**Development plan**

**MoSCoW method used for the car rental systems.**

**Must have:**

1. Implementation of API endpoints for “CRUD” operations on cars, users and bookings:

• The API endpoints are crucial for the core functionality of the car rental system and allow users to interact with the system by adding, viewing, updating and deleting records for cars, users and bookings.

2. Implementation of authentication and authorization:

• Any application dealing with user data must be cared securely. Therefore, authentication ensures that only authorized users can access the system, while the authorization itself controls what actions each user can perform in the system.

3. Implementation of repository pattern for data access layer:

• The repository pattern promotes separation of concerns by abstracting the data access logic from the rest of the application which makes the codebase more maintainable and testable.

**Should have:**

1. Implementation of the documentation of Swagger for API endpoints:

• The documentation enhances the usability of the API and reduces the learning curve for developers integrating with the system, all of which provides a clear and interactive way for developers to understand and test API endpoints.

2. Foreign key relationships in the database schema:

• These relationships ensure data integrity by enforcing referential integrity between related tables which prevents orphaned records and maintains consistency in the database.

3. Angular SPA for Frontend CRUD operations:

• A single page application (SPA) provides a responsive UX by reducing page reloads which offers features like two-way data binding and dependency injection.

**Could have:**

1. Implementation of email notifications for booking confirmation:

• Email notifications improve the user experience by providing timely updates and reminders which can improve communication between the system and users, reducing the likelihood of missed bookings or misunderstandings.

2. Implementation of review and rating system for cars and users:

• A review and rating system enables customers to submit feedback on their rental experiences, allowing prospective users to make more informed judgments. It provides value to the platform by promoting trust and accountability.

**Won’t have:**

Integration with external services like GPS or social media:

• While connectivity with external services might improve the system's usefulness, it is outside the scope of a simple automobile rental system. These features can be considered for future upgrades if necessary.

**GANTT Chart**

|  |  |  |  |
| --- | --- | --- | --- |
| **Task** | **Start Date** | **End Date** | **Duration** |
| API endpoints for CRUD operations | 16.03.2024 | 20.03.2024 | 4 days |
| Repository Pattern for data access layer | 17.03.2024 | 20.03.2024 | 3 days |
| Swagger documentation for API endpoints | 18.03.2024 | 20.03.2024 | 2 days |
| Foreign key relationships in database | 19.03.2024 | 20.03.2024 | 1 day |
| Angular SPA for frontend CRUD operations | 20.03.2024 | 20.03.2024 | 1 day |

**Database Schema**

****

**Github**

**Link:** <https://github.com/wiut00013975/car-rental-system.git>

**System architecture**

My car rental system's architecture relies heavily on the Repository Pattern. It separates the data access logic from the rest of the application by adding a layer of abstraction to the database. In our approach, we set up repositories for each system entity, such as CarRepository and RentalRepository. These repositories encapsulate CRUD activities for their respective entities, offering a simple and uniform interface for dealing with the underlying data storage.  
  
  
Implementing the Repository Pattern improved the maintainability and flexibility of our data access layer. One of the primary benefits is the ability to easily switch between data sources. For example, if we decide to transition from a SQL database to a NoSQL database in the future, we won't have to change the higher-level application logic. This separation of data access issues from the rest of the program encourages code reuse and testing. Furthermore, by isolating data access functionality into repositories, we enhanced code structure and made it easier to understand and maintain.

Dependency Injection (DI): Application  
Dependency Injection is a basic design pattern that we employ extensively across our Angular-based frontend. It allows us to establish loose coupling between components and services by defining dependencies outside rather than within the component itself. Angular has DI, which is often used to inject services into components, modules, and other services. For example, in our CarService, we use HttpClient to send HTTP queries, and in our components, we use CarService to retrieve and handle automobile data.  
  
  
Using Dependency Injection has improved our codebase's modularity and testing efficiency. Dependencies are explicitly stated and made available externally, making it simple to alter or fake them during testing. This improves code maintainability and reuse by allowing components and services to be readily altered or extended without compromising their dependencies. Furthermore, Dependency Injection fosters concern separation by assigning presentation logic to components and business logic and data access to services respectively. Overall, Dependency Injection has helped make our frontend codebase more scalable, manageable, and tested.

**Documentation of API**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated