

Project part 1

Car Management- Web Application



**Aimen El-Sayed, Anna Erdi,
Rukayah Jabr, Aziz Ftaiti,
Georg Pappenheim**

Contents

Tools & technologies used.	3
Description of Web service APIs	2
Project Outline: Car Rental Application	4
Design of data model	7
Back-End	7
Front – End	12
Database	16
REST API	17
Database connection	17
Currency Converter	18
Cloud Deployment	19
Live DEMO	20
	20
GIT Repository	21
code Quality and Version	22
Lessons learned.	25
References	26

Description of Web service APIs

Project Summary

The Car Rental Management System is a web application designed to streamline the operations of a car rental service, catering to both customers and administrators. The system ensures a seamless experience for users to book vehicles and for administrators to manage the inventory and reservations efficiently.

Key Features

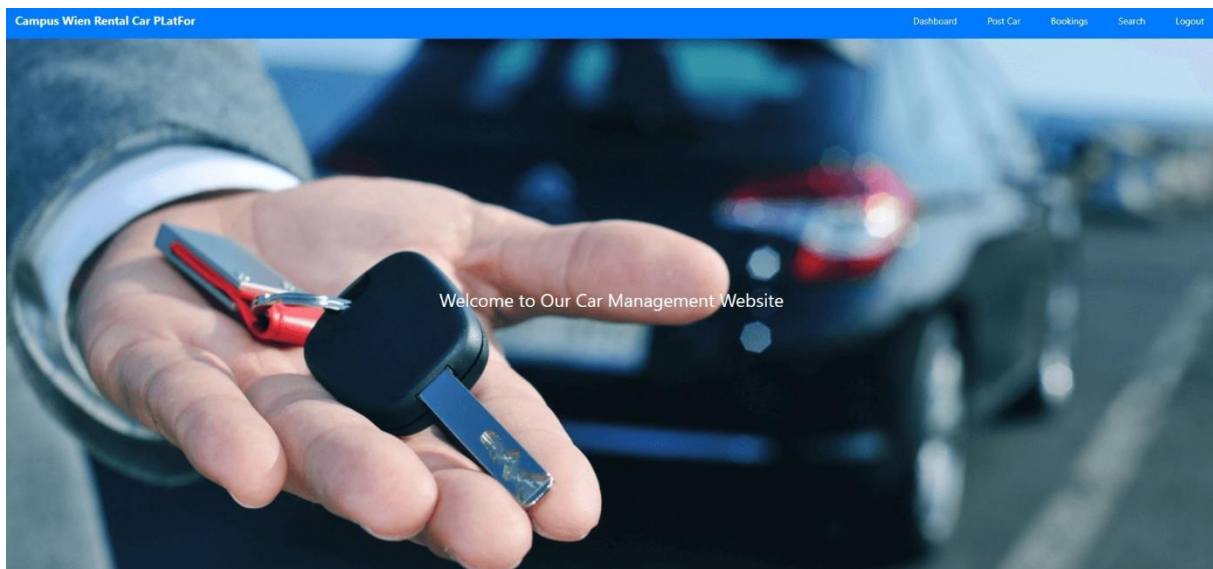
Customer Features:

User Registration and Login: Secure account creation and login with data validation.

Vehicle Search: Search for available vehicles based on date, type, and location.

Booking and Reservations: Book vehicles with real-time availability checks and manage reservations.

Admin accept **Reservations:** Admin can accept and refuse reservations.



Tools & technologies used.

Database – SQL

- MariaDB v_10.6
- phpMyAdmin
- Hostinger as host



Back-end

- Java v_17
- Spring boot v_3.1.
- Maven v_4
- Lombok, Rest



Front-end

- Angular v_16
- ng-zorro v_16
- bootstrap



TEST and ensure Code Quality

- karma-jasmine v_6
- Postman
- Mockito
- Chat GPT
- sonarlint



Deploy

- Git
- Github desktop
- Azure Cloud



Project Outline: Car Rental Application

1. Requirements Analysis

- **Customers:**
 - Register and manage their accounts.
 - Search for available vehicles based on criteria (e.g., date, type, location).
 - Book and cancel reservations.
 - View rental history.
- **Administrators:**
 - Manage vehicle inventory (add, update, delete vehicles).
 - View and manage customer reservations.

2. System Architecture

- **Backend:** Java (Spring Boot)
- **Frontend:** Angular for a web application
- **Database:** MySQL
- **Build Tool:** Maven
- **Version Control:** Git

3. Database Schema

- **Tables:**
 - Users (id, name, email, password, role)
 - Vehicles (id, make, model, year, price_per_day, status, etc....)
 - Reservations (id, user_id, vehicle_id, start_date, end_date, total_cost, status, etc...)

4. Backend Implementation

- **Entities and Repositories:**
 - User, Vehicle, Reservation, Payment
- **Services:**
 - UserService (register, login, booking car)
 - VehicleService (add, update, delete, search vehicles)
 - ReservationService (create, update, cancel reservations)
- **Controllers:**
 - AdminController
 - VehicleController
 - UserController

5. Frontend Implementation

- **User Interface:**
 - **Customer UI:**
 - Registration and Login forms
 - Vehicle search and booking forms
 - Reservation and payment history
 - **Admin UI:**
 - Vehicle management forms
 - Reservation management dashboard

6. Features and Functionalities

- **Customer Features:**
 - User registration and login with validation.
 - Search for vehicles by date, type, and location.
 - Book a vehicle with real-time availability checks.
 - View and manage reservations.
 - Secure payment processing.
- **Admin Features:**
 - CRUD operations for vehicle inventory.
 - View and manage all reservations.
 - Manage customer information and provide support.

7. Security and Authentication

- Implement authentication and authorization using Spring Security.
- Password encryption and secure storage.
- Role-based access control (customer vs. admin).

8. Testing

- **Unit Tests:** JUnit and Mockito for backend services.
- **Integration Tests:** Testing API endpoints.
- **User Acceptance Testing (UAT):** Ensuring the application meets business requirements.

9. Deployment

- **Local Deployment:** Using Docker for containerization.
- **Cloud Deployment:** AWS, Azure, or Heroku for cloud hosting.
-

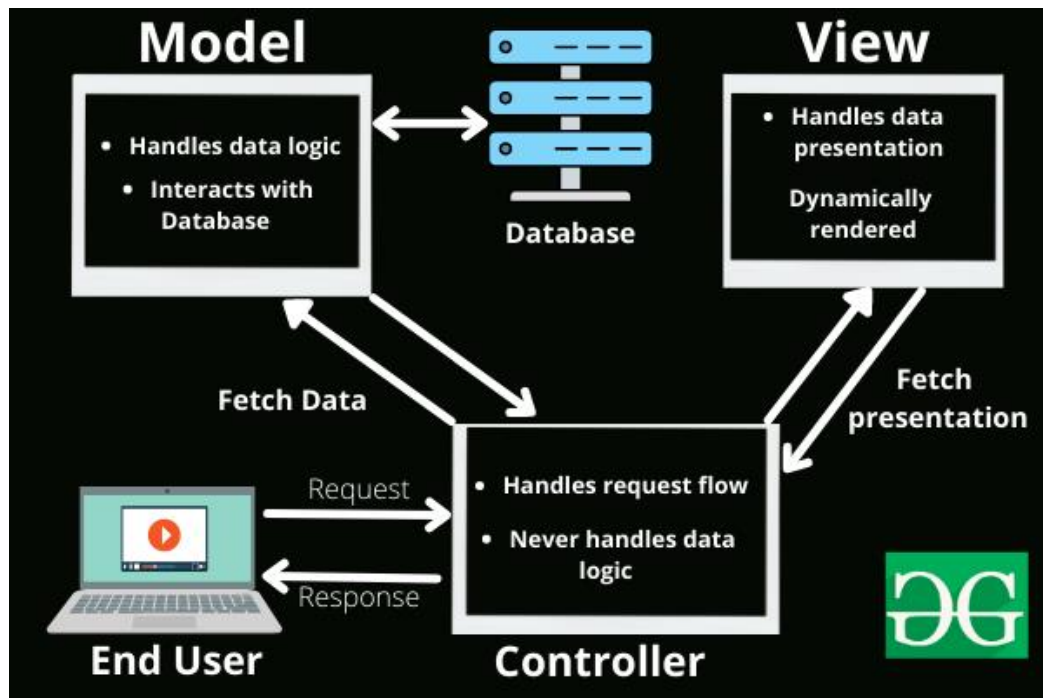
10. Documentation

- Detailed user manual for customers and administrators.
- API documentation using REST API.
- Technical documentation covering system architecture, database schema, and deployment instructions.

Design of data model

MVC

Model-View-Controller is a software architectural pattern that separates an application into three main logical components: the Model, responsible for managing data and business logic; the View, responsible for user interface and presentation logic; and the Controller, responsible for handling user input and updating the Model and View accordingly. This separation enhances code maintainability, scalability, and reusability.

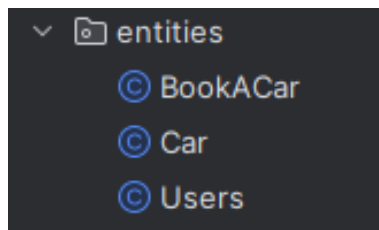


Back-End

1. Maven Dependency
2. Entities
3. Dtos
4. Repositories
5. Services
6. Controllers
7. Security

Entities

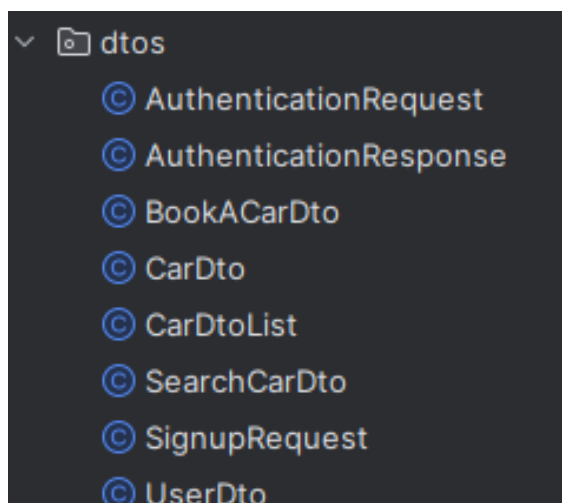
Entities in software development refer to objects or components that represent real-world concepts, such as a user, a product, or an order. They encapsulate both data and behavior related to a specific concept within the application's domain. In many cases, entities correspond directly to database tables in relational database systems.



```
1 package FHCampus.CarRental.entities;
2
3 > import ...
4
5 39 usages
6 @Entity
7 @Data
8 @Table(name = "cars")
9 public class Car {
10
11     @Id
12     @GeneratedValue(strategy = GenerationType.IDENTITY)
13     private Long id;
14
15     @NotBlank
16     @Size(max = 50)
17     private String name;
18
19     @NotBlank
20     @Size(max = 50)
21     private String color;
22
23     @NotBlank
24     @Size(max = 50)
25     private String transmission;
26 }
```

Dtos

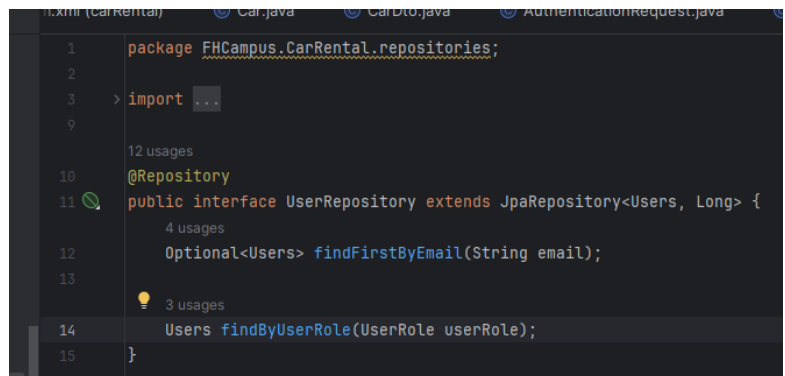
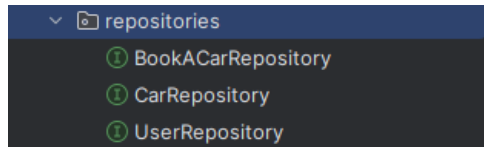
DTOs, or Data Transfer Objects, are simple objects used to transfer data between software application subsystems. They typically contain only data fields and no business logic, serving as a container for transferring data between different parts of an application, such as between the server and the client in a distributed system.



```
1 package FHCampus.CarRental.dtos;
2
3 import FHCampus.CarRental.enums.UserRole;
4 import lombok.Data;
5
6 8 usages
7 @Data
8 public class UserDto {
9
10     private Long id;
11
12     private String name;
13
14     private String email;
15
16     private String password;
17
18     private UserRole userRole;
19 }
```

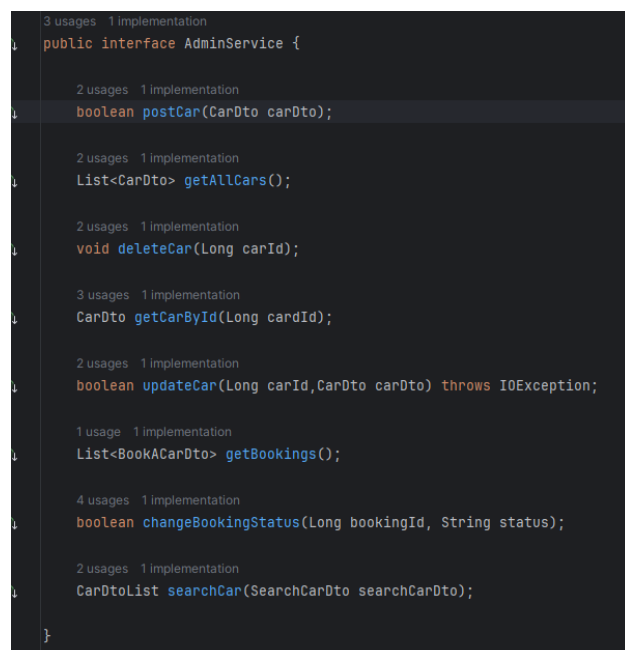
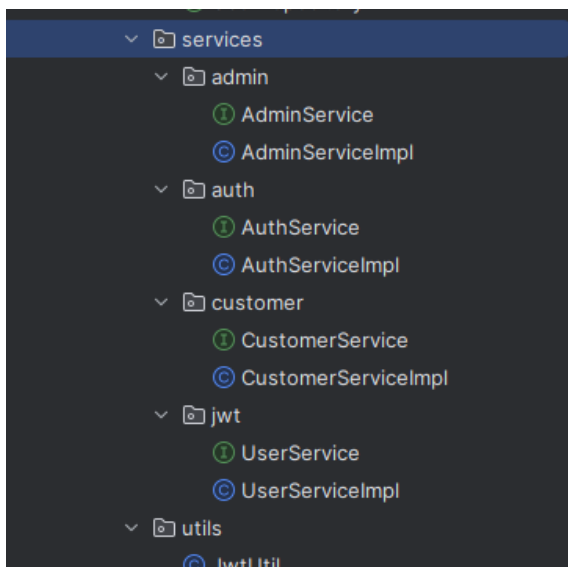
Repositories

Repositories in software development serve as an abstraction layer that encapsulates the logic required to access and manipulate data stored in a data source, such as a database. They provide a set of methods for querying, inserting, updating, and deleting data, thereby centralizing data access logic and promoting code reuse and maintainability.



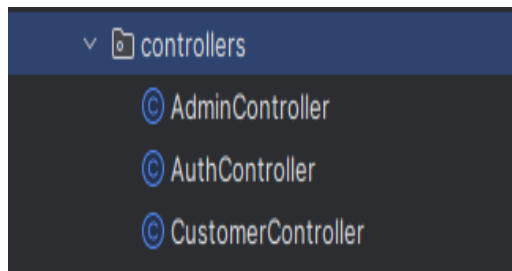
Services

Services in software development encapsulate business logic and application-specific functionality. They abstract complex operations into reusable components that can be called by different parts of the application. Services help maintain a separation of concerns by keeping business logic separate from presentation and data access layers, promoting modularity, scalability, and code maintainability.



Controllers

Controllers in software development are components responsible for handling incoming requests, interpreting user input, and coordinating the application's response. They typically contain methods or functions that map to specific endpoints or actions within the application. Controllers retrieve data from the model, process it as required, and pass it to the view for presentation to the user. They play a crucial role in implementing the logic that governs how the application responds to user interactions.



```
1 package FHCampus.CarRental.controllers;
2
3 > import ...
4
5 @RestController
6 @RequiredArgsConstructor
7 @RequestMapping("/api/admin")
8 public class AdminController {
9
10     private final AdminService adminService;
11
12     @PostMapping("/v1/car")
13     public ResponseEntity<?> postCar(@ModelAttribute CarDto carDto) {
14         boolean success = adminService.postCar(carDto);
15         if (success)
16             return ResponseEntity.status(HttpStatus.CREATED).build();
17         return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
18     }
19
20     @GetMapping("/v1/cars")
21     public ResponseEntity<List<CarDto>> getAllCars() {
22         List<CarDto> carDtoList = adminService.getAllCars();
23         return ResponseEntity.ok(carDtoList);
24     }
25
26     @DeleteMapping("/v1/car/{carId}")
27     public ResponseEntity<Void> deleteCar(@PathVariable Long carId) {
28         adminService.deleteCar(carId);
29         return ResponseEntity.noContent().build();
30     }
31
32     @GetMapping("/v1/car/{carId}")
33     public ResponseEntity<CarDto> getCarById(@PathVariable Long carId) {
34         CarDto carDto = adminService.getCarById(carId);
35     }
36 }
```

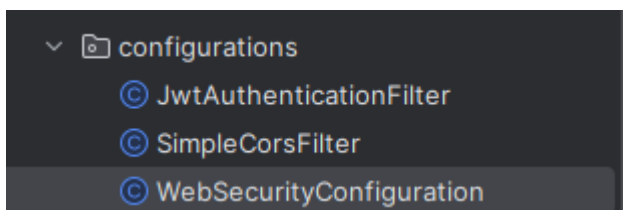
Security

JWT (JSON Web Token) is commonly used in security mechanisms to manage user authentication and authorization in web applications. When a user logs in, the server generates a JWT containing information about the user (claims) and signs it using a secret key. This token is then sent to the client, typically stored in local storage or cookies.

For subsequent requests, the client includes this JWT in the request headers. The server verifies the JWT's signature using the same secret key and extracts the user's information from the claims. This allows the server to authenticate the user and authorize access to protected resources based on the information contained in the JWT.

JWT provides several benefits for security:

1. Stateless: Since JWT contains all necessary information, there's no need to store session data on the server, making the system more scalable and easier to maintain.
2. Token-based: Tokens are self-contained and can be easily passed between client and server, making them suitable for use in distributed systems and APIs.
3. Claims-based: JWT allows for the inclusion of custom claims, providing flexibility in defining user roles, permissions, and other attributes.



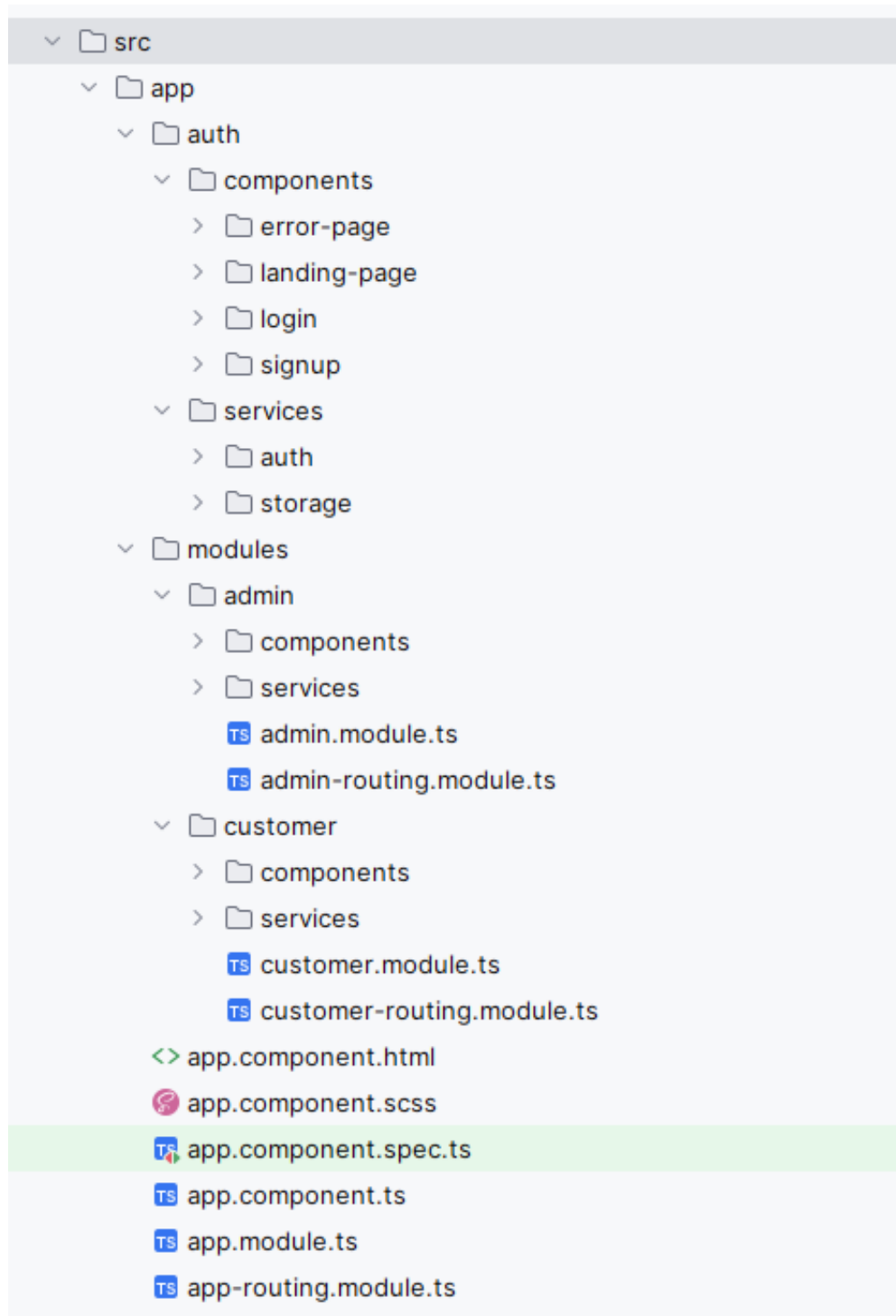
```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http.csrf(AbstractHttpConfigurer::disable).authorizeHttpRequests(request ->
        request.requestMatchers("/api/auth/**").permitAll()
            .requestMatchers("/api/admin/**").hasAnyAuthority(UserRole.ADMIN.name())
            .requestMatchers("/api/user/**").hasAnyAuthority(UserRole.CUSTOMER.name())
            .anyRequest().authenticated()).sessionManagement(manager ->
        manager.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authenticationProvider(authenticationProvider())
        .addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
```

```
1 usage
private String generateToken(Map<String, Object> extraClaims, UserDetails userDetails) {
    return Jwts.builder().setClaims(extraClaims).setSubject(userDetails.getUsername())
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 24))
        .signWith(getSigningKey(), SignatureAlgorithm.HS256).compact();
}

no usages
public String generateRefreshToken(Map<String, Object> extraClaims, UserDetails userDetails) {
    return Jwts.builder().setClaims(extraClaims).setSubject(userDetails.getUsername())
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + 604800000))
        .signWith(getSigningKey(), SignatureAlgorithm.HS256).compact();
}
```

Front – End

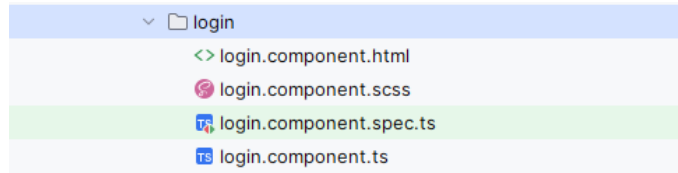
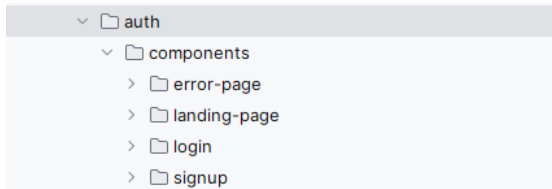
1. Components
2. Services
3. Admin module
4. Customer module
5. Routing module



Auth

Components

Login Example



```
<form nz-form [formGroup]="loginform">
  <nz-form-item>
    <nz-form-control nzErrorTip="Please submit your email!">
      <nz-input-group nzPrefixIcon="user">
        <input type="text" nz-input placeholder="Email" formControlName="email" />
      </nz-input-group>
    </nz-form-control>
  </nz-form-item>
  <nz-form-item>
    <nz-form-control nzErrorTip="Please input Password, more than 7 char!">
      <nz-input-group nzPrefixIcon="lock">
        <input type="password" nz-input placeholder="Password" formControlName="password" />
      </nz-input-group>
    </nz-form-control>
  </nz-form-item>

  <button nz-button class="login-form-button login-form-margin" [nzType]="primary" (click)="login()"
    [disabled]="loginform.invalid">Log in</button>
  Or <a routerLink="/register"> register now! </a>
</form>
```

```
9 @Component({
10   selector: 'app-login',
11   templateUrl: './login.component.html',
12   styleUrls: ['./login.component.scss']
13 })
14 export class LoginComponent {
15
16   isSpinning: boolean = false;
17   loginform: FormGroup;
18
19   no usages  ± Aimen
20   constructor(private fb: FormBuilder,
21     private authService: AuthService,
22     private message: NzMessageService,
23     private router: Router) { }
24
25   no usages  ± Aimen
26   ngOnInit(): void {
27     this.loginform = this.fb.group({ controls: {
28       email: [null, [Validators.required, Validators.email]],
29       password: [null, [Validators.required, Validators.minLength(7)]]
30     }});
31
32   2 usages  ± Aimen
33   login(): void {
34     console.log(this.loginform.value);
35     this.authService.login(this.loginform.value).subscribe( next: (res) : void => {
36       console.log(res);
37       if (res.userId != null) {
38         const user : (id: any, role: any) = {
39           id: res.userId,
40           role: res.userRole
41         };
42         StorageService.saveToken(res.jwt);
43         StorageService.saveUser(user);
44         if (StorageService.isAdminLoggedIn())
45           this.router.navigateByUrl( url: "/admin/dashboard");
46         else
```

Service -> Code snippet

```
const BASIC_URL : string[] = ["http://localhost:8080"];

5+ usages  ± Aimen *
@Injectable({
  providedIn: 'root'
})
export class AuthService {

  no usages  ± Aimen
  constructor(private http: HttpClient) { }

  2 usages  ± Aimen
  register(signupRequest: any): Observable<any> {
    return this.http.post( url: BASIC_URL + "/api/auth/v1/signup", signupRequest);
  }

  2 usages  ± Aimen
  login(loginRequest: any): Observable<any> {
    return this.http.post( url: BASIC_URL + "/api/auth/v1/login", loginRequest);
  }
}
```

Storage -> Code snippet

```
import { Injectable } from '@angular/core';

const USER : "user" = "user";
const TOKEN : "token" = "token";

5+ usages  ± Aimen
@Injectable({
  providedIn: 'root'
})
export class StorageService {

  no usages  ± Aimen
  constructor() { }

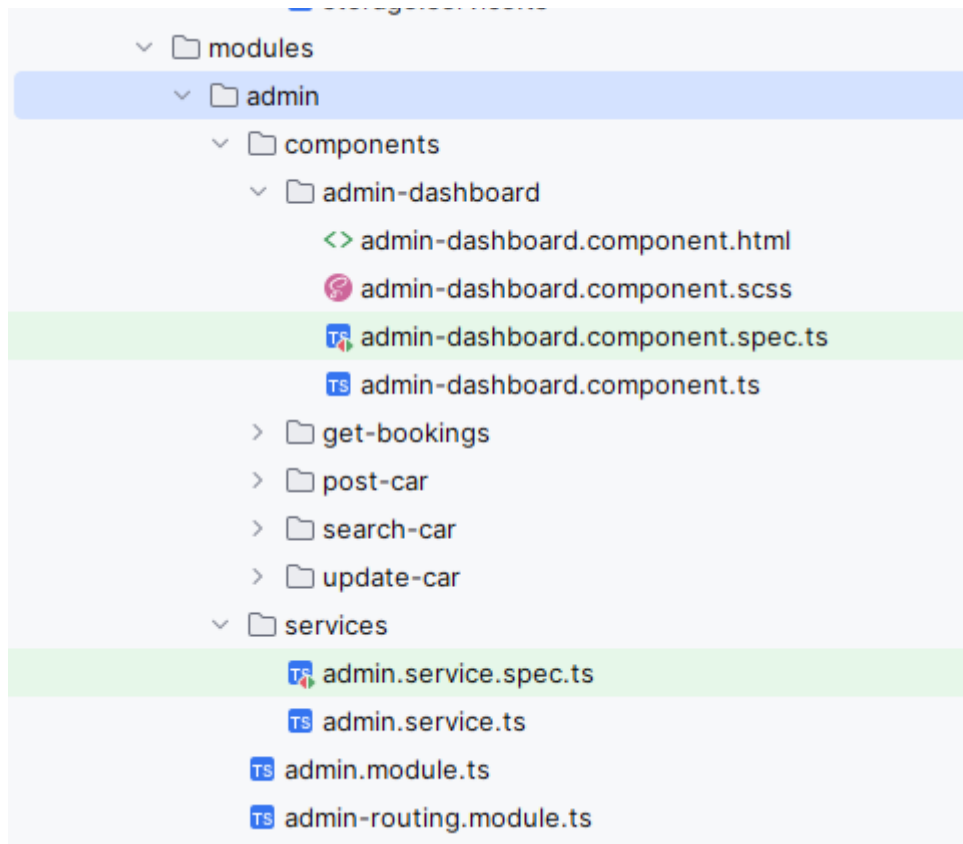
  1 usage  ± Aimen
  static saveToken(token: string): void {
    window.localStorage.removeItem(TOKEN);
    window.localStorage.setItem(TOKEN, token);
  }

  1 usage  ± Aimen
  static saveUser(user: any): void {
    window.localStorage.removeItem(USER);
    window.localStorage.setItem(USER, JSON.stringify(user));
  }

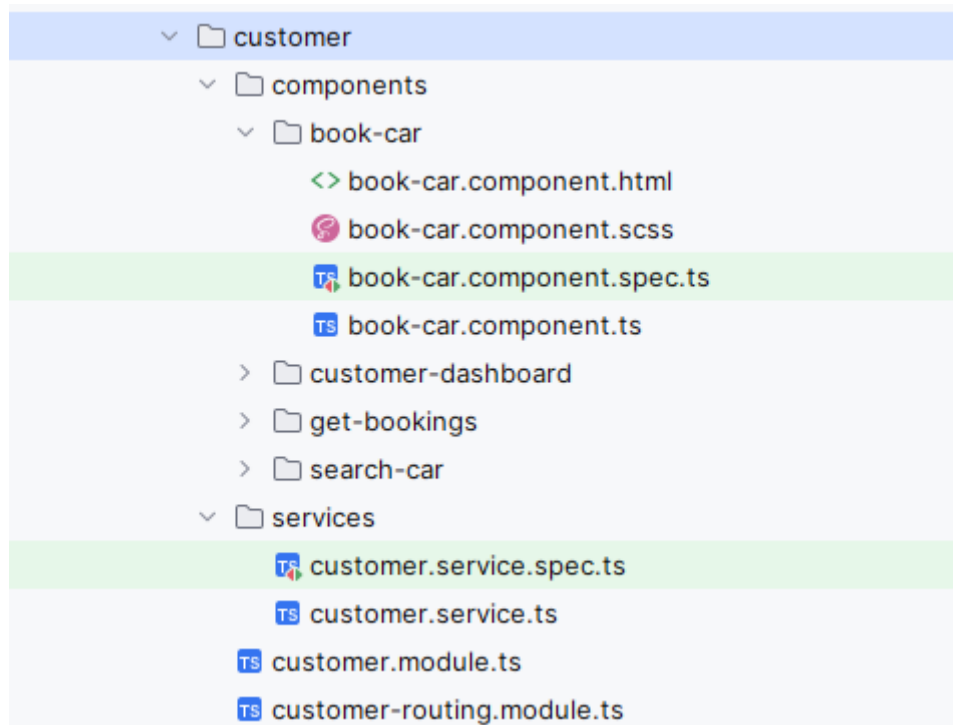
  2 usages  ± Aimen
  static getUser() {
    return JSON.parse(window.localStorage.getItem(USER));
  }

  4 usages  ± Aimen
  static getToken() : string {
    return window.localStorage.getItem(TOKEN);
  }
}
```

Admin Module -> Code snippet



Customer Module -> Code snippet



App - Routing

```
const routes: Routes = [
  { path: "register", component: SignupComponent },
  { path: "login", component: LoginComponent },
  { path: "admin", loadChildren: () => import("./modules/admin/admin.module").then(m => m.AdminModule) },
  { path: "customer", loadChildren: () => import("./modules/customer/customer.module").then(m => m.CustomerModule) },
  { path: '', component: LandingPageComponent },
  { path: '**', component: ErrorPageComponent }
];
```

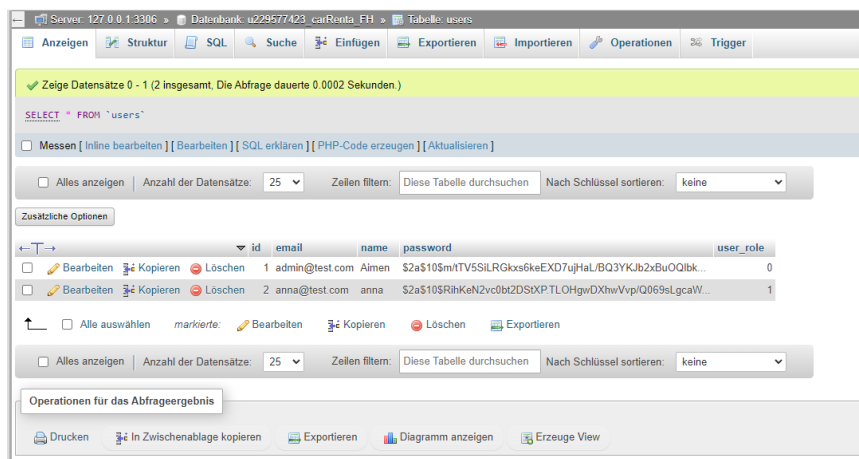
Admin – Routing

```
const routes: Routes = [
  { path: "dashboard", component: AdminDashboardComponent },
  { path: "car", component: PostCarComponent },
  { path: "car/:id/edit", component: UpdateCarComponent },
  { path: "bookings", component: GetBookingsComponent },
  { path: "car/search", component: SearchCarComponent },
];
```

Customer-Routing

```
const routes: Routes = [
  { path: "dashboard", component: CustomerDashboardComponent },
  { path: "book_car/:id", component: BookCarComponent },
  { path: "bookings", component: GetBookingsComponent },
  { path: "search", component: SearchCarComponent },
];
```

Database



The screenshot shows a database management interface with a query result for the 'users' table. The query is 'SELECT * FROM `users`'. The result shows two rows of user data.

	id	email	name	password	user_role
<input type="checkbox"/>	1	admin@test.com	Aimen	\$2a\$10\$m1TV5SILRGkxs6keEXD7ujHaL/BQ3YKJb2xBuOQlBk...	0
<input type="checkbox"/>	2	anna@test.com	anna	\$2a\$10\$RihKeN2vc0bt2DSIXP.TLOHgwDXhwVvp/Q069sLgcaW...	1

Operations for the query result: Drucken, In Zwischenablage kopieren, Exportieren, Diagramm anzeigen, Erzeuge View

REST API

-> Code snippet

```
1 usage  ± Aimen
postCar(carDto: any) : Observable<Object> {
    return this.http.post( url: BASIC_URL + "/api/admin/v1/car", carDto, options: {
        headers: this.createAuthorizationHeader()
    })
}

1 usage  ± Aimen
getAllCars(): Observable<any> {
    return this.http.get( url: BASIC_URL + "/api/admin/v1/cars", options: {
        headers: this.createAuthorizationHeader()
    })
}

1 usage  ± Aimen
deleteCar(carId: number): Observable<any> {
    return this.http.delete( url: BASIC_URL + "/api/admin/v1/car/" + carId, options: {
        headers: this.createAuthorizationHeader()
    })
}

1 usage  ± Aimen
getCarById(carId: number): Observable<any> {
    return this.http.get( url: BASIC_URL + "/api/admin/v1/car/" + carId, options: {
        headers: this.createAuthorizationHeader()
    })
}

1 usage  ± Aimen
updateCar(carId: number, carDto: any) : Observable<Object> {
    return this.http.put( url: BASIC_URL + "/api/admin/v1/car/" + carId, carDto, options: {
        headers: this.createAuthorizationHeader()
    })
}
```

Database connection

```
spring.datasource.url=jdbc:mysql://srv1373.hstgr.io/u229577423_carRenta_FH
spring.datasource.username=u229577423_admin
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect =org.hibernate.dialect.MySQL8Dialect
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.show-sql=true
```

Currency Converter

Description of the Web Service API & Application

The initial SOAP service is written in Python using the “spyne” framework to create a currency conversion API (“CurrencyConverterService”). This service provides methods for converting amounts between different currencies using SOAP Web Service Description Language (WSDL). The service setup includes the following:

- WSDL Document: The service exposes a WSDL file to describe the methods and expected data formats for API clients.
- Service Implementation: The Python application includes a SOAP server that hosts the currency conversion logic.

Cloud Deployment

We have chosen to deploy the application's components, including the Java Spring Boot monolithic backend, the application's Angular frontend, and the currency converter service, to Microsoft's Azure Cloud. Azure Web Apps serves as the hosting platform for these components.

An Azure Web App is a fully managed web hosting service provided by Microsoft Azure that allows developers to build, deploy, and scale web applications, APIs, and mobile backends quickly and securely. It supports various programming languages and frameworks, such as Python, .NET, Node.js, and Java, and offers features like automatic scaling, continuous integration and deployment (CI/CD), load balancing, and secure communication with built-in SSL certificates. As a Platform-as-a-Service (PaaS) offering, Azure Web App simplifies infrastructure management, allowing developers to focus on writing code and delivering business value.

Although we use a monorepository, to streamline and expedite the setup of GitHub Actions, the individual components were copied into separate directories and deployed to the pre-configured Azure Web Apps through a GitHub Actions pipeline. This automatic deployment build ensures that the latest version is consistently pushed to the cloud whenever a change is made to the main branch.

The individual repositories with the copied code and the streamlined CI/CD pipelines with GitHub Actions can be found on the following links:

- Currency converter: <https://github.com/annaerdi/soap-currency-converter>
- Backend: <https://github.com/annaerdi/car-rental-mono>
- Frontend: <https://car-rental-application-and-web-service-public.vercel.app/>

The default domains of the deployed components are the following:


- Currency converter: <https://soap-currency-converter.azurewebsites.net>
- Backend: <https://car-rental-mono.azurewebsites.net>
- Frontend: <https://car-rental-application-and-web-service-public.vercel.app/>



Live DEMO







GIT Repository



Link: <https://github.com/AimenFH/car-rental-application-and-web-service.git>



**car-rental-application-and-web-service** Public









 Pin  Unwatch 1



 main  3 Branches  0 Tags



 Add file  Code

**AimenFH** update database resources c7ee7c2 · 2 days ago  15 Commits

	application-car rental-services	update database resources	2 days ago
	application-cloud deployment	setup git folders	2 months ago
	application-currency converter-services	added currency converter soap service	last month
	application-documentation & presentation	setup git folders	2 months ago
	application-google maps -services	setup git folders	2 months ago
	application-ui	update error and landing page	2 weeks ago
	.gitignore	Create .gitignore	3 weeks ago
	README.md	Initial commit	2 months ago

 README 

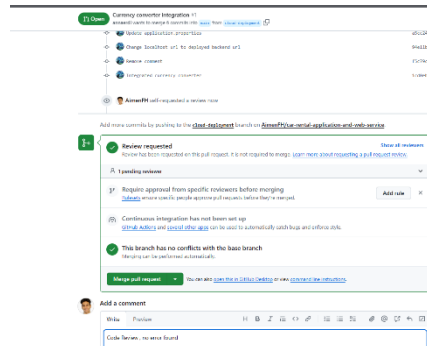
car-rental-application-and-web-service

Streamline your car rental experience with our intuitive application and web service, offering seamless reservations and reliable customer support.

code Quality and Version

1. Code Reviews:

- **Peer Reviews:** Regular code reviews by peers to identify potential issues and suggest improvements.



2. Coding Standards:

- **Style Guides:** Follow a consistent coding style guide ensure code readability and maintainability.
- **Linting Tools:** Use linting tools (e.g., Check style) to enforce coding standards and catch style violations early.

3. Unit Testing:

- **JUnit:** Write comprehensive unit tests for all critical components using JUnit.
- **Mockito:** Use Mockito to create mocks and stubs for testing interactions between components.
- **Spring Test:** Leverage Spring Boot's testing capabilities for integration tests.
- portion of the codebase is tested.

```
class AdminServiceImplTest {
    @Autowired
    @Mock
    private CarRepository carRepository;

    @Autowired
    @Mock
    private BookCarRepository bookCarRepository;

    @Autowired
    @Mock
    private AdminServiceImpl adminService;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.initMocks(this);
    }

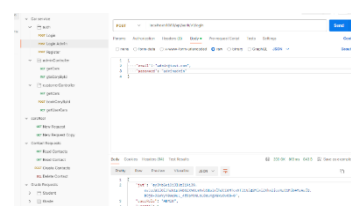
    @Test
    void testPostCar_exception() {
        Car car = new Car("Test car", "Ford", "Blue", 2020, "SUV", "Manual", "Ford", "2022", "Description", "Description");
        when(carRepository.save(any(Car.class))).thenReturn(car);
        boolean result = adminService.postCar(car);
        assertFalse(result);
    }

    @Test
    void testGetAllCars() {
        when(carRepository.findAll()).thenReturn(Arrays.asList(new Car(), new Car()));
        List<Car> result = adminService.getAllCars();
        assertEquals(2, result.size());
    }
}
```

4. Documentation:

- **Code Documentation:** Use Javadoc to document classes, methods, and other code elements.

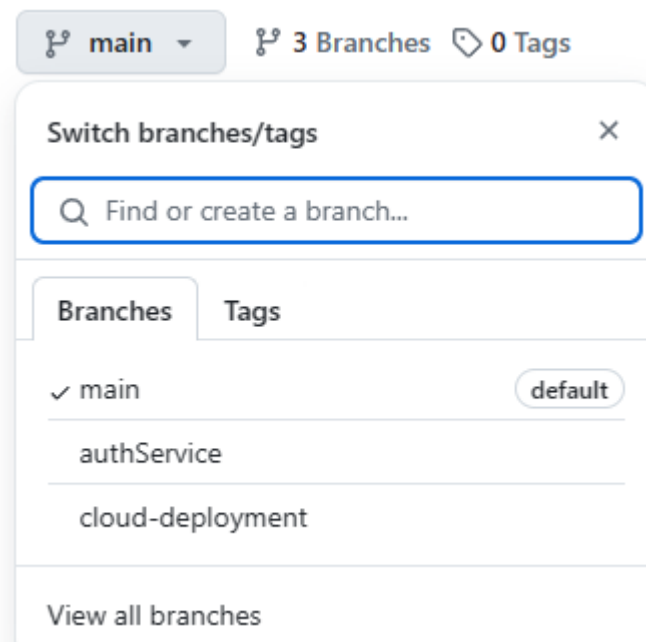
5. Postman Test during back end deploy.



Version Control

1. Git Workflow:

- **Branching Strategy:** Adopt a branching strategy like GitHub Flow to manage feature development, releases, and hotfixes.
- **Feature Branches:** Develop new features in separate branches to isolate changes until they are ready to be merged.

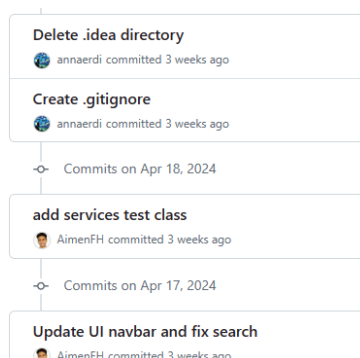


2. Repository Management:

- **Central Repository:** Use a central repository on platforms like GitHub, GitLab, or Bitbucket to collaborate with team members.
- **Access Control:** Manage access control to ensure that only authorized team members can push changes to the repository.

3. Commit Practices:

- **Commit Messages:** Write clear and meaningful commit messages that describe the changes made.



4. Versionen:

- **Semantic Versioning:** Follow semantic versioning label releases and communicate the impact of changes.

```
@RequestMapping("/api/admin")
public class AdminController {

    private final AdminService adminService;

    @PostMapping("/v1/car")
    public ResponseEntity<?> postCar(@ModelAttribute CarDto carDto) {
        boolean success = adminService.postCar(carDto);
        if (success)
            return ResponseEntity.status(HttpStatus.CREATED).build();
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
    }

    @GetMapping("/v1/cars")
    public ResponseEntity<List<CarDto>> getAllCars() {
        List<CarDto> carDtoList = adminService.getAllCars();
        return ResponseEntity.ok(carDtoList);
    }
}
```

```
2 usages - Aimen
register(signupRequest: any): Observable<any> {
    return this.http.post( url: BASIC_URL + "/api/auth/v1/signup", signupRequest);
}

2 usages - Aimen
login(loginRequest: any): Observable<any> {
    return this.http.post( url: BASIC_URL + "/api/auth/v1/login", loginRequest);
}
}
```

Lessons learned.

Backend

Proper GitHub OAuth credentials and secret management are important. Ensuring accurate authorization and permissions prevents authentication errors. Reissuing tokens and managing repository secrets directly can help maintain smooth OAuth functionality.

The standard local ports used during development (like port 8080 for Spring Boot) differ from the production ports (80/443) used by Azure. By utilizing the fully qualified domain name (FQDN) provided by Azure App Service, requests are correctly routed through Azure's reverse proxy, aligning the backend API endpoints with Azure's port configuration.

Frontend

For frontend applications, ensure API endpoints are accurately updated to reflect the backend's cloud URL. Modifying configuration variables (e.g., BASIC_URL in Angular) to point to the Azure App Service URL ensures that the frontend can communicate effectively with the cloud-hosted backend. This replaces hard-coded local references, enabling consistent access to the deployed backend.

Currency Converter

An initial application error on startup highlighted the importance of ensuring compatibility between dependencies and the environment. We learned that using specific versions of libraries like urllib3 and requests in requirements.txt is crucial, particularly when there are OpenSSL version constraints.

When the SOAP service failed to start due to hanging requests, we realized the significance of carefully managing dependencies and ensuring all required packages are available. Adjusting the requirements.txt file to include the correct dependencies helped with smooth service initialization.

Using consistent protocols in SOAP endpoints. The "Forcing soap: address location to HTTPS" message taught us that WSDL files must align with the actual deployment environment. Explicitly referencing the correct https:// address in the WSDL ensured consistent and secure communication between clients and the Azure-hosted SOAP service.

Version

in front end there is for example angular and node js work together and it so important to keep version of both suitable to each other, also with spring boot and spring security as it take a long of time.

Keep thing simple and spent more time in design.

We start with a very big backend project to find us at end that we have too many service that not needed which also affect our cloud deployment, this way we must restructure back end which cost use a lot of time.

debug

during deployment we face a lot of bugs and error which spent much time in stack overflow for example to solve it, which teach us a lot of debug skill and how to read error from console.

test

is better approach to test ever small part you implement, as to wait to write big block of code , which make it easier to detect error

References

Helpful tool

- <https://chatgpt.com/?oai-dm=1>
- <https://www.canva.com/>

YouTube explain videos.

- https://www.youtube.com/watch?v=sm-8qfMWEV8&list=PLqq-6Pq4ITTYTEooakHchTGglSvkZAJnE&ab_channel=JavaBrains
- https://www.youtube.com/watch?v=BOUMR85B-V0&list=PLhs1urmduZ2-yp3zID5rMEmXDETn8xvMo&ab_channel=GenuineCoder
- https://www.youtube.com/@code_with_projects
- https://www.youtube.com/watch?v=YofHJOWea-I&list=PLgYFT7gUQL8GNv-tqTsmwh6O_onYBFCiu&ab_channel=CodeWithProjects
- https://www.youtube.com/watch?v=bB6A490Uh5M&ab_channel=JavaGuides

code snippet.

<https://github.com/danielmiessler/SecLists>
<https://github.com/Tomasz3976/car-rental-project/blob/main/src/main/java/com/example/carrentalproject/security/LoggedInUser.java>
<https://github.com/boris-ns/spring-security-boilerplate/tree/master/src/main>

Documentation

- <https://support.hostinger.com/en/articles/1864324-how-to-upload-and-set-up-your-database-at-hostinger>
- <https://www.w3schools.com/java/>
- <https://stackoverflow.com/>
- <https://www.baeldung.com/mockito-series>