30/06/2024

My Flat-Platform

Back-End

El-Sayed Aimen, Markus Mayer, Aziz Ftaiti
Matthias Werkl, Eler Wohlmuth

Table of Contents

Description

Project Summary

The My-Flat application is a comprehensive platform designed to streamline the management and communication processes for rental properties. It aims to enhance the efficiency and effectiveness of property management by providing a robust, user-friendly interface for both tenants and property managers. The platform facilitates various essential functions, including tenant registration, property and apartment management, booking management, defect reporting, and secure communication between tenants and property managers. By leveraging modern technologies and best practices, My-Flat ensures secure, scalable, and maintainable operations, ultimately improving the overall rental experience for all stakeholders.

Tools & Technologies Used

- **Frontend**: React.js, JavaScript
- **Backend**: Spring Boot, Java, Maven, Lombok
- **Database**: MySQL, phpMyAdmin, Hostinger as Host
- **Security**: JWT for authentication, HTTPS for secure communication
- **Document:** Postman
- **Test:** Mockito, Postman
- **Clean code:** Snorlint
- **IDE:** IntelliJ, GitHub Desktop, Docker
- **Others**: CORS Handling, REST APIs, Git pilot, ChatGPT

Project Summary and Key Features

Requirements Analysis

- Customers (Tenants):
    - login and manage their accounts.
    - Communicate with property management.
    - Access documents and notifications.
    - Report defects and issues.
- Administrators (Property Managers):
    - Manage tenant accounts and property information.
    - Distribute notifications and documents.
    - Track and manage defect reports.
    - Manage key issuance and redemption.

System Architecture

- Backend: Java (Spring Boot)
- Frontend: React.js, JavaScript
- Database: MySQL, phpMyAdmin
- Build Tool: Maven
- Version Control: Git

Database Schema

- Tables:
    - Users (id, name, email, password, role, phoneNumber)
    - Apartments (id, area, floor, number, price)
    - Properties (id, numberOfApartments, numberOfFloors, propertyAddress, propertyName)
    - Bookings (id, userId, apartmentId, fromDate, toDate, amount, status)
    - Defects (id, category, description, image, location, status, timestamp)
    - Messages (id, document, message)
    - Documents (id, content, isArchived, title)

- KeyManagement (id, apartmentId, issuanceDate, keyNumber, propertyId, redemptionDate, replacementRequested, userId)
- Notifications (id, buildingId, document, message, title, topId)
- Feedback (id, message, tenantId, timestamp)
- Appointments (id, date, description, title)

Backend Implementation

- Entities and Repositories:
  - Entities: User, Apartment, Property, Booking, Defect, Message, Document, KeyManagement, Notification, Feedback, Appointment
  - Repositories: UserRepository, ApartmentRepository, PropertyRepository, BookingRepository, DefectRepository, MessageRepository, DocumentRepository, KeyManagementRepository, NotificationRepository, FeedbackRepository, AppointmentRepository
- Services:
  - UserService: Handle user registration, login, and profile management.
  - PropertyService: Manage property and apartment data.
  - BookingService: Handle booking creation, updates, and cancellations.
  - DefectService: Manage defect reporting and tracking.
  - KeyManagementService: Handle key issuance and redemption.
- Controllers:
  - AuthController: Handle user authentication.
  - PropertyManagementController: Manage properties, apartments, bookings, and defects.
  - TenantController: Manage tenant-related actions such as viewing documents and reporting defects.
  - PropertyManagementCommunicationController: Handle communication between property managers and tenants.

Testing

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

Deployment

- Local Deployment: Using Docker for containerization.
- Cloud Deployment: Hosting on platforms like AWS, Azure, or Heroku.

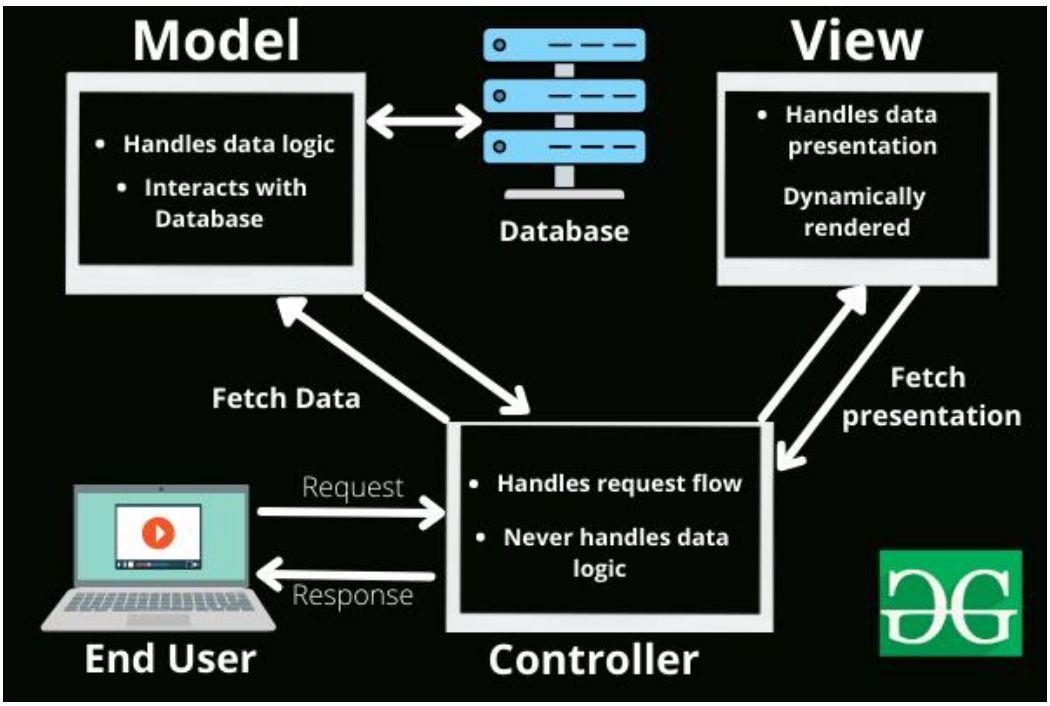
Documentation

- Detailed user manual for tenants and administrators.
- API Dokumentation using Postman.
- 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
8. API
9. Database

Maven Dependency

The pom.xml file is a Project Object Model (POM) file used by Maven to manage a project's build, reporting, and documentation. Here's a breakdown of the main elements in your pom.xml:

<modelVersion>: This element indicates the version of the object model this POM is using. The version here is 4.0.0.

<parent>: This element indicates that this POM inherits from a parent POM. The parent here is spring-boot-starter-parent with version 3.1.5.

<groupId>, <artifactId>, and <version>: These elements identify the project uniquely across all projects.

<properties>: This element is used to define properties, which are environment-specific parameters. Here, it defines the Java version and the JaCoCo Maven plugin version.

<dependencies>: This element is used to list all dependencies needed to build and run the project. Each <dependency> element specifies a dependency.

<build>: This element is used to provide build-specific information. The <plugins> element within the build element is used to configure the plugins.

The spring-boot-maven-plugin is used for building Spring Boot applications. The **<excludes>** element is used to exclude certain dependencies from the build.

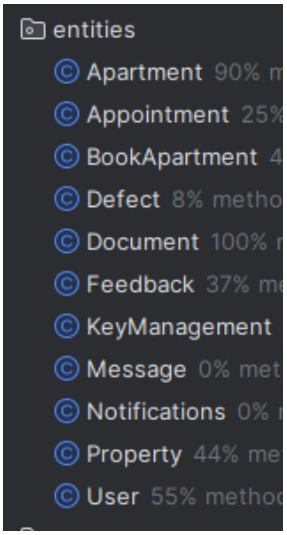
The jacoco-maven-plugin is used for generating code coverage reports. The <excludes> element is used to exclude certain files from the coverage report.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.1.5</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
```

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.

Example User Entity



The User class is an entity class in the Spring Boot application that represents the users table in the database. Here's a breakdown of its components:

Annotations: The class is annotated with @Entity, @Table, @Data, @AllArgsConstructor, and @NoArgsConstructor. @Entity indicates that the class is a JPA entity, @Table specifies the name of the database table to be used, and @Data, @AllArgsConstructor, and @NoArgsConstructor are Lombok annotations that automatically generate getters, setters, constructors, equals, hash Code, and to String methods.

Fields: The class has several fields, each representing a column in the user's table. Each field is annotated with JPA and validation annotations to enforce constraints:

id: The primary key of the user's table.

name, email, password, user Role, and phone Number: These fields represent the user's name, email, password, role, and phone number, respectively.

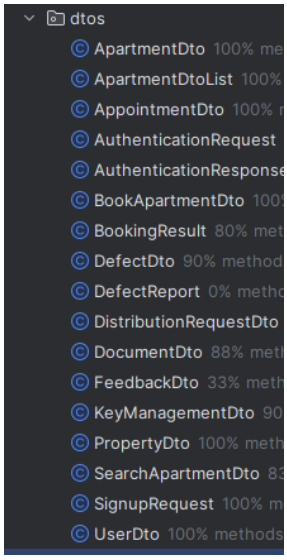
book Apartments and documents: These fields represent the one-to-many relationships between the User and Book Apartment and Document entities, respectively.

```java
@Entity
@AllArgsConstructor
@NoArgsConstructor
@Data
@Table(name = "users")
public class User implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Size(max = 20)
    private String name;

    @Size(max = 50)
    @Email
    private String email;

    @Size(max = 120)
    private String password;

    private UserRole userRole;
```

Methods: The class implements the User Details interface, which is used by Spring Security to handle user information. The overridden methods provide the details required by Spring Security for authentication and authorization. The getUserDto method is used to convert the User entity to a UserDto object.

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.



Example UserDTO

The **UserDto** class is a Data Transfer Object (DTO) in the application. DTOs are often used in Spring Boot applications to transfer data between processes or across network connections. Here's a breakdown of its components:

Annotations: The class is annotated with @Data and @NoArgsConstructor. @Data is a Lombok annotation that automatically generates getters, setters, equals, hashCode, and toString methods. @NoArgsConstructor is another Lombok annotation that generates a no-argument constructor.

Fields: The class has several fields, each representing a piece of user data that can be transferred:

id: The unique identifier of the user.

name: The name of the user.

email: The email of the user.

password: The password of the user.

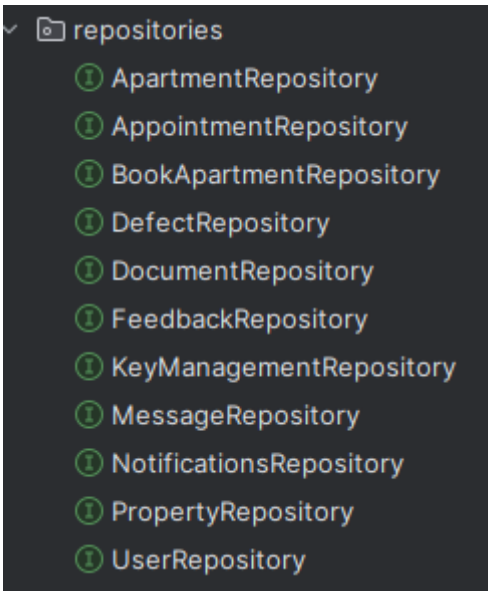
userRole: The role of the user, represented by an enum UserRole.

phoneNumber: The phone number of the user

```java
package fhcampus.myflat.dtos;

import ...

@NoArgsConstructor
@Data
public class UserDto {

    private Long id;
    private String name;
    private String email;
    private String password;
    private UserRole userRole;
    private  String phoneNumber;
    public UserDto(User user) {
        this.id = user.getId();
        this.name = user.getName();
        this.email = user.getEmail();
        this.password = user.getPassword();
        this.userRole = user.getUserRole();
        this.phoneNumber = user.getPhoneNumber();
    }
}
```

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.



Example User Repository

The UserRepository interface is a Spring Data JPA repository for the User entity. It extends JpaRepository, which provides methods for CRUD operations (Create, Read, Update, Delete) on the User entity. Here's a breakdown of its components:

Annotations: The interface is annotated with @Repository, which indicates that it's a Spring Data Repository.

Methods: The interface declares a method findFirstByEmail(String email). This method is used to find the first User entity that matches the provided email. It returns an Optional<User>, which can be empty if no User entity with the provided email is found.

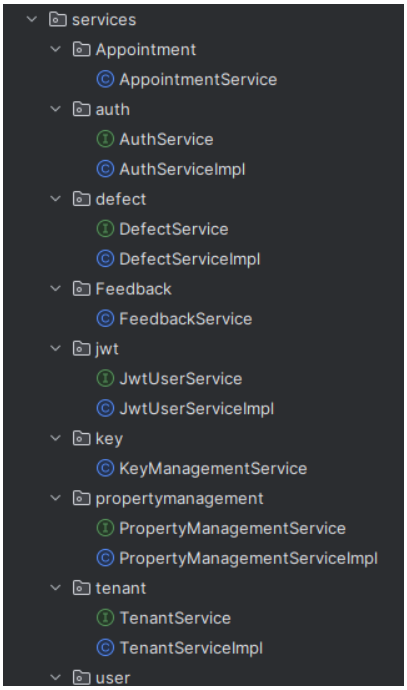
```java
package fhcampus.myflat.repositories;

import ...

14 usages
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    11 usages
    Optional<User> findFirstByEmail(String email);
}
```

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.

Example User Service

The UserServiceImpl class is a service class in the application that implements the UserService interface. Here's a breakdown of its components:



Annotations: The class is annotated with @Service and @RequiredArgsConstructor. @Service is a Spring annotation that indicates that the class is a service component. @RequiredArgsConstructor is a Lombok annotation that generates a constructor for all final fields, or fields marked with @NonNull.

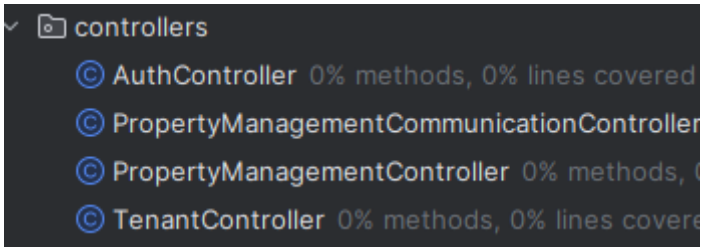
Methods: The class has a method getCurrentUser(). This method is used to get the currently authenticated user. It does this by calling SecurityContextHolder.getContext(). getAuthentication().getPrincipal(), which returns the principal (in this case, a User object) associated with the current authenticated user

```
@Service
@RequiredArgsConstructor
public class UserServiceImpl implements UserService {

    4 usages
    public User getCurrentUser() {
        return (User) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
    }
}
```

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.

Example Auth Controller



The Auth Controller class is a REST controller in the application that handles authentication-related requests. Here's a breakdown of its components:

Annotations: The class is annotated with @RestController, @RequiredArgsConstructor, and @RequestMapping. @RestController indicates that the class is a controller where every method returns a domain object instead of a view. @RequiredArgsConstructor is a Lombok annotation that generates a constructor for all final fields, or fields marked with @NonNull. @RequestMapping maps HTTP requests to handler methods of this controller.

Fields: The class has a field authService which is an instance of AuthService. This service is used to handle the business logic related to authentication.

Methods: The class has two methods, registerPropertyManagement and login.

```java
@RestController
@RequiredArgsConstructor
@RequestMapping("/api/auth")
public class AuthController {

    private final AuthService authService;

    @PostMapping("/v1/register/property-management")
    public ResponseEntity<?> registerPropertyManagement(@RequestBody SignupRequest signupRequest) {
        UserDto createdUserDto;
        try {
            createdUserDto = authService.createPropertyManagement(signupRequest);
        } catch (EmailAlreadyExistsException e) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(e.getMessage());
        }
        return ResponseEntity.status(HttpStatus.CREATED).body(createdUserDto);
    }

    @PostMapping("/v1/login")
    public ResponseEntity<?> login(@RequestBody AuthenticationRequest authenticationRequest) {
        AuthenticationResponse authenticationResponse;
        try {
            authenticationResponse = authService.login(authenticationRequest);
```

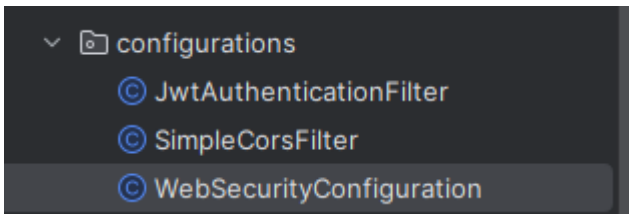
Security

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 uses 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.



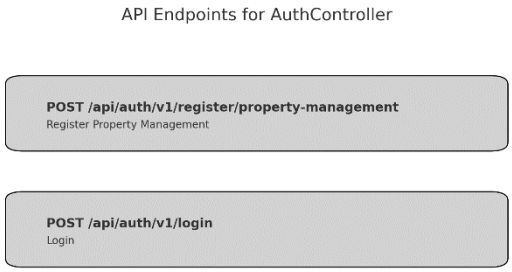

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http.csrf(AbstractHttpConfigurer::disable).authorizeHttpRequests(request ->
                    request.requestMatchers("/api/auth/**").permitAll()
                            .requestMatchers("/api/property-management/**").hasAnyAuthority(UserRole.PROPERTY_MANAGEMENT
                            requestMatchers("/api/tenant/**").hasAnyAuthority(UserRole.TENANT.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();
}
```

API

The MyFlat platform's backend features a comprehensive set of RESTful APIs designed to manage authentication, user interactions, and communication between tenants and property managers. Key APIs include:

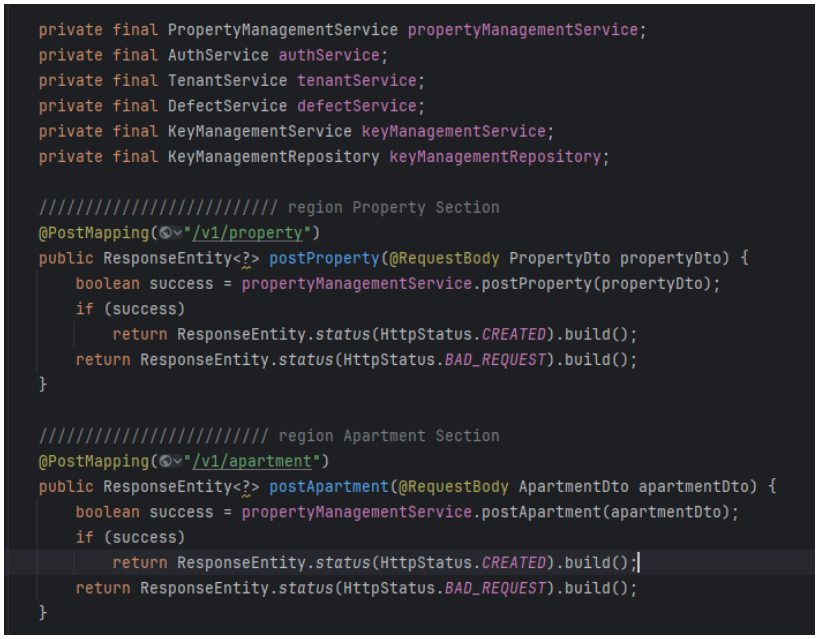
- **AuthController:** Handles user authentication processes such as login and registration.
- **PropertyManagementController:** Manages operations related to property management, including tenant account management and maintenance request handling.
- **TenantController:** Facilitates tenant interactions, providing access to documents and enabling defect reporting.
- **PropertyManagementCommunicationController:** Supports communication between property managers and tenants, allowing message exchange and document sharing.




```java
@RestController
@RequiredArgsConstructor
@RequestMapping("/api/auth")
public class AuthController {

    private final AuthService authService;

    @PostMapping("/v1/register/property-management")
    public ResponseEntity<?> registerPropertyManagement(@RequestBody SignupRequest signupRequest) {
        UserDto createdUserDto;
        try {
            createdUserDto = authService.createPropertyManagement(signupRequest);
        } catch (EmailAlreadyExistsException e) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(e.getMessage());
        }
        return ResponseEntity.status(HttpStatus.CREATED).body(createdUserDto);
    }

    @PostMapping("/v1/login")
    public ResponseEntity<?> login(@RequestBody AuthenticationRequest authenticationRequest) {
        AuthenticationResponse authenticationResponse;
        try {
            authenticationResponse = authService.login(authenticationRequest);
```

PropertyManagementController:

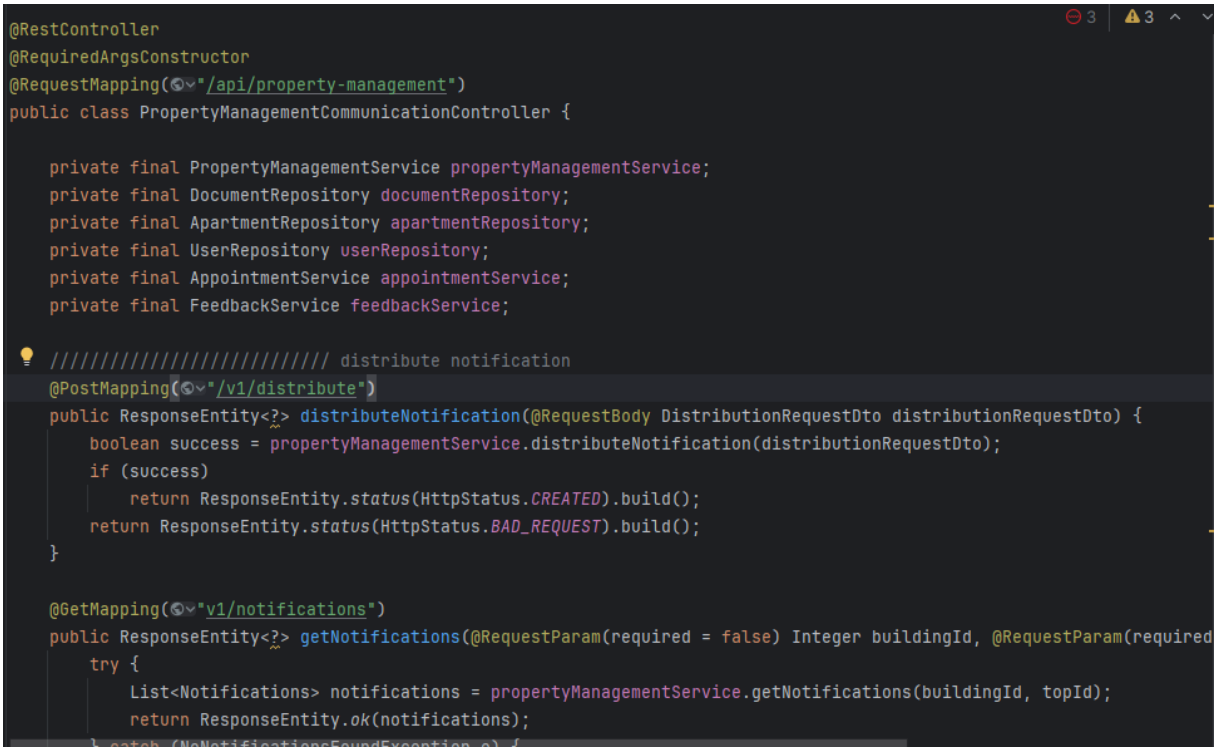
- POST /api/property-management/v1/property
- POST /api/property-management/v1/apartment
- GET /api/property-management/v1/apartments
- DELETE /api/property-management/v1/apartment/{apartmentId}
- GET /api/property-management/v1/apartment/{apartmentId}
- PUT /api/property-management/v1/apartment/{apartmentId}
- PUT /api/property-management/v1/apartment/booking/{bookingId}/{status}
- GET /api/property-management/v1/apartment/bookings
- POST /api/property-management/v1/apartment/search
- GET /api/property-management/v1/apartment/bookings/{userId}
- GET /api/property-management/v1/{userId}
- POST /api/property-management/v1/tenant/create
- POST /api/property-management/v1/apartment/book/{apartmentId}
- POST /api/property-management/v1/defect
- GET /api/property-management/v1/defects
- POST /api/property-management/v1/key-management
- GET /api/property-management/v1/key-management
- GET /api/property-management/v1/key-management/user/{userId}
- PUT /api/property-management/v1/key-management
- DELETE /api/property-management/v1/key-management



```
private final PropertyManagementService propertyManagementService;
private final AuthService authService;
private final TenantService tenantService;
private final DefectService defectService;
private final KeyManagementService keyManagementService;
private final KeyManagementRepository keyManagementRepository;

////////////////////////// region Property Section
@PostMapping("/v1/property")
public ResponseEntity<?> postProperty(@RequestBody PropertyDto propertyDto) {
    boolean success = propertyManagementService.postProperty(propertyDto);
    if (success)
        return ResponseEntity.status(HttpStatus.CREATED).build();
    return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
}

////////////////////////// region Apartment Section
@PostMapping("/v1/apartment")
public ResponseEntity<?> postApartment(@RequestBody ApartmentDto apartmentDto) {
    boolean success = propertyManagementService.postApartment(apartmentDto);
    if (success)
        return ResponseEntity.status(HttpStatus.CREATED).build();
    return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
}
```

PropertyManagementCommunicationController:

- POST /api/property-management/v1/distribute
- GET /api/property-management/v1/notifications
- POST /api/property-management/v1/document
- PUT /api/property-management/v1/document/{apartmentId}/{documentId}
- GET /api/property-management/v1/document
- DELETE /api/property-management/v1/document
- DELETE /api/property-management/v1/document/{documentId}
- POST /api/property-management/appointment/create
- DELETE /api/property-management/appointment/delete/{id}
- GET /api/property-management/appointment/all
- GET /api/property-management/feedback



```
@RestController
@RequiredArgsConstructor
@RequestMapping("/api/property-management")
public class PropertyManagementCommunicationController {

    private final PropertyManagementService propertyManagementService;
    private final DocumentRepository documentRepository;
    private final ApartmentRepository apartmentRepository;
    private final UserRepository userRepository;
    private final AppointmentService appointmentService;
    private final FeedbackService feedbackService;

    /////////////////////////// distribute notification
    @PostMapping("/v1/distribute")
    public ResponseEntity<?> distributeNotification(@RequestBody DistributionRequestDto distributionRequestDto) {
        boolean success = propertyManagementService.distributeNotification(distributionRequestDto);
        if (success)
            return ResponseEntity.status(HttpStatus.CREATED).build();
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
    }

    @GetMapping("v1/notifications")
    public ResponseEntity<?> getNotifications(@RequestParam(required = false) Integer buildingId, @RequestParam(required
        try {
            List<Notifications> notifications = propertyManagementService.getNotifications(buildingId, topId);
            return ResponseEntity.ok(notifications);
```

TenantController:

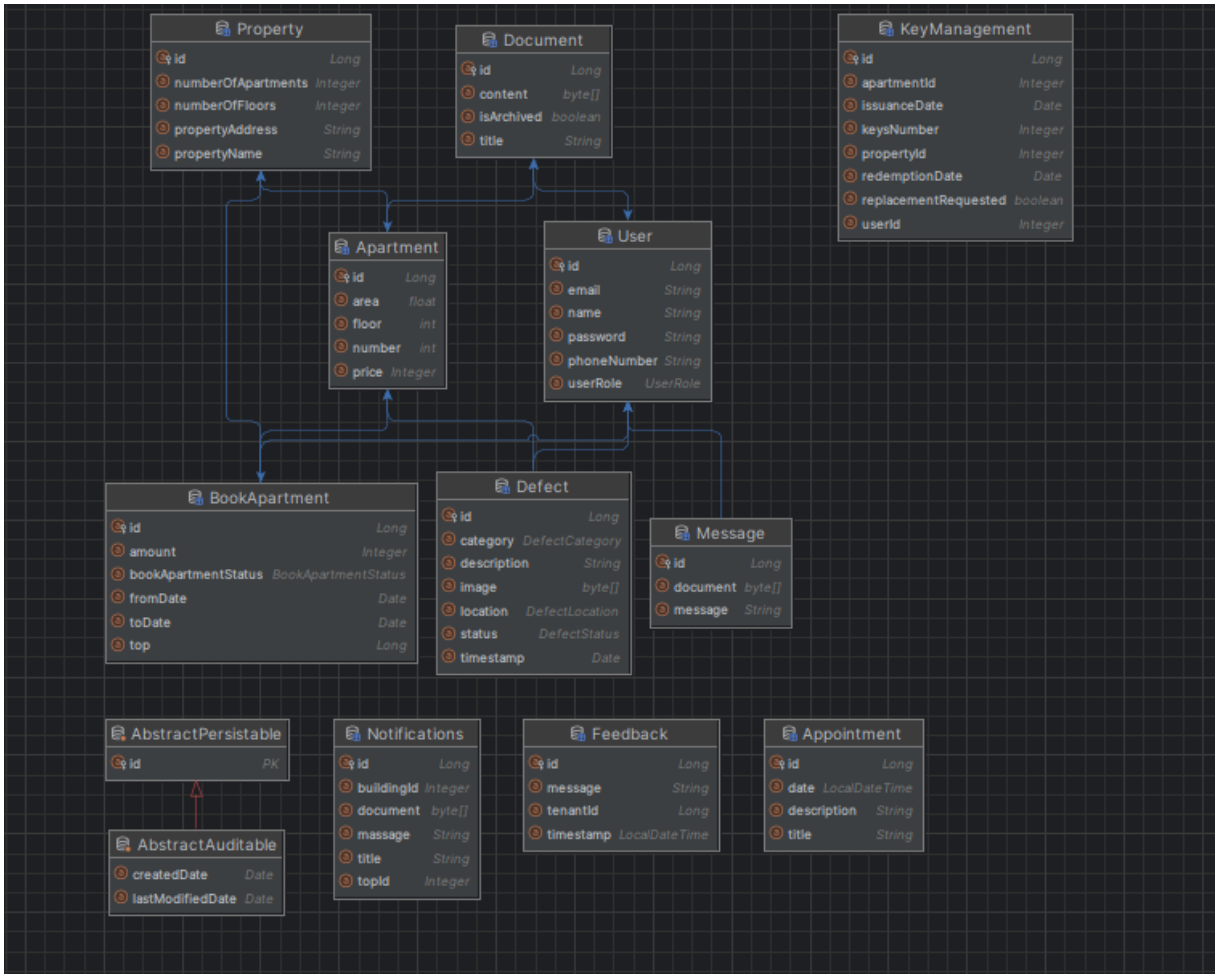
- GET /api/tenant/v1/{userId}
- GET /api/tenant/v1/apartment/bookings/{userId}
- POST /api/tenant/v1/defect
- GET /api/tenant/v1/document/{userId}/{apartmentId}
- GET /api/tenant/v1/document/{userId}/{apartmentId}/tenant
- GET /api/tenant/v1/document/{userId}/{apartmentId}/general
- GET /api/tenant/v1/document/{userId}/{apartmentId}/archived
- GET /api/tenant/defects/user/{userId}
- POST /api/tenant/v1/distribute
- GET /api/tenant/v1/notification/top/{topId}
- GET /api/tenant/appointment/all
- POST /api/tenant/feedback

```
public class TenantController {

    private final TenantService tenantService;
    private final DefectService defectService;
    private final DocumentRepository documentRepository;
    private final NotificationsRepository notificationsRepository;
    private final PropertyManagementService propertyManagementService;
    private final AppointmentService appointmentService;
    private final FeedbackService feedbackService;

    @GetMapping("/v1/{userId}")
    public ResponseEntity<UserDto> getTenantById(@PathVariable long userId) {
        UserDto userDto = tenantService.getTenantById(userId);
        if (userDto != null) return ResponseEntity.ok(userDto);
        return ResponseEntity.notFound().build();
    }

    @GetMapping("/v1/apartment/bookings/{userId}")
    public ResponseEntity<?> getBookingsByUserId(@PathVariable Long userId) {
        return ResponseEntity.ok(tenantService.getBookingsByUserId(userId));
    }

    @PostMapping(value = "/v1/defect")
    public ResponseEntity<String> reportDefect(
            @RequestPart("defectDto") DefectDto defectDto,
            @RequestParam("image") MultipartFile image
```

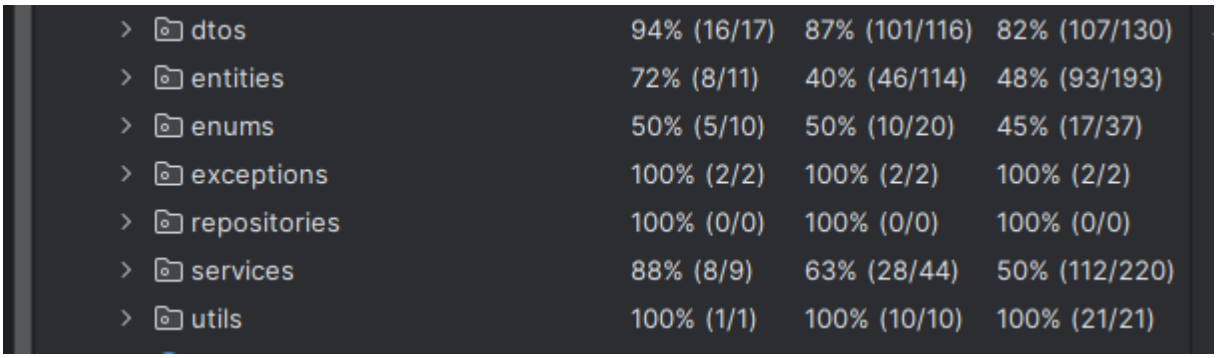
Database

Short Description of the ER Diagram

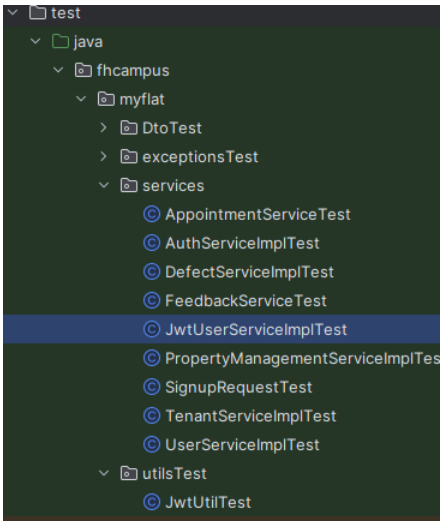
- **Property**: Contains multiple apartments.
- **Apartment**: Belongs to a property, associated with users, can be booked.
- **User**: Associated with apartments, sends messages.
- **BookApartment**: Represents apartment bookings.
- **Defect**: Reports issues, linked to messages.
- **Message**: Sent by users, can report defects.
- **Document**: Holds content, potentially linked to messages or defects.
- **KeyManagement**: Manages apartment keys, associated with users and apartments.
- **Notifications**: Notifies users, related to buildings and documents.
- Feedback: User feedback, linked to messages.
- Appointment: Schedules events, linked to users or properties.
- AbstractPersistable and AbstractAuditable: Base entities for persistence and auditing.

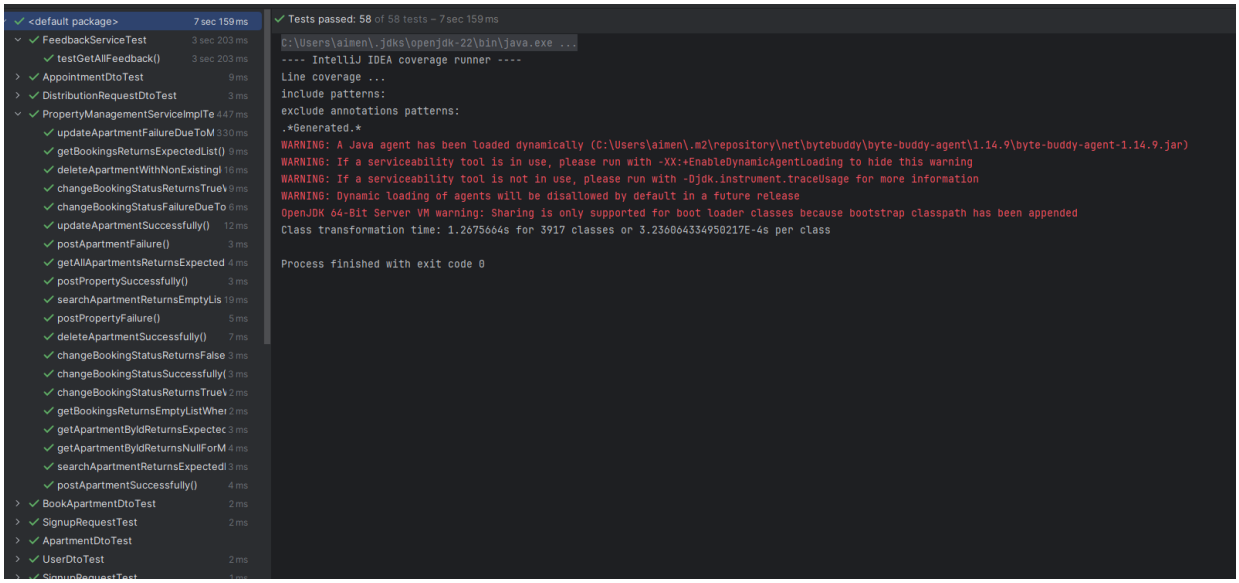


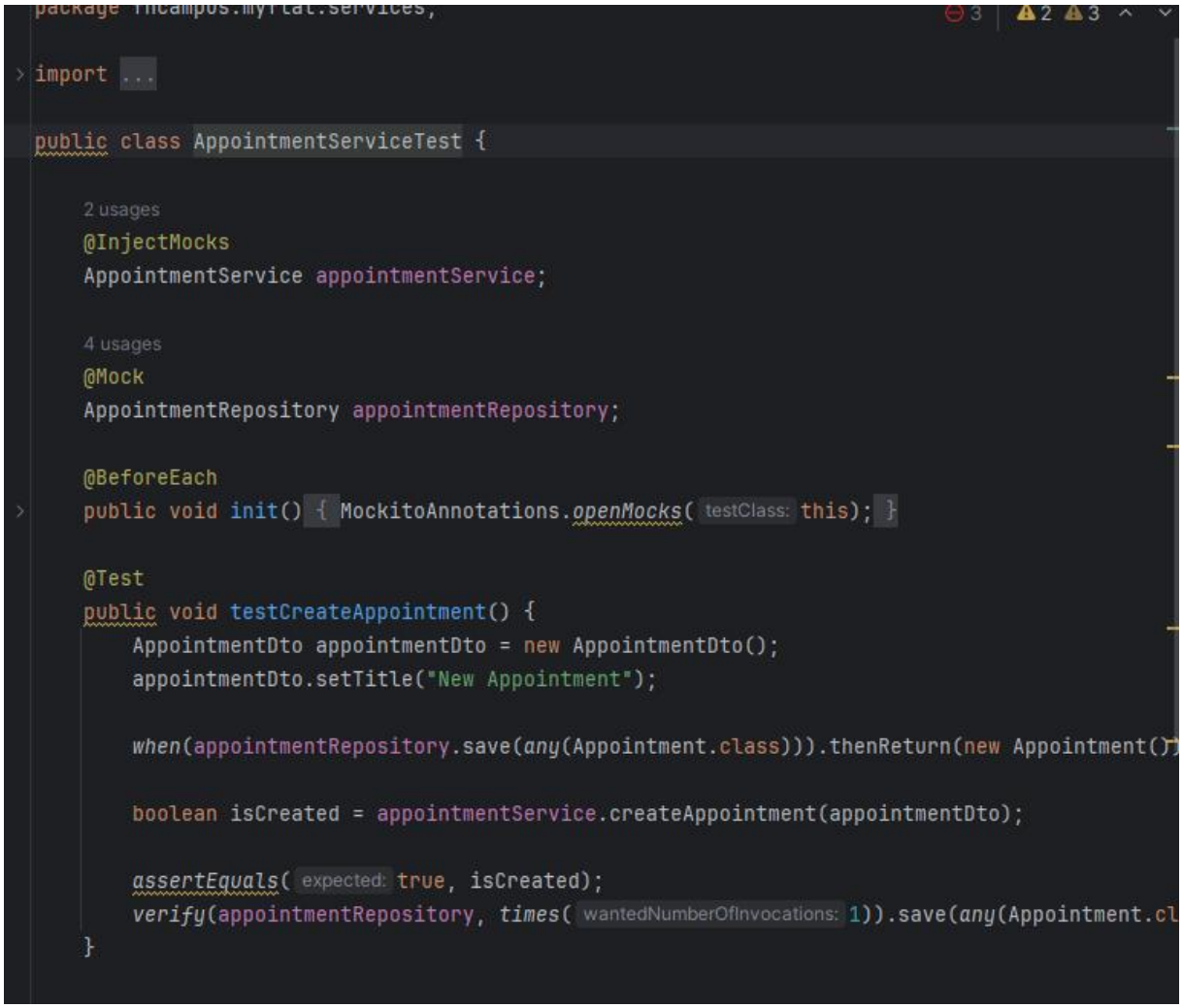
TEST-CASES



- Database Driver: This is a software component that allows your application to interact with the database. The driver acts as a bridge between the application and the database. For example, if you're using a MySQL database, you would use the MySQL JDBC driver.
- Connection: This is the link between your application and the database. You establish a connection using a connection string, which includes the database URL and credentials (username and password). Once the connection is established, you can send SQL commands to the database and retrieve results.
- SQL Statements: These are the commands that you send to the database to perform operations. This could be anything from creating a table, inserting data, updating data, retrieving data, etc. In Java, you can create these statements using the Statement or PreparedStatement classes.
- Result Set: When you execute a query that retrieves data, the data is returned in a ResultSet object. You can iterate over this object to access the returned data.
- Closing the Connection: After you're done interacting with the database, it's important to close the connection and other resources like Statement and ResultSet to free up resources.

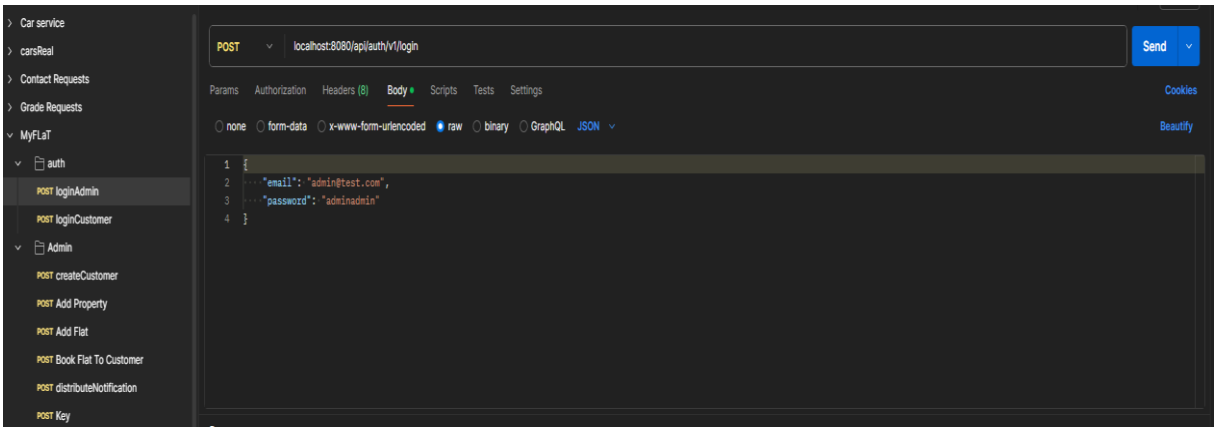


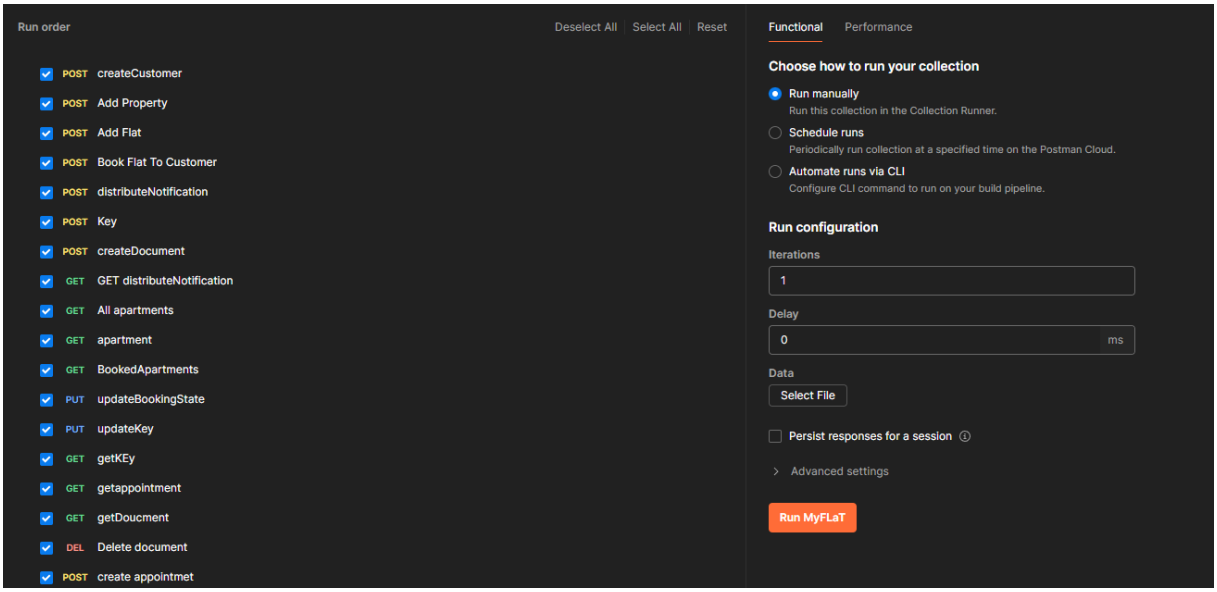

✓ <default package> 7 sec 159 ms
✓ FeedbackServiceTest 3 sec 203 ms
✓ testGetAllFeedback() 3 sec 203 ms
✓ AppointmentDtoTest 9 ms
✓ DistributionRequestDtoTest 3 ms
✓ PropertyManagementServiceImplTe 447 ms
✓ updateApartmentFailureDueToM 330 ms
✓ getBookingsReturnsExpectedList() 9 ms
✓ deleteApartmentWithNonExistingI 16 ms
✓ changeBookingStatusReturnsTrueV 9 ms
✓ changeBookingStatusFailureDueTo 6 ms
✓ updateApartmentSuccessfully() 12 ms
✓ postApartmentFailure() 3 ms
✓ getAllApartmentsReturnsExpected 4 ms
✓ postPropertySuccessfully() 3 ms
✓ searchApartmentReturnsEmptyLis 19 ms
✓ postPropertyFailure() 5 ms
✓ deleteApartmentSuccessfully() 7 ms
✓ changeBookingStatusReturnsFalse 3 ms
✓ changeBookingStatusSuccessfully(3 ms
✓ changeBookingStatusReturnsTrueV 2 ms
✓ getBookingsReturnsEmptyListWher 2 ms
✓ getApartmentByIdReturnsExpectec 3 ms
✓ getApartmentByIdReturnsNullForM 4 ms
✓ searchApartmentReturnsExpectedI 3 ms
✓ postApartmentSuccessfully() 4 ms
✓ BookApartmentDtoTest 2 ms
✓ SignupRequestTest 2 ms
✓ ApartmentDtoTest
✓ UserDtoTest 2 ms
Tests passed: 58 of 58 tests – 7 sec 159 ms
C:\Users\aimen\.jdks\openjdk-22\bin\java.exe ...
---- IntelliJ IDEA coverage runner ----
Line coverage ...
include patterns:
exclude annotations patterns:
.*Generated.*
WARNING: A Java agent has been loaded dynamically (C:\Users\aimen\.m2\repository\net\bytebuddy\byte-buddy-agent\1.14.9\byte-buddy-agent-1.14.9.jar)
WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoading to hide this warning
WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.traceUsage for more information
WARNING: Dynamic loading of agents will be disallowed by default in a future release
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
Class transformation time: 1.2675664s for 3917 classes or 3.236064334950217E-4s per class
Process finished with exit code 0

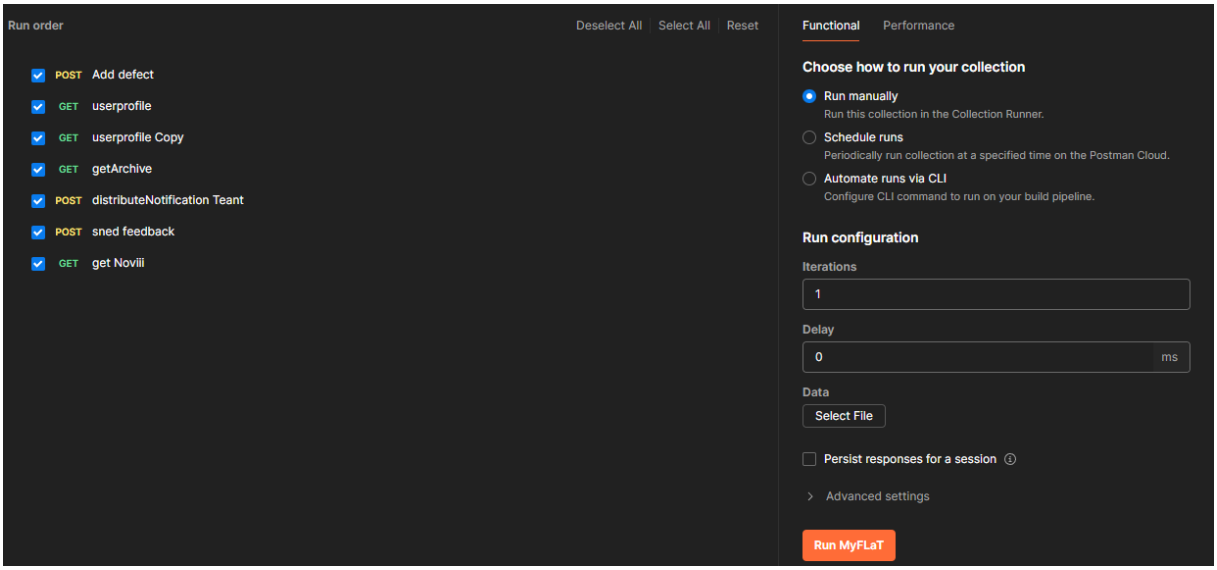

import ...
public class AppointmentServiceTest {
2 usages
@InjectMocks
AppointmentService appointmentService;
4 usages
@Mock
AppointmentRepository appointmentRepository;
@BeforeEach
public void init() { MockitoAnnotations.openMocks(testClass: this); }
@Test
public void testCreateAppointment() {
AppointmentDto appointmentDto = new AppointmentDto();
appointmentDto.setTitle("New Appointment");
when(appointmentRepository.save(any(Appointment.class))).thenReturn(new Appointment()
boolean isCreated = appointmentService.createAppointment(appointmentDto);
assertEquals(expected: true, isCreated);
verify(appointmentRepository, times(wantedNumberOfInvocations: 1)).save(any(Appointment.cl
}

TEST API

Using postman

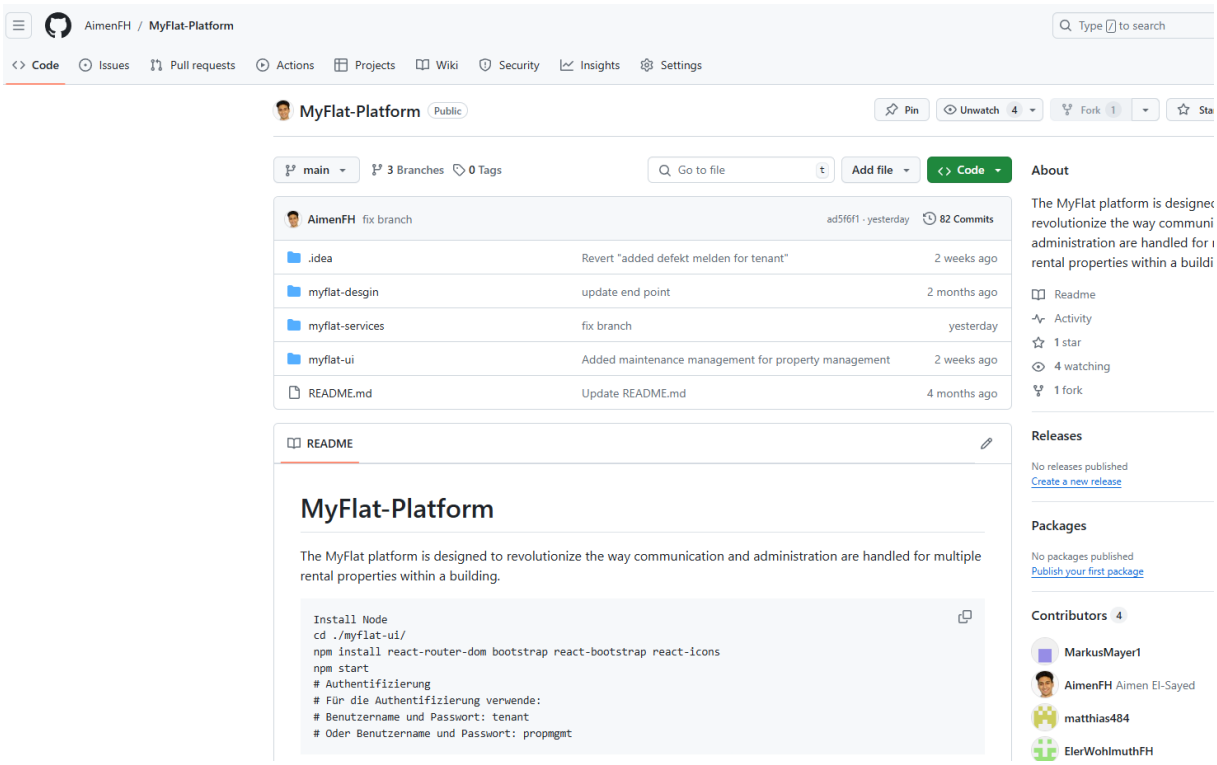






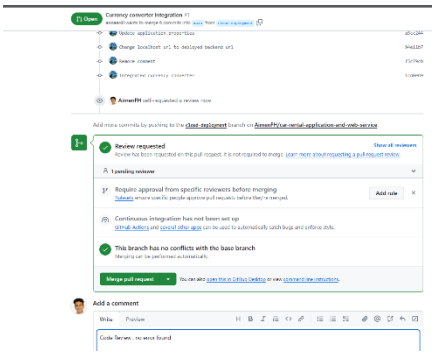
GIT Repository

Link: [AimenFH/MyFlat-Platform: The MyFlat platform is designed to revolutionize the way communication and administration are handled for multiple rental properties within a building. (github.com)]



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.

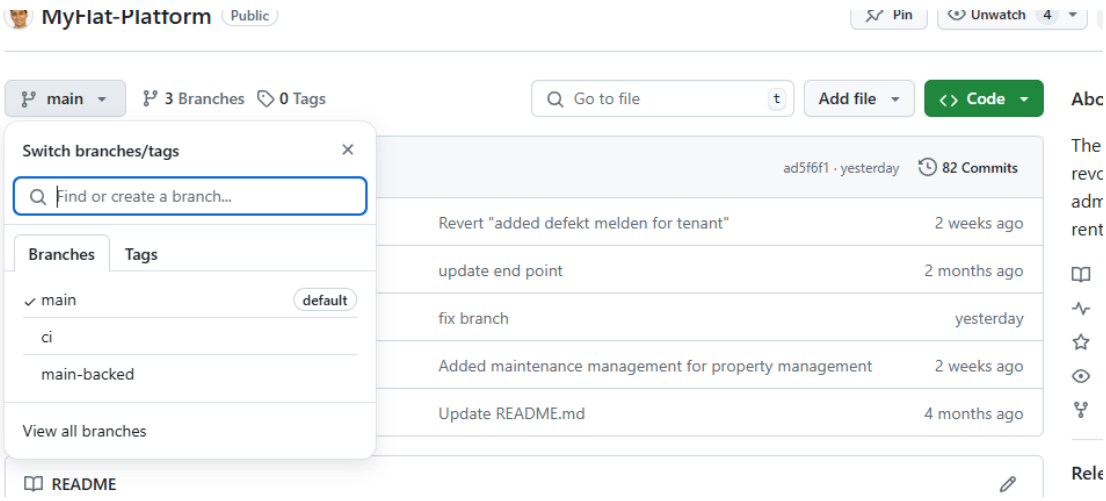
3. **Unit Testing:**
    - **JUnit:** Write comprehensive unit tests for all critical components using JUnit.
    - **Mockito:** Use Mockito to create mocks and stubs for testing interactions between components.
    - **Spring Test:** Leverage Spring Boot's testing capabilities for integration tests.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

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.

```
//////////////////////////// distribute notification
@PostMapping("/v1/distribute")
public ResponseEntity<?> distributeNotification(@RequestBody DistributionRequestDto
    boolean success = propertyManagementService.distributeNotification(distributionR
    if (success)
        return ResponseEntity.status(HttpStatus.CREATED).build();
    return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
}

@GetMapping("v1/notifications")
public ResponseEntity<?> getNotifications(@RequestParam(required = false) Integer bu
    try {
        List<Notifications> notifications = propertyManagementService.getNotificatio
        return ResponseEntity.ok(notifications);
    } catch (NoNotificationsFoundException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(e.getMessage());
    }
```

Lessons learned.

Backend

Proper Credential and Secret Management:

Managing OAuth credentials and secrets accurately is crucial for security and functionality. Ensuring the correct setup and storage of these credentials can prevent authentication errors. Utilizing tools like AWS Secrets Manager or Azure Key Vault for managing secrets can enhance security. Regularly rotating credentials and reissuing tokens can also help maintain smooth OAuth functionality.

Version Compatibility:

Ensuring compatibility between different parts of the tech stack is essential. For instance, frontend frameworks like Angular and Node.js must have compatible versions to work seamlessly together. Similarly, maintaining compatibility between Spring Boot and Spring Security versions in the backend is critical. This requires careful planning and regular updates to keep dependencies aligned.

Simplicity and Design Focus:

Keeping the design simple and spending more time in the planning phase can prevent complexities later in the development process. Starting with a minimal viable product (MVP) approach and gradually adding features can avoid the creation of unnecessary services and reduce complexity. This strategy also aids in better cloud deployment and resource management.

Service Management:

Starting with many services led to challenges in cloud deployment and maintenance. It was necessary to restructure and consolidate services, which was time-consuming. The lesson learned is to start small, only adding essential services initially and expanding as needed based on actual requirements.

Debugging Skills:

Deployment phases often revealed numerous bugs and errors. Spending significant time on platforms like Stack Overflow for solutions helped improve debugging skills. Learning to read and interpret console errors effectively is a valuable skill gained during this project.

Testing Approach:

Implementing tests for small parts of the application incrementally is more effective than writing tests for large blocks of code. This approach makes it easier to identify and fix errors early, ensuring more stable and reliable code. Writing comprehensive unit tests using JUnit and Mockito for backend services and integration tests for API endpoints provided significant benefits.

Documentation and Communication:

Clear and comprehensive documentation is essential. Both user manuals for customers and administrators and technical documentation for developers were crucial in this project. Using tools like Postman for API documentation and maintaining up-to-date technical documentation helped ensure clarity and ease of use.