mapped to a specific base URL using @RequestMapping, making API routes predictable and structured.

2. Dependency Injection

- Uses @Inject for injecting the respective service classes.
- Ensures a clean separation between controllers and business logic.
- Enhances modularity and testability.

3. HTTP Methods & API Endpoints

Each controller follows a RESTful approach with the following common API patterns:

Example API patterns:

- /entity/all → Fetch all records.
- /entity/{id} → Fetch a specific record by ID.
- /entity (POST) → Create a new record.
- /entity (PUT) → Update an existing record.

---

4. Use of DTOs (Data Transfer Objects)

- All API interactions use DTOs for request and response handling.
- Enhances data encapsulation and prevents exposing internal database entities.
- Helps in maintaining flexibility while processing API requests.

Example DTOs:

- BookingRequestDTO, BookingResponseDTO
- DriverRequestDTO, DriverResponseDTO

---

5. Service Layer Delegation

- Controllers **do not contain business logic**; instead, they delegate operations to service classes.

Example:

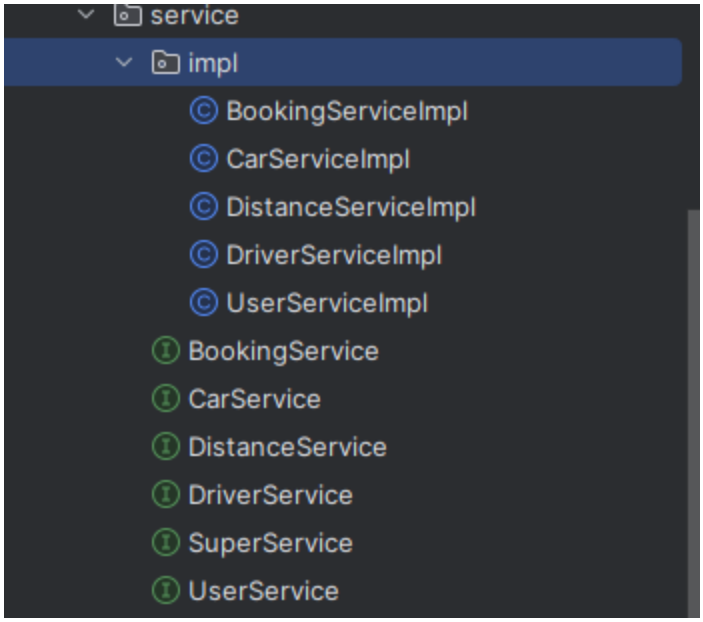
```
@RequestMapping(method = RequestMapping.HttpMethod.POST)
public ResponseDTO<BookingResponseDTO> book(@RequestBody BookingRequestDTO
requestDTO) {
   return bookingService.createBooking(requestDTO);
}
```

- This improves maintainability and scalability.

Services

Services Architecture

Overview



In this architecture, the **service layer** is responsible for implementing business logic and serves as a medium between the controllers and the data access layer. This is done by adhering to a structured interface-implementation pattern, thus helping in flexibility, maintainability, and testability.

---

1. Interface-Implementation Pattern

- **Interfaces (BookingService, CarService, etc.)**
    - Define the contract for business logic operations.
    - Provide abstraction, allowing multiple implementations if needed.
- **Implementations (BookingServiceImpl, CarServiceImpl, etc.)**
    - Contain actual business logic.
    - Implement methods defined in the interfaces.

- Can include transactional handling, data validation, and service orchestration.

---

2. Services Present

The system contains services for handling various functionalities:

- **BookingService** – Responsible for managing booking operations, including creating, retrieving, and handling booking-related logic.
- **CarService** – Manages car-related functionalities, such as adding, updating, and retrieving vehicle details.
- **DistanceService** – Handles distance-related calculations, possibly for fare estimation or route optimization.
- **DriverService** – Manages driver operations, including registration, updates, and retrieval of driver details.
- **UserService** – Handles user management, authentication, and related functionalities.
- **SuperService** – Base service, providing shared logic or utilities that other services may extend.

UserService

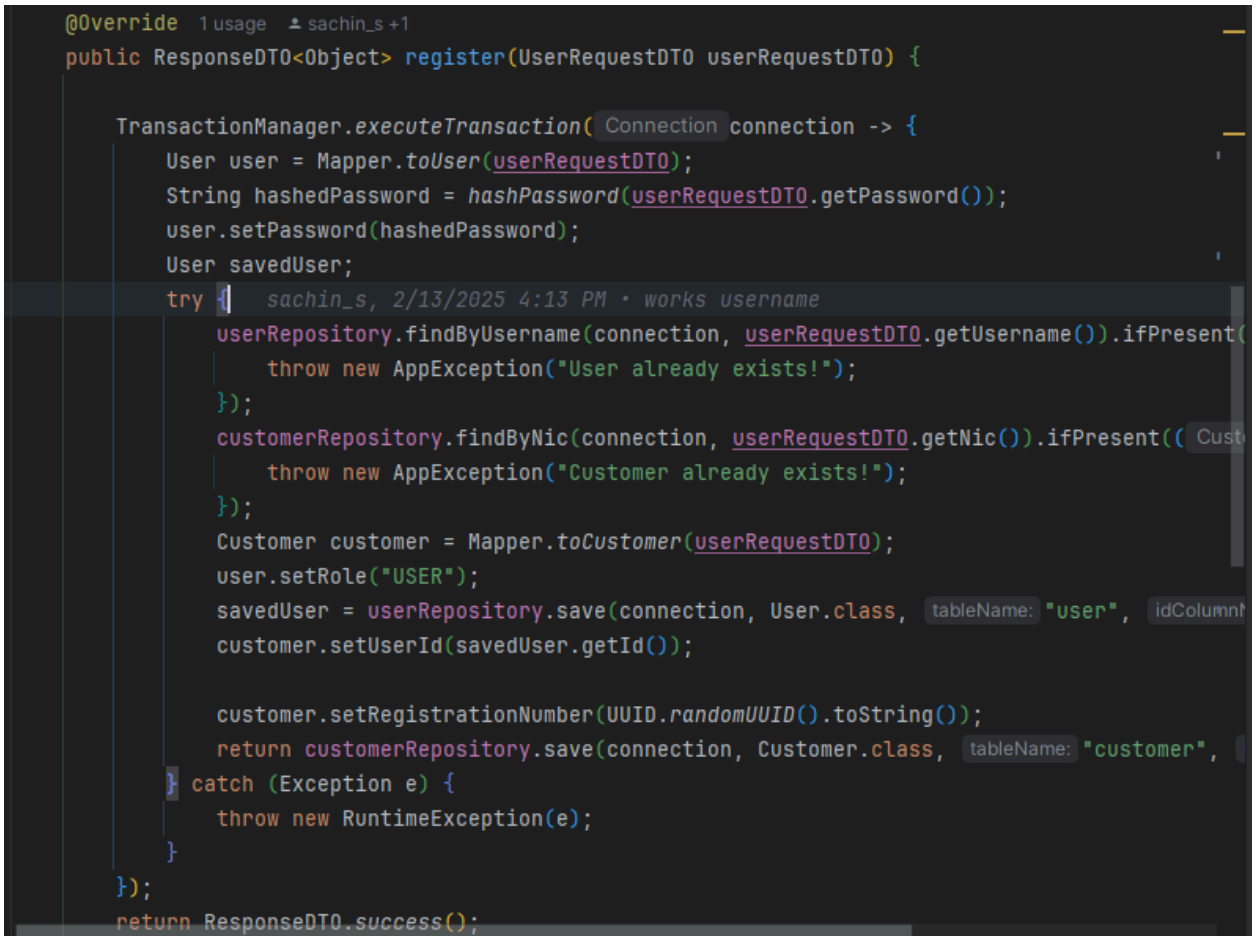
The UserService interface extends SuperService, indicating that it inherits common functionalities applicable across multiple services. This interface defines essential user-related operations, including authentication and user management.

1. **getUserLoggedInUser()** – The getUserLoggedInUser() method retrieves the currently authenticated user by first establishing a database connection. It then extracts the username from the JWT token stored in headerHolder.getToken(). Using this username, it queries the userRepository.findByUsername(connection, username), throwing an exception if the user is not found. Finally, it closes the database connection and returns the authenticated User object. This method ensures secure identification of the logged-in user.

```
@ApplicationScoped  sachin_s +1 *
public class UserServiceImpl implements UserService {

    @Inject  4 usages
    private UserRepository userRepository;
    @Inject  2 usages
    private CustomerRepository customerRepository;
    @Inject  1 usage
    private HeaderHolder headerHolder;


    @Override  no usages  sachin_s *
    public User getUserLoggedInUser() {
        try {
            Connection connection = DBConnection.getInstance().getConnection();
            String username = JwtUtil.getClaims(headerHolder.getToken()).getSubject();
            User user = userRepository.findByUsername(connection, username).orElseThrow();    sachi
            connection.close();
            return user;
        } catch (Exception e) {
            throw new AppException("User Not found!");
        }
    }
```

2. **register(UserRequestDTO userRequestDTO)** – It takes a UserRequestDTO object as input and uses a TransactionManager to execute database operations within a transaction. The method first checks if a user with the provided username or NIC (National Identity Card number) already exists in the userRepository; if so, it throws an AppException. If not, it hashes the password, maps the request to a User object, sets the role to "USER", and saves the user to the database, retrieving the saved user's ID. A Customer object is then created, linked to the saved user ID, assigned a random registration number, and saved to the customerRepository. The method handles any exceptions by throwing a

RuntimeException and returns a ResponseDTO.success() object upon successful completion.



3. **login(UserLoginRequestDTO requestDTO)** – The method establishes a database connection using DBConnection.getInstance().getConnection(). It retrieves the username and password from the request DTO. The userRepository.findByUsername method checks for an existing user; if none is found, it throws an AppException. It then verifies the password using checkPassword, comparing the input with the stored hashed password. If the password matches, a JWT token is generated with the username and role "USER" using JwtUtil.generateToken, and a LoginResponseDTO is built with the token

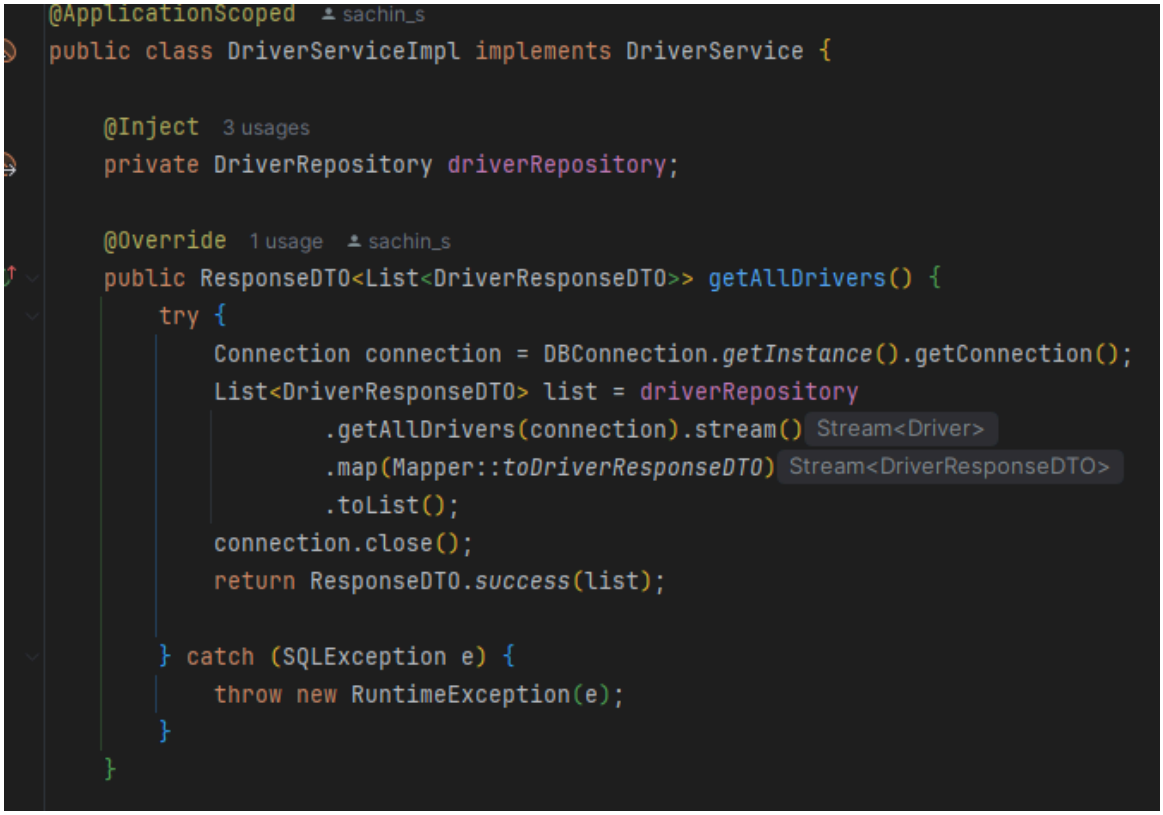
and its expiration time. The method returns a successful ResponseDTO with the login response. If the password is incorrect, it throws an AppException; any other exceptions are caught and rethrown as an AppException with a generic error message.

```
    @Override  1 usage   sachin_s *
    public ResponseDTO<LoginResponseDTO> login(UserLoginRequestDTO requestDTO) {
        try {
            Connection connection = DBConnection.getInstance().getConnection();

            String username = requestDTO.getUsername();
            String password = requestDTO.getPassword();

            User user = userRepository.findByUsername(connection, username)
                    .orElseThrow(() -> new AppException("User not found"));

            boolean checkPassword = checkPassword(password, user.getPassword());
            if (checkPassword) {
                String token = JwtUtil.generateToken(username, role: "USER");
                LoginResponseDTO loginResponseDTO = LoginResponseDTO.builder()
                        .accessToken(token)
                        .expTime(String.valueOf(JwtUtil.EXPIRATION_TIME))
                        .build();
                return ResponseDTO.success(loginResponseDTO);
            }
            throw new AppException("Incorrect password!!");

        } catch (Exception e) {
            throw new AppException("Something went wrong!!");
        }
    }
}
```

Driver Service

The service acts as a means of retrieving all drivers, creating a new driver, and updating a driver's existing state. Each method will respond with a ResponseDTO object, which will probably carry the response data along with other status-related information. This service is designed to manipulate information about drivers in a systematic and consistent way, efficiently managing the fetch, creation, and update of drivers.

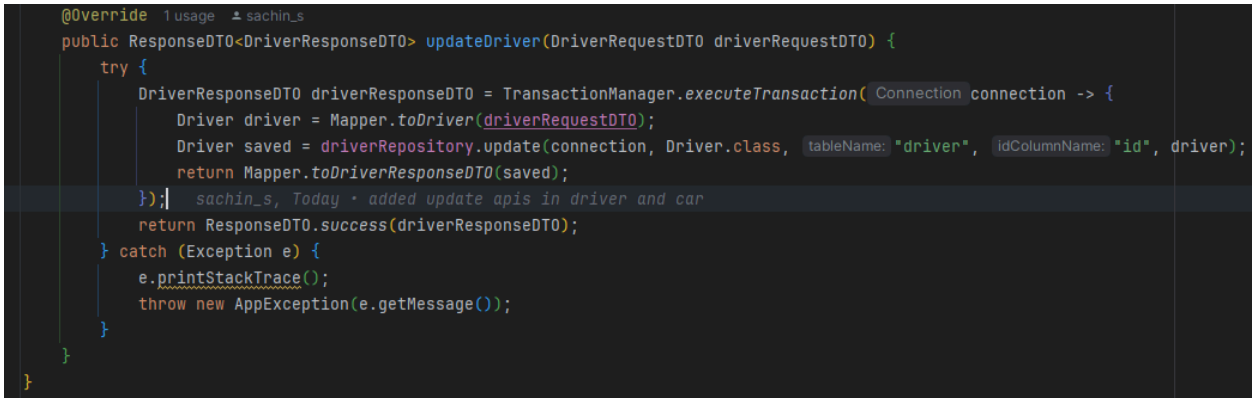
1. **getAllDrivers() -** Retrieves a list of all drivers from the database. It uses a DriverRepository to interact with the database, fetching the driver data and converting it into a list of DriverResponseDTO objects using a Mapper utility. The method handles database connections carefully, ensuring they are properly closed after use. If any SQL exceptions occur during the process, they are caught and rethrown as runtime exceptions. This implementation ensures that driver data is retrieved efficiently and consistently, providing a robust solution for managing driver information within the application.



2. createDriver(DriverRequestDTO driverRequestDTO) - handles the creation of new driver records. It converts a DriverRequestDTO to a Driver entity, saves it to the database using driverRepository, and returns the saved data as a DriverResponseDTO. The process is wrapped in a transaction for data integrity. Any exceptions are logged and rethrown as AppException, ensuring errors are managed effectively. This method provides a reliable way to add new drivers to the system.

```
@Override  1 usage  sachin_s
public ResponseDTO<DriverResponseDTO> createDriver(DriverRequestDTO driverRequestDTO) {
    try {
        DriverResponseDTO driverResponseDTO = TransactionManager.executeTransaction( Connection connection -> {
            Driver driver = Mapper.toDriver(driverRequestDTO);
            Driver saved = driverRepository.save(connection, Driver.class, tableName: "driver", idColumnName: "id", driver);
            return Mapper.toDriverResponseDTO(saved);
        });
        return ResponseDTO.success(driverResponseDTO);
    } catch (Exception e) {
        e.printStackTrace();
        throw new AppException(e.getMessage());
    }
}
```

3.updateDriver(DriverRequestDTO driverRequestDTO) It takes a DriverRequestDTO as input, converts it to a Driver entity, and updates the corresponding record in the database using driverRepository.update. The updated data is then returned as a DriverResponseDTO. The operation is performed within a transaction to ensure data consistency. If any exceptions occur, they are logged and rethrown as AppException, providing a reliable way to manage and update driver information in the system.



Distance Service

The DistanceService interface manages distance-related operations, including retrieving all distances and creating new distance records, using ResponseDTO for structured responses.

1. getAllDistances() - retrieves all distance records from the database. It uses DistanceRepository to fetch data, converts it to DistanceResponseDTO using a Mapper, and returns the list within a ResponseDTO. Database connections are managed carefully, and SQL exceptions are handled by throwing AppException. This method ensures efficient and reliable retrieval of distance data.

2. createDistance(DistanceRequestDTO distanceRequestDTO) - handles the creation of new distance records. It converts a DistanceRequestDTO to a Distance entity, saves it to the database using distanceRepository, and returns the saved data as a DistanceResponseDTO. The method manages database connections and handles exceptions by throwing AppException, ensuring reliable and consistent creation of distance records.

Car Service

Includes methods to retrieve available cars, save new car records, and update existing car information. Each method returns a ResponseDTO for structured responses, ensuring consistent handling of car data within the application.

```
/**
 * Author : SachinSilva
 */
public interface CarService extends SuperService{ 4 usages 1 impler
    sachin_s, 2/24/2025 2:40 PM • available cars completed
    ResponseDTO<List<CarResponseDTO>> getAvailableCars(); 1 usage

    ResponseDTO<CarResponseDTO> saveCar(CarRequestDTO car); 1 us

    ResponseDTO<CarResponseDTO> updateCar(CarRequestDTO car); 1
}
```

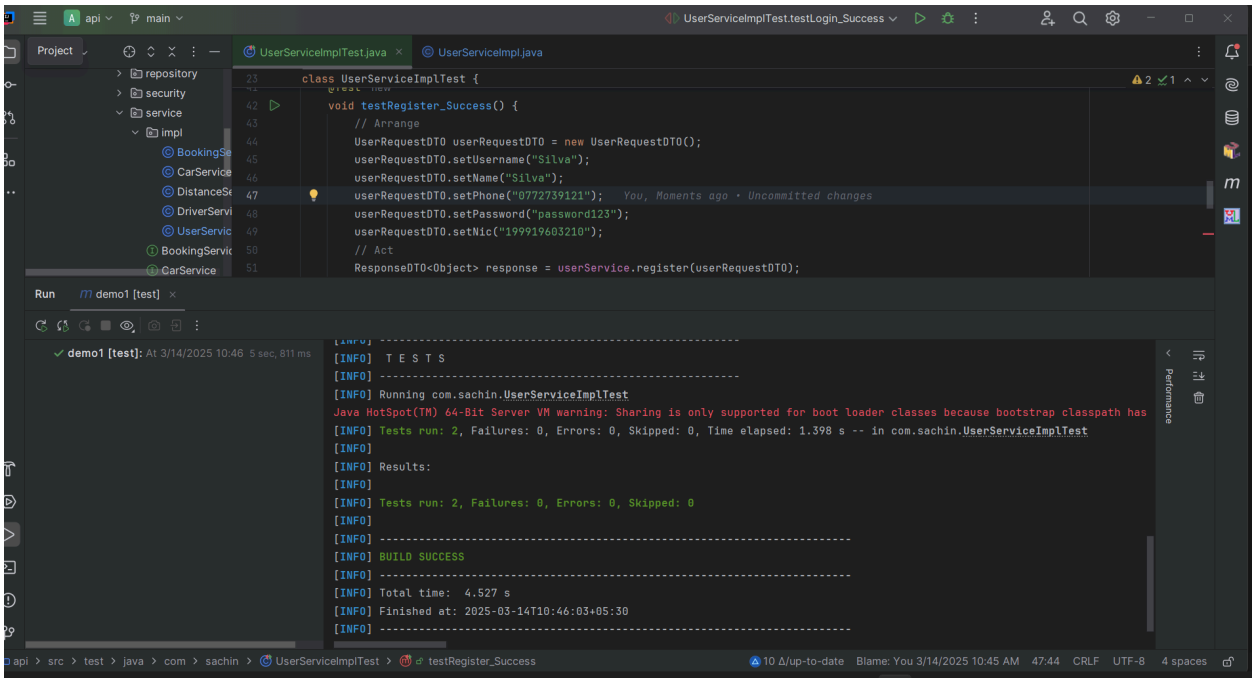
1. getAvailableCars() - retrieves a list of available cars from the database. It uses CarRepository to fetch the data and returns it as a list of CarResponseDTO objects within a ResponseDTO. The method ensures proper management of database connections and handles exceptions by throwing AppException, providing a reliable way to access available car information.

2.saveCar(CarRequestDTO carRequestDTO) - handles the creation of new car records. It converts a CarRequestDTO to a Car entity, saves it to the database using carRepository, and returns the saved data as a CarResponseDTO. The method ensures proper database connection management and handles exceptions by throwing AppException, providing a reliable way to add new cars to the system.

3. updateCar(CarRequestDTO carRequestDTO) - updates existing car records. It converts a CarRequestDTO to a Car entity, updates the corresponding record in the database using carRepository.update, and returns the updated data as a CarResponseDTO. The method ensures proper database connection management and handles exceptions by throwing AppException, providing a reliable way to modify car information in the system.

Testing

User Testing



Driver Testing