**Основи об'єктно-орієнтованого програмування**

**Лабораторна робота № 1a**

**Моделювання з використанням UML**

**Вступ**

Для опису структури існуючого коду та його рефакторінгу було обрано вже існуючу програму, власний навчальний проєкт з першого семестру навчання. Код має достатньо складну будову для аналізу, покращень та створення UML діаграм.

Проєкт реалізує веб-сервіс для бронюванням кімнат у готелі. Сайт розроблено на з застосуванням бібліотеки React.js (JavaScript) для фронтенду та фреймворку Spring Boot 3 (Java) для бекенду. Для даної лабораторної роботи ми вважаємо, що найдоцільніше було б розглядати бекенд частину проєкту, оскільки Java як мова програмування за своєю природою об’єктно-орієнтована.

**Аналіз старого коду в проєкті**

Бекенд частина складається з багатьох класів та інтерфейсів, що виконують різні функції. Основними елементами програми є:

* **Основний клас застосунка**: Відповідає за запуск проєкту
* **Моделі**: Представляють дані
* **Репозиторії**: Забезпечують доступ до бази даних (У нашому випадку було використано PostgreSQL)
* **Контролери**: Обробляють HTTP-запити від React.js частини проєкту

Ця структура є типовою для більшості Spring Boot проєктів.

### **Аналіз дотримання принципів архітектури та проектування:**

#### **Основі принципи ООП**

1. **Інкапсуляція**:
   * Дані всіх класів захищені приватними полями, доступ до яких надається через геттери та сеттери.
2. **Успадкування**:
   * Успадкування вбудованих у фреймворк класів та інтерфейсів є невід’ємною частиною процесу розбудови Spring Boot застосунку. Приклад успадковуваного абстрактного класу: OncePerRequestFilter. Приклади імплементованих інтерфейсів: UserDetails, JpaRepository.
3. **Поліморфізм**:
   * Поліморфізм досягається через використання інтерфейсів і залежностей на базі інтерфейсів. Наприклад, імплементація вбудованого інтерфейсу JpaRepository,що використовується всіма класами-репозиторіями.

#### **Загальні принципи**

**KISS (Keep It Simple, Stupid)**:

* Виконання всіх CRUD операцій чітко структуровано відповідно до наданих фреймворком інструментів. Код розбито на класи та функції настільки, наскільки це можливо й доцільно, відповідно до

**YAGNI (You Aren't Gonna Need It)**:

* Код не містить непотрібної функціональності. Однак присутні частини, що досі до кінця не реалізовані, а отже не на даний момент не використовуються.

**DRY (Don't Repeat Yourself)**:

* Прикладом дотримання цього принципу є факт того, що всі методи доступу до даних централізовано в репозиторії. Код контролера не повторює логіку доступу до даних, оскільки кожна частина проєкту робить виключно свою частину роботи. Однак, присутня проблема дублювання рядкових констант.

**Розподіл функціональності**:

* Кожен клас виконує конкретну роль: модель, репозиторій, контролер.

**Можливість підтримки**:

* Код легко підтримувати завдяки простій архітектурі та чіткій структурі.

#### **Принципи SOLID**

1. **Single Responsibility Principle (SRP)**:
   * Кожен клас виконує одну конкретну задачу: модель представляє дані, репозиторій управляє доступом до даних, контролер обробляє HTTP-запити.
2. **Open/Closed Principle (OCP)**:
   * Класи легко розширити новою функціональністю без зміни існуючого коду (наприклад, додавання нових методів до репозиторію).
3. **Liskov Substitution Principle (LSP)**:
   * Використання інтерфейсів (наприклад, JpaRepository) гарантує, що заміна однієї реалізації іншою не порушить роботу програми.
4. **Interface Segregation Principle (ISP)**:
   * Інтерфейси невеликі та спеціалізовані. JpaRepositoryзабезпечує тільки необхідну функціональність.
5. **Dependency Inversion Principle (DIP)**:
   * Наприклад, контролери залежать від інтерфейсу репозиторію, а не від конкретної реалізації.

#### **Патерни проектування**

1. **Repository Pattern**:
   * Використання класів-репозиторіїв для доступу до даних.
2. **Controller Pattern**:
   * Використання класів-контролерів для обробки HTTP-запитів.
3. **Dependency Injection**:
   * Досягається використанням анотацій типу @Component, @Repository, @Service, @Autowired, @Bean, тощо.

### **Висновки**

Проект дотримується багатьох принципів проектування та архітектурно правильно скомпонований, забезпечуючи простоту, розширюваність та підтримуваність коду. Використання Spring Boot не тільки допомагає, але й змушує реалізовувати і підтримувати основні принципи ООП та загальні принципи проектування.

Однак, було помічено ряд незначних недоліків у реалізації, які можна виправити, що будуть перелічені в наступному розділі.

**Покращення в реалізації програми**

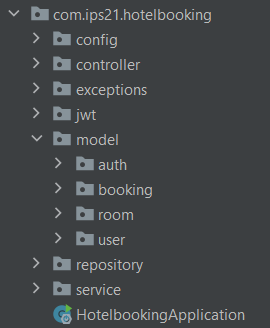
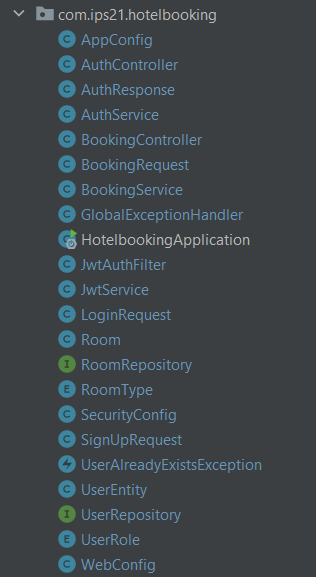
Згідно з висновками, наведеними після аналізу коду старої версії проєкту, бекенд частина веб-сервісу не потребує значних змін, у тому числі архітектурних. Сам проєкт замалий для використання в ньому яких-небудь патернів, окрім тих, що вбудовані в сам фреймворк. Тим не менш, нам вдалося придумати, як покращити якість коду в проєкті. Відповідні пропозиції та зміни буде наведено далі.

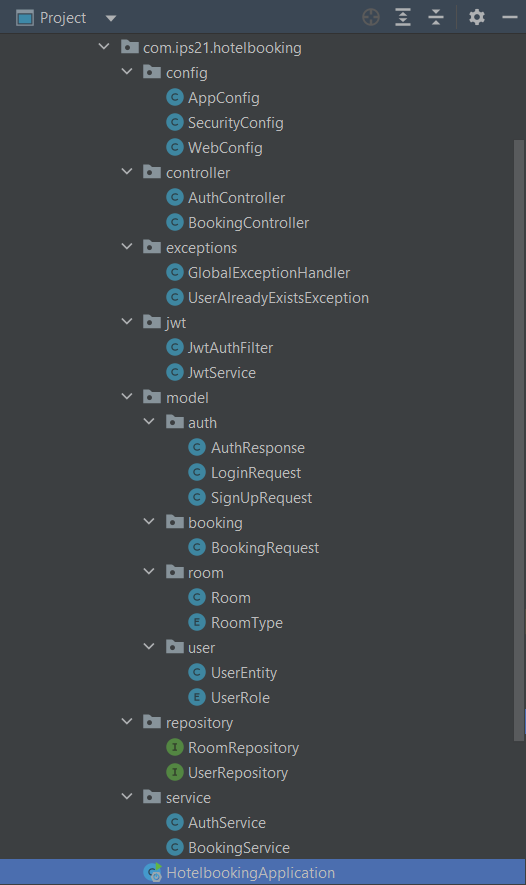
*(Не всі зміни коду будуть представлені на скриншотах, оскільки це просто займе забагато місця)*

1. **Реорганізація файлів проєкту.**

Першим очевидним недоліком була відсутність організації source-файлів у проєкті. Тому їх було чітко розділено на відповідно категорії.

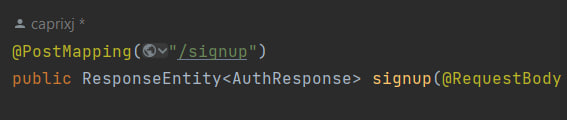
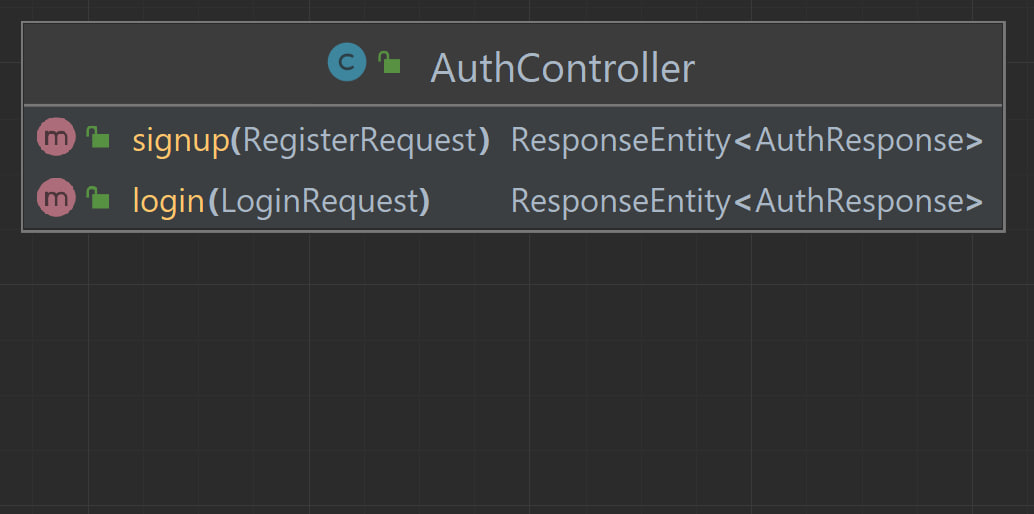
*1-ий скриншот: до змін. 2-ий та 3-ій скриншоти: після змін.*

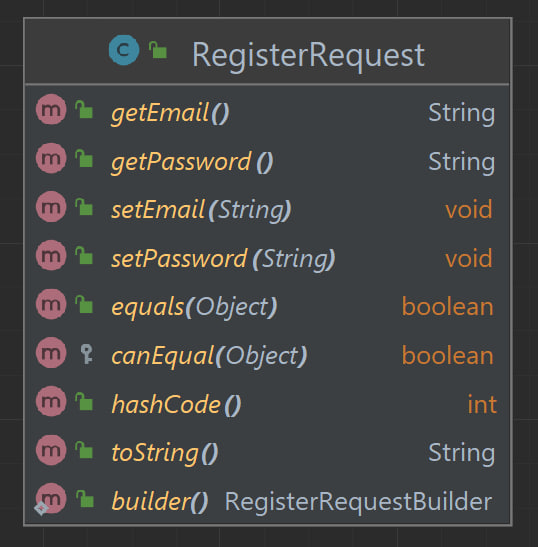
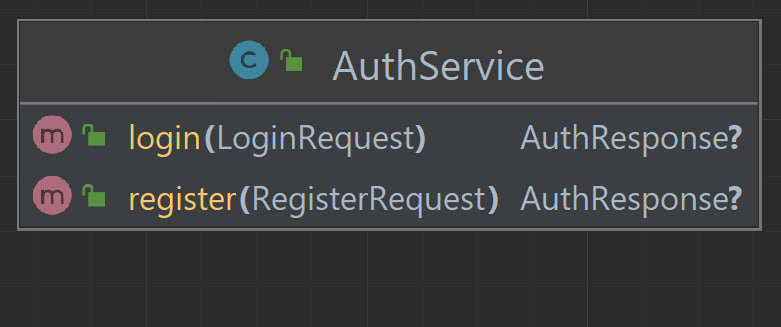




1. **Послідовність у виборі назв компонентів програми.**

Виявлено та виправлено розходження в назвах функцій та класів однакового логічного змісту *(Register vs Sign Up)*:

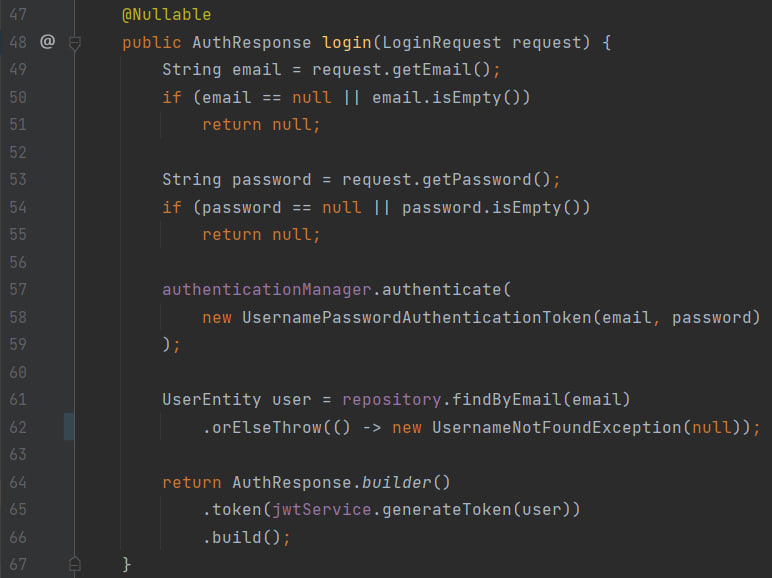
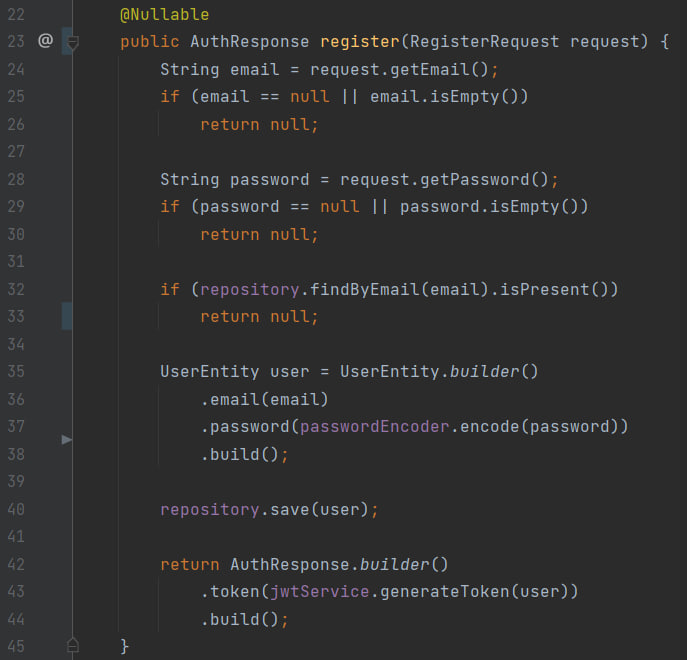




Результатом рефактиронгу стало викорінення терміну Register та заміною його на SignUp у назвах класів та функцій.

1. **Впровадження централізованого перехоплення та опрацювання виключень (Exceptions). Перевірка коректності та адекватності вхідних даних.**

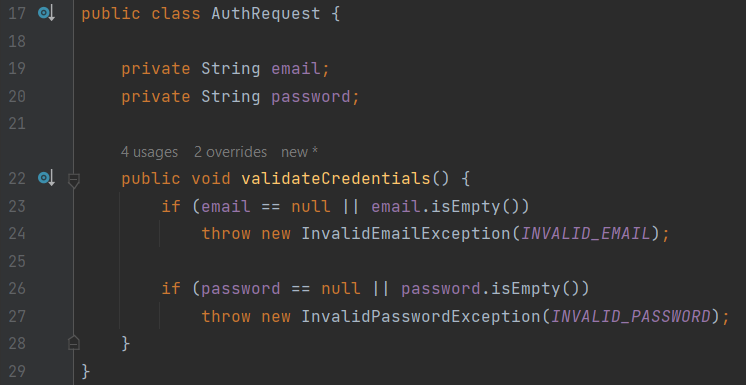
У старій версії програми маємо дві проблеми. Перша: повторення коду під час перевірки вхідних даних. Друга: зловживання командою return null; та непослідовне використання довільних виключень як результат опрацювання некоректних даних/запитів з клієнту.

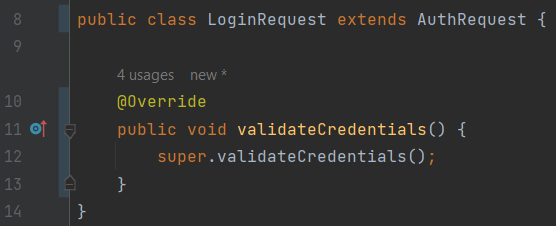


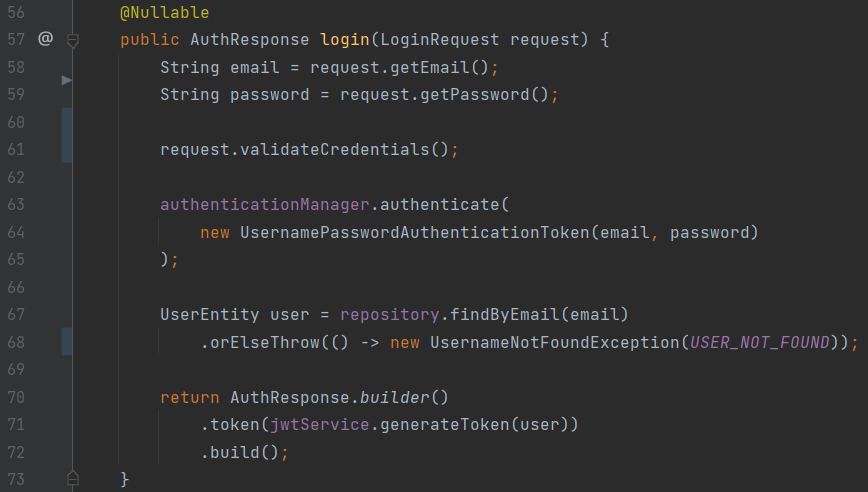


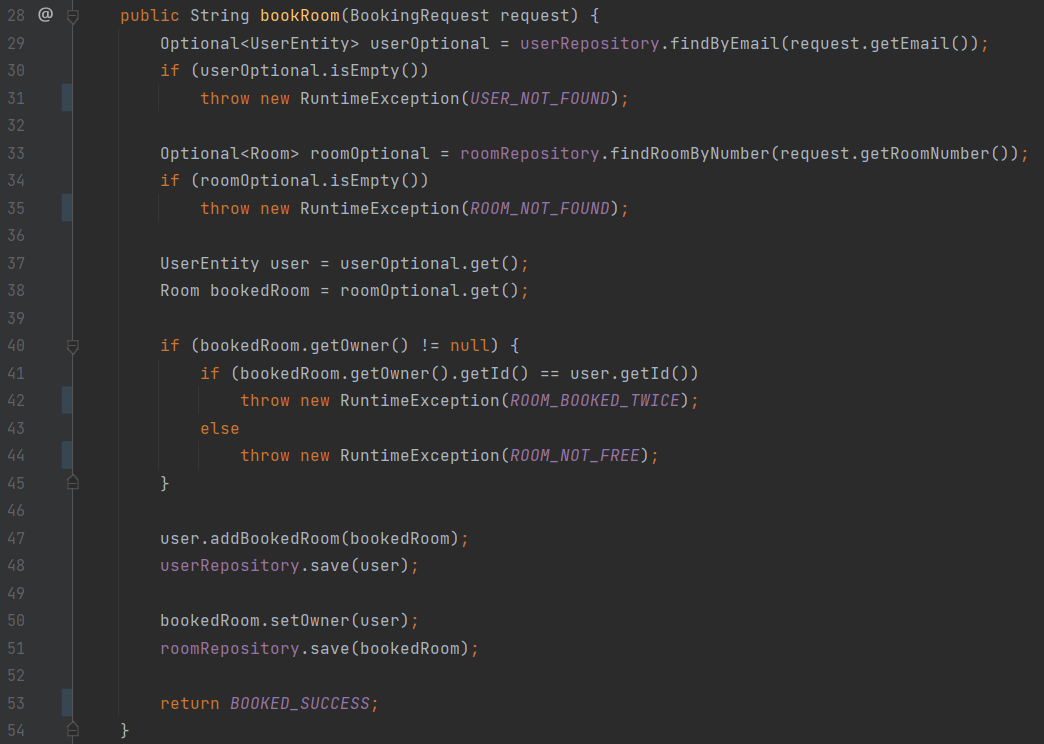
Дана реалізація є неприпустимою для підтримання чистоти та коду, тому беремо за мету це виправити наступним чином.

Перша проблема вирішується успадкуванням: створюємо публічний метод validateCredentials та новий суперклас AuthRequest, у якому цей метод буде реалізовано, аби використовувати через LoginRequest та SignUpRequest:

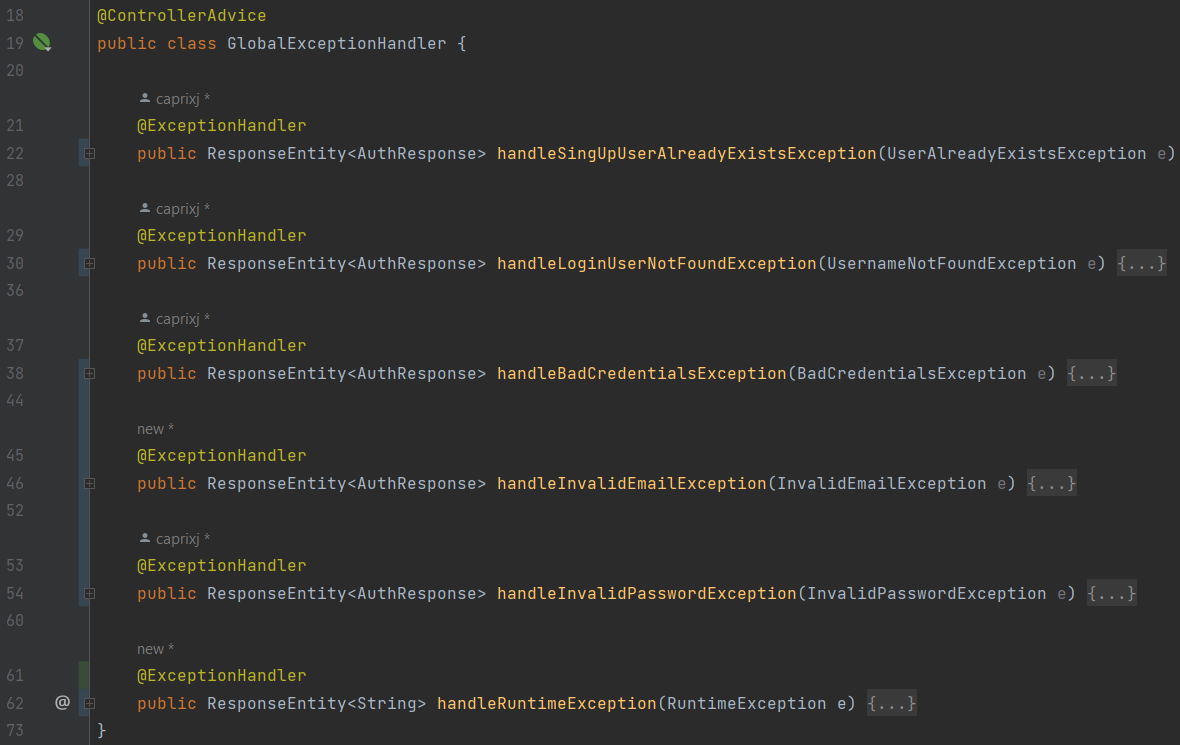


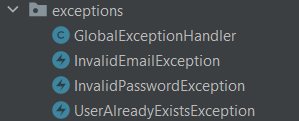






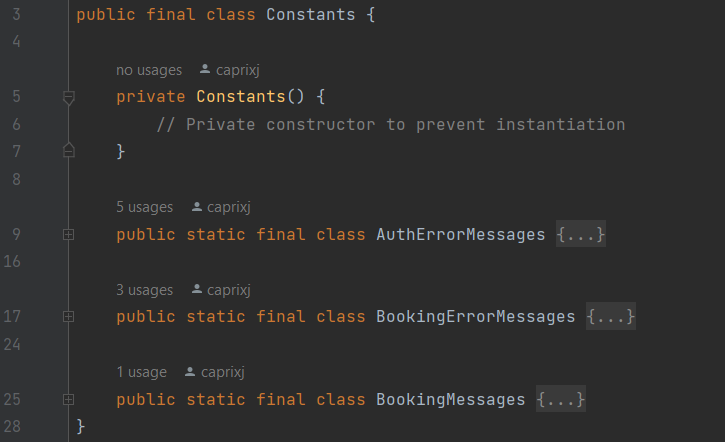
Вирішенням другої проблеми є створення спеціального класу з анотацією @ControllerAdvice, що допоможе нам обробляти всі потрібні нам виключення. Замінемо повернення null на виключення (видно на попередніх скриншотах) і створимо власні виключення, де це виявиться доцільним:





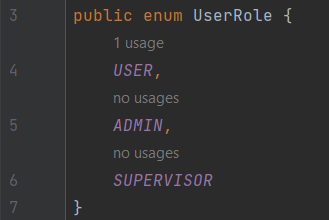
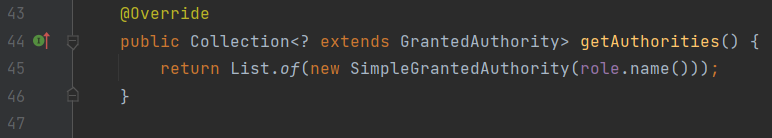
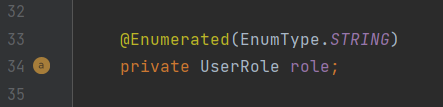
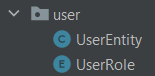
1. **Створення класу, що зберігатиме дубльовані рядкові константи.**

(Використання цього новоствореного класу видно на попередніх скриншотах)



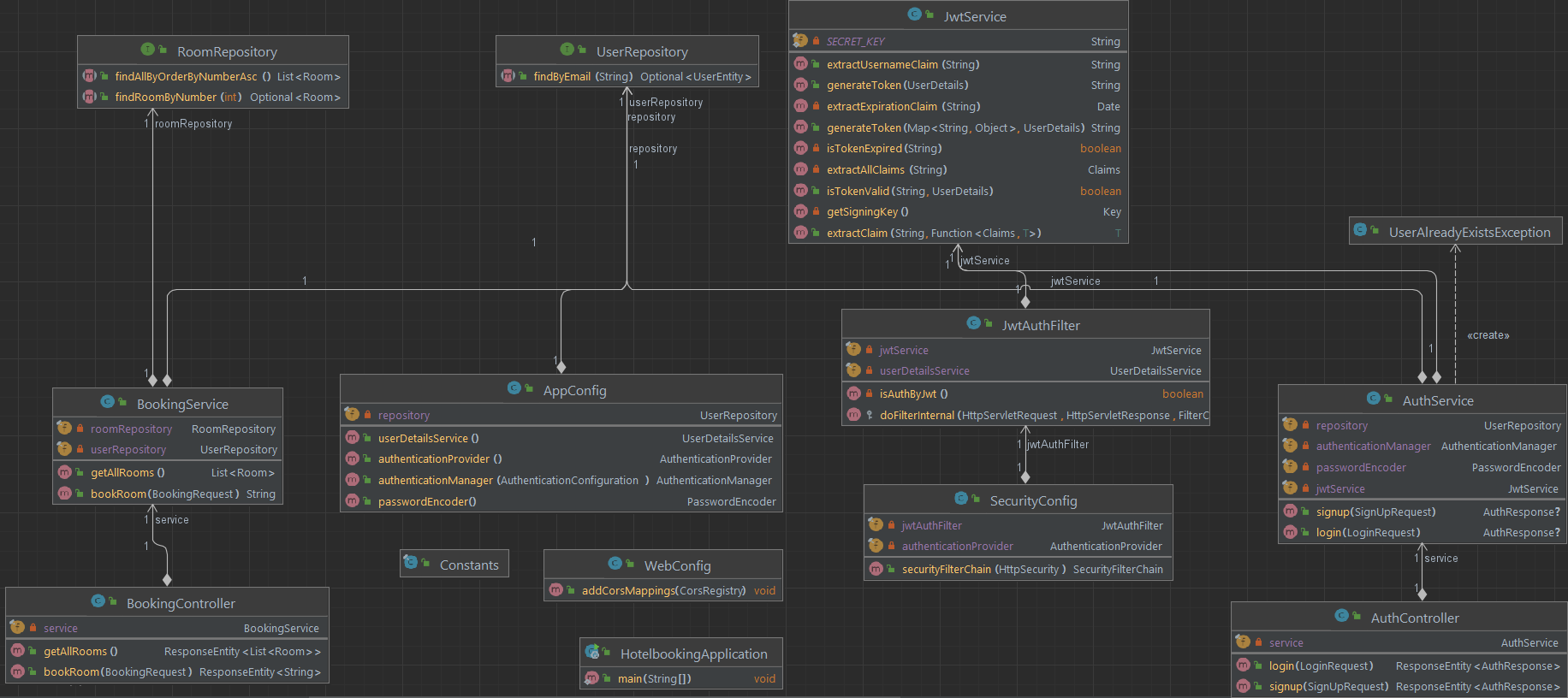


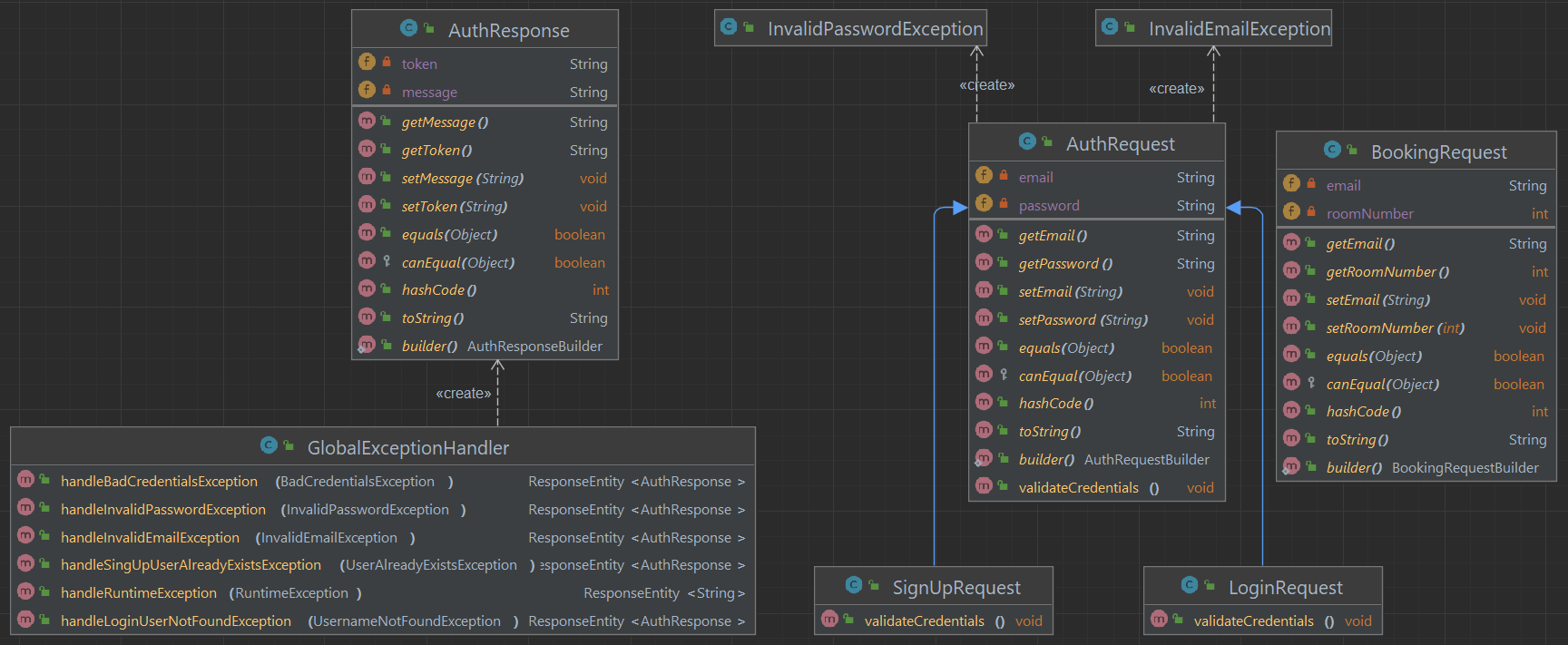
1. Також у процесі дописування проєкту було створено enum UserRole, додано відповідне поле в UserEntity та трохи змінено код, аби в майбутньому додати функціональність розділення користувачів на клієнтів та адміністраторів.

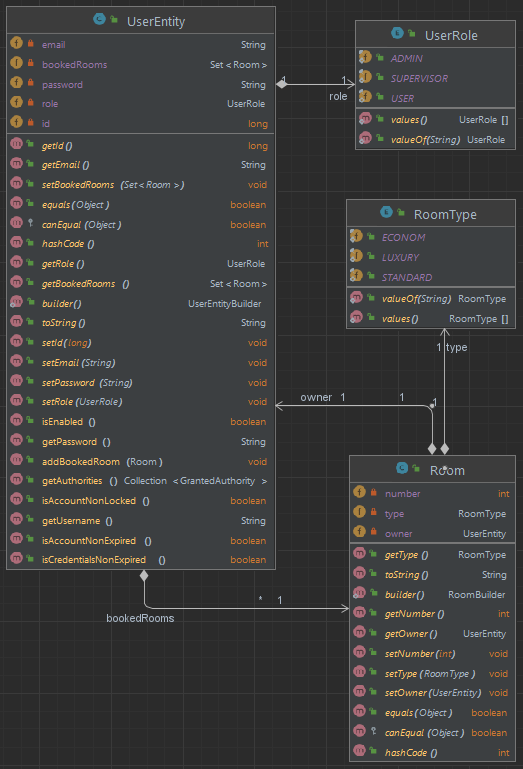


**UML діаграми**

Для наступних UML діаграми було використано вбудований інструментарій інтегрованого середовища розробки Intellij Idea. Дані діаграми описують структуру покращеної версії програми: усі використані класи, їхні поля, методи, взаємовідносини між класами:







**Unit Testing за допомогою JUnit 5 та Mockito**

У якості прикладу одного з unit тестів можна привести класс AuthServiceTests.

Функція testSignupSuccess перевіряє, чи метод реєстрації signup

правильно реєструє користувача, шифрує його пароль, зберігає UserEntity, генерує JWT токен і повертає коректний AuthResponse.

Функція testSignupUserAlreadyExists перевіряє, чи метод реєстрації signup створює виключення UserAlreadyExistsException, коли користувач із вказаною електронною поштою вже існує.

