math_2023_020

Міністерство освіти і науки України

Чернівецький національний університет

імені Юрія Федьковича

Факультет математики та інформатики
кафедра математичного моделювання

Розробка вебдодатку для майстерні по ремонту та обслуговуванню кавових апаратів

Кваліфікаційна робота

Рівень вищої освіти - другий (магістерський)

Виконав:
студент 6 курсу, групи 607 .
Пелепяк Віктор Володимирович.
Керівник:
кандидат фізико-математичних наук,
доцент Юрченко І.В.

До захисту допущено
на засіданні кафедри
протокол № 9 від 5 грудня 2023 р.
Зав. кафедрою ____________ проф. Черевко І.М.

Чернівці – 2023

АНОТАЦІЯ

У кваліфікаційній роботі спроектовано та реалізовано вебдодаток, що дозволяє здійснювати ведення обліку кавових апаратів та записів їх обслуговування.

Детально описано всі аспекти проектування програмного продукту, його програмної реалізації та використання.

Розробку програмного продукту виконано засобами середовища IntelliJ IDEA, мови програмування Java, скриптової мови Typescript, а також фреймворків Spring Boot та Angular.

ABSTRACT

In the qualification work, a web application was designed and implemented, which allows keeping records of coffee machines and records of their maintenance.

All aspects of product design, its software implementation and application using are described in detail.

The software product was developed using the IntelliJ IDEA environment, the Java programming language, the Typescript scripting language, as well as the Spring Boot and Angular frameworks.

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів наукових досліджень інших авторів мають посилання на відповідне джерело.

__________________ В.В. Пелепяк
(підпис)

ЗМІСТ

ВСТУП **5**

РОЗДІЛ 1. ТЕОРЕТИЧНА ЧАСТИНА **8**

1.1. Фреймворк Spring Boot 8

1.1.1. Основні модулі фреймворку. *9*

1.1.2. Spring Boot Starter Web *10*

1.1.3. Spring Data JPA *11*

1.2. Генерація додатку 12

1.2.1. Генерація проекту засобами start.spring.io *12*

1.2.2. Генерація проекту засобами середовища IntelliJ IDEA *14*

1.3. СУБД PostgreSQL 15

1.4. Фреймворк Angular 16

РОЗДІЛ 2. ПРАКТИЧНА ЧАСТИНА **19**

2.1. Проектування програмного продуту 19

2.1.1. Історії користувачів *19*

2.1.2. Концептуальна модель «сутність-зв'язок» *21*

2.2. Розробка програмного продуту 23

2.2.1. Класи моделей *23*

2.2.2. Рівень доступу до баз даних. *27*

2.2.3. Сервісний рівень. *28*

2.2.4. Рівень контролерів. *29*

2.3. Використані шаблони проектування 34

2.3.1. Одинак *34*

2.3.2. Замісник *35*

2.3.3. Стратегія *35*

РОЗДІЛ 3. ІНСТРУКЦІЯ КОРИСТУВАЧА ТА ОПИС ДОДАТКУ **37**

3.1. Реєстрація нових користувачів 37

3.2. Підтвердження адреси електронної пошти 38

3.3. Вхід в обліковий запис 39
3.4. Адміністрування 40
3.5. Додавання кавових апаратів до реєстру та перегляд записів 41
3.6. Додавання придбаних кавових апаратів 44
ПЕРЕЛІК ПОСИЛАНЬ 46
ДОДАТКИ 47
Додаток А. Схема Бази даних. 48
Додаток Б. Програмний код класу EmailServiceImpl 49
Додаток В. Програмний код класу CoffeeMachineController 51
Додаток Г. Програмний код класу CoffeeMachineServiceImpl 53

ВСТУП

Актуальність роботи. Невід’ємною складовою сучасного життя виступають інформаційні технології. В наш час стає надзвичайно важко уявити яке-небудь підприємство, котре б не використовувало жодного програмного забезпечення. Особливо популярною така тенденція стала в умовах карантину, локдауну та стану війни, коли чи не найважливішу роль відіграє можливість працівників здійснювати свою діяльність у віддаленому режимі. Абсолютно звичним стало використання веб-технологій та ресурсів для спілкування, роботи, ведення обліку та іншої діяльності. Чи не найголовнішим аспектом є повноцінна адаптація звичних (повсякденних) робочих процесів до таких умов.

Збір, аналіз та систематизація інформації є одним із таких видів діяльності. Особливої цінності набуває можливість здійснення даних операцій з подальшим збереженням інформації у віддалені сховища даних, що забезпечує можливість доступу до них з будь-якого місця. Традиційними засобами, що використовуються для виконання задач цього типу можна вважати BAF, Odoo чи Dilovod. Проте, дані сервіси не можуть виступати у ролі повністю універсального рішення для тих аспектів діяльності, коли необхідно забезпечити використання чіткої структури записів, яка є специфічною для конкретної предметної області.

Природнім постало питання проектування, розробки та впровадження власної системи, яка б забезпечувала можливість ведення обліку даних специфічного формату, згідно з вимогами предметної області.

Метою кваліфікаційної роботи є проектування та розробка вебдодатку для управління майстернею по ремонту та обслуговуванню кавових апаратів з використанням технологій мови програмування Java, скриптової мови програмування Typescript, а також фреймворків Spring Boot та Angular.

Завдання роботи – забезпечення майстерні по ремонту та обслуговуванню кавових апаратів вебдодатком для ведення обліку пристроїв та записів щодо їх обслуговування.

Для досягнення мети розв’язано наступні задачі:

- проведено збір та узгодження з замовником вимог до функціоналу вебдодатку;
- завершено проектування класів-моделей, з урахуванням їх реляційного відображення;
- реалізовано бекенд-частину додатку, що імплементує необхідний функціонал засобами мови програмування Java та фреймворку Spring Boot;
- реалізовано фронтенд-частину додатку, що орієнтована на використання у браузері як веб-сторінки засобами скриптової мови програмування Typescript та фреймворку Angular;
- здійснено тестування створеного додатку.

Об’єктом дослідження є вебдодатки для ведення обліку продукції.

Наукова новизна і практичне значення роботи. Наукова новизна полягає у розробці вебдодатку для управління майстернею по ремонту та обслуговуванню кавових апаратів за допомогою сучасних технологій зі забезпеченням зручного, інтуїтивно зрозумілого інтерфейсу користувача.

Практичне значення кваліфікаційної роботи полягає у наступному: спроектований та розроблений вебдодаток може використовуватися на практиці, а саме, для ведення обліку кавових апаратів та записів щодо обслуговування у спеціалізованій майстерні по їх ремонту та обслуговуванню.

Робота складається зі вступу, трьох розділів, висновків, переліку використаної літератури та додатків.

Перший розділ роботи присвячений огляду сучасних технологій, які використовувались при розробці вебдодатку.

Другий розділ описує практичні підходи, що були використані при проектуванні структури предметної області, бази даних, що використовується в додатку. Тут також описано етапи та підходи проектування та розробки вебдодатку.

Третій розділ присвячений опису інтерфейсу користувача та функціоналу вебдодатку. В цьому розділі також наведено інструкцію користувача.

РОЗДІЛ 1. ТЕОРЕТИЧНА ЧАСТИНА

1.1. Фреймворк Spring Boot

Spring Boot – це фреймворк на основі мови програмування Java з відкритим вихідним кодом, розроблений компанією Pivotal Software (зараз частина VMware). Основною ціллю якого є спрощення процесу створення та розгортання програмних продуктів з мінімізацією процесів налаштування.

Ключовими особливостями фреймворку Spring Boot є

- Впровадження залежностей: механізм впровадження залежностей у Spring Boot спрощує розробку шляхом інверсії керування через IoC контейнер. Це дозволяє фреймворку легко керувати залежностями компонентів, що, в свою чергу, сприяє модульності та гнучкому налаштуванню
- Вбудований сервер: Spring Boot фреймворк постачається з вбудованим HTTP-сервером, наприклад, Tomcat. Це дозволяє здійснювати збірку програмного продукту в jar-файл, що дозволяє запускати безпосередньо його, не потребуючи наявності сервера на машині;
- Автоконфігурація: Spring Boot володіє потужної інфраструктурою, що дозволяє значно зменшити необхідність мануальної конфігурації додатків, при цьому фреймворк буде опиратись на вказані розробниками бібліотеки;
- Автономність: продукти розроблені засобами фреймворку Spring Boot можна запускати як автономні програми, що не вимагають попереднього розгортання;
- Визначені параметри за замовчуванням: Spring Boot забезпечує інтерактивні параметри за замовчуванням для властивостей та налаштувань програмного продукту, що значно зменшує кількість налаштувань зі сторони програміста та пришвидшує розробку.

Загалом Spring Boot спрощує процес розробки, а також дає змогу розробникам створювати прості, надійні, масштабовані та придатні для обслуговування вебдодатки на основі мови програмування Java зі значно меншою кількістю шаблонного коду та налаштувань.

1.1.1. Основні модулі фреймворку.

Не зважаючи на те, що Spring Boot фреймворк базується на основних модулях та концепціях значно ширшого Spring, він не є явно розділеним на окремі модулі так, як традиційно організовано структуру Spring фреймворку. Оскільки Spring Boot був розроблений як повністю самостійний фреймворк для створення готових до розгортання програм і поєднує в собі різні бібліотеки та модулі для спрощення процесу розробки.

При цьому є ключові концепції та функції, які можна розглядати як модулі або ж компоненти:

- Spring Boot Starter: стартери – це фундаментальна концепція. Вони являють собою набір зручних дескрипторів залежностей, які можна включити у програму. Вони розроблені для забезпечення послідовного і гарантованого набору залежностей, що використовуються для певної мети;
- Автоматична конфігурація: це потужний інструмент фреймворку, який надає можливість автоматичного налаштування програм на основі вказаних залежностей, що допомагає значно зменшити кількість явно вказаних налаштувань:
- Spring Boot Actuator: інструмент, що надає готовий функціонал для контролю та управління програмним продуктом. Він надає кінцеві точки для збору корисної інформації про продукт, а саме інформацію про працездатність, показники та змінні середовища;
- Spring Boot CLI: інтерфейс командного рядка дозволяє запускати Groovy скрипти з конфігураціями Spring;

- DevTools Spring Boot: різноманітні інструменти, що допомагають полегшити розробку додатку та зробити її більш плавною;
- Spring Boot Web: веб-модуль, що містить набір функцій для розробки вебдодатків в тому числі включаючи підтримку вбудованого веб-сервера;
- Spring Boot Data: модуль, що спрощує доступ до даних та забезпечує підтримку різних баз даних;
- Spring Boot Security: модуль фреймворка, що надає функціонал для управління безпекою програм, в тому числі авторизацією та автентифікацією.

1.1.2. Spring Boot Starter Web

Spring Boot – це фреймворк, який відмінно підходить для розробки вебдодатків, оскільки він дає можливість створити повністю самостійний HTTP-сервер за допомогою вбудованих технологій, таких як Tomcat, Jetty, Undertow та інші. Більшість таких додатків використовують модуль spring-boot-starter-web для швидкого створення та запуску.

Для створення вебдодатків на основі сервлетів Spring Boot надає можливість скористатися системою автоналаштувань для Spring MVC або Jersey.

Фреймворк Spring Web MVC, або як його ще називають Spring MVC – це розширений веб-фреймворк типу «модель-представлення-контролер». Він дозволяє створювати спеціальні класи-компоненти позначені анотаціями @Controller або @RestController для обробки HTTP-запитів. Для позначення URL-адреси, якій відповідає той чи інший контролер використовується анотація @RequestMapping з параметром «name», через який і задається потрібне значення. Для позначення методів класу-контролера, які відповідатимуть на обробку самих HTTP-запитів використовують анотації @GetMapping, @PostMapping, @PutMapping, @PatchMapping та @DeleteMapping, які також володіють параметром «name» для додаткової конфігурації.

```java
@RestController
@RequestMapping("machine/characteristic_type")
public class CharacteristicTypeController {

    private final CharacteristicTypeService characteristicTypeService;

    @Autowired
    public CharacteristicTypeController(CharacteristicTypeService characteristicTypeService) {
        this.characteristicTypeService = characteristicTypeService;
    }

    @GetMapping
    public GenericResponse<List<CharacteristicType>> getAllCharacteristicTypes() {
        return GenericResponse.of(characteristicTypeService.getAll());
    }

    @PostMapping
    public GenericResponse<CharacteristicType> saveCharacteristicType(@RequestBody SaveCharacteristicTypeDto saveCharacteristicTypeDto) {
        return GenericResponse.of(this.characteristicTypeService.save(saveCharacteristicTypeDto));
    }

    @PutMapping
    public GenericResponse<CharacteristicType> updatePartTypeName(@RequestBody EditCharacteristicTypeDto editCharacteristicTypeDto) {
        try {
            return GenericResponse.of(characteristicTypeService.editCharacteristicType(editCharacteristicTypeDto));
        } catch (NoSuchElementException e) {
            return GenericResponse.error(String.format("Characteristic type with id=%d was not found", editCharacteristicTypeDto.getId()));
        }
    }

    @DeleteMapping("{characteristicTypeId}")
    public GenericResponse<Long> deleteCharacteristicType(@PathVariable Long characteristicTypeId) {
        try {
            characteristicTypeService.deleteCharacteristicTypeWithId(characteristicTypeId);
            return GenericResponse.withSuccessMessage(String.format("Characteristic type with id=%d was successfully deleted", characteristicTypeId));
        } catch (NoSuchElementException e) {
            return GenericResponse.error(String.format("Characteristic type with id=%d was not found", characteristicTypeId));
        }
    }
}
```

Рис. 1.1. Приклад класу-контролера

1.1.3. Spring Data JPA

Spring Data JPA – це модуль фреймворка, що дозволяє легко впроваджувати репозиторії для роботи з даними на основі Java Persistence API, що значно полегшує створення додатків на основі Spring та Spring Boot, які використовують технології доступу до баз даних.

Реалізація доступу до баз даних у додатках може досягати громіздких розмірів, оскільки навіть найпростіші операції з даними потребують великої кількості шаблонного коду. Spring Data JPA призначений значно спростити та покращити процес розробки рівнів доступу до баз даних. Таке покращення досягається за рахунок відмінно розвиненої інфраструктури модуля, що зводить процес створення рівнів роботи з базами даних до написання інтерфейсів з переліком необхідних методів. При цьому Spring Data JPA надає можливість використовувати шаблонні конструкції для опису методів, а також залишає можливість написання запитів до баз даних нативними мовами запитів, специфічними для конкретної бази даних.

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    Optional<User> findByUsernameOrEmail(String username, String email);

    default Optional<User> findByLogin(String login) { return this.findByUsernameOrEmail(login, login); }

    List<User> findAllByRolesContains(UserRole role);

    default List<User> findAllMasters() { return findAllByRolesContains(UserRole.MASTER); }

    Optional<User> findByUsername(String username);
}
```

Рис. 1.2. Приклад репозиторія

1.2. Генерація додатку

Перед початком роботи над програмним продуктом необхідно створити проект для нього. Це зробити дуже легко, оскільки для створення проекту Spring Boot передбачено дві зручні можливості: генерація проекту по шаблону засобами IntelliJ Idea та створення проекту засобами веб-ресурсу start.spring.io

1.2.1. Генерація проекту засобами start.spring.io

start.spring.io - це веб-сервіс розроблений командою Spring і призначений для генерації базової структури проекту для проектів Spring Boot. Даний інструмент слугує простим та зручним способом створення нового проекту з необхідними конфігураціями та залежностями.

Основні функції start.spring.io включають:

- Генерація проекту: сервіс надає можливість користувачам вказувати метадані проекту, наприклад назву проекту, опис, тип збірки, засіб автоматичного збирання та залежності, які необхідні для роботи проекту. Сервіс генерує архів проекту із заданою конфігурацією, який можна завантажити;
- Вибір залежностей: веб-інтерфейс надає користувачам можливість вибирати залежності з широкого асортименту, який включає в себе різні версії фреймворку Spring Boot, модулі для роботи з веб-технологіями, засоби роботи з базами даних, драйвера, механізми

шаблонів та багато іншого. Це допомагає легко пристосувати проект до конкретних потреб розробника;

- Збірка проекту: сервіс надає можливість користувачам вибрати спосіб збірки свого проекту, а саме як JAR (архів Java) або WAR (веб-архів);

Створений проект містить базову структуру каталогів, яка є притаманною для стандартних проектів Spring, основні властивості, конфігурації та просту програму «Hello World!». Все це слугує початковою точкою для розробників для створення і розширення своїх програмних продуктів написаних засобами фреймворку Spring Boot. За рахунок узагальненої стандартної структури, згенерований проект можна легко інтегрувати в різні середовища розробки, наприклад, IntelliJ IDEA, Eclipse, Visual Studio Code та інші.

Веб-ресурс start.spring.io дозволяє користувачам налаштовувати згенеровані проекти, вказуючи додаткові конфігурації та залежності за допомогою наданих параметрів. Таким чином, даний веб-ресурс є цінним інструментом для розробки програмних продуктів засобами фреймворка Spring Boot, оскільки він значно спрощує процес створення проектів з потрібними залежностями та конфігураціями, що дозволяє значно зекономити час і забезпечити чітку і надійну платформу для розширення та розвитку.

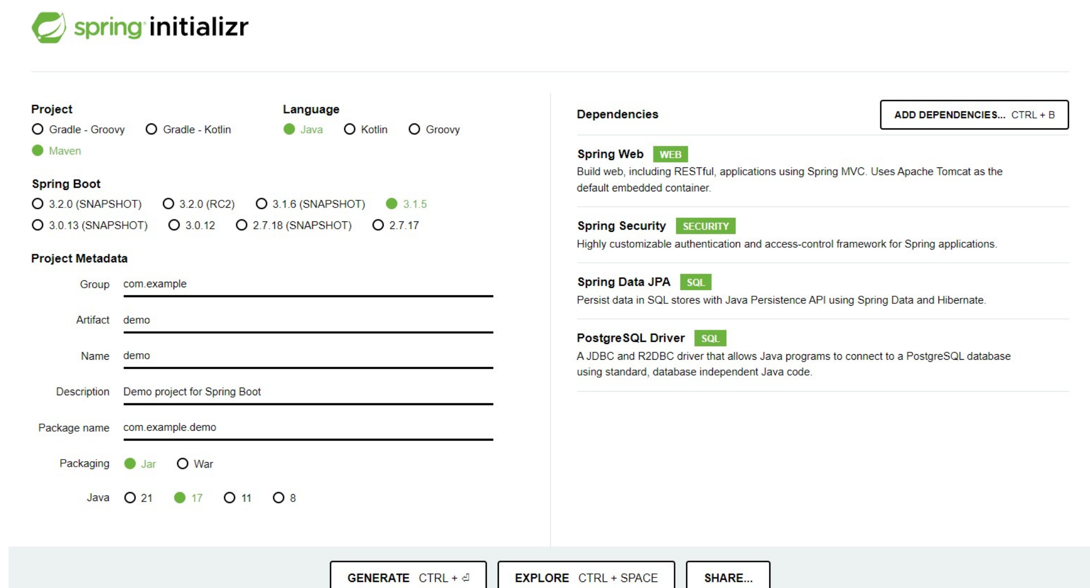


Рис. 1.3. Інтерфейс сервісу start.spring.io

1.2.2. Генерація проекту засобами середовища IntelliJ IDEA

Сучасне інтерактивне середовище розробки IntelliJ IDEA також надає можливість генерації шаблонного додатку на основі Spring Boot фреймворка.

Для здійснення генерації проекту за допомогою шаблона необхідно перейти в меню File => New => Project… та вибрати пункт Spring Initializr.

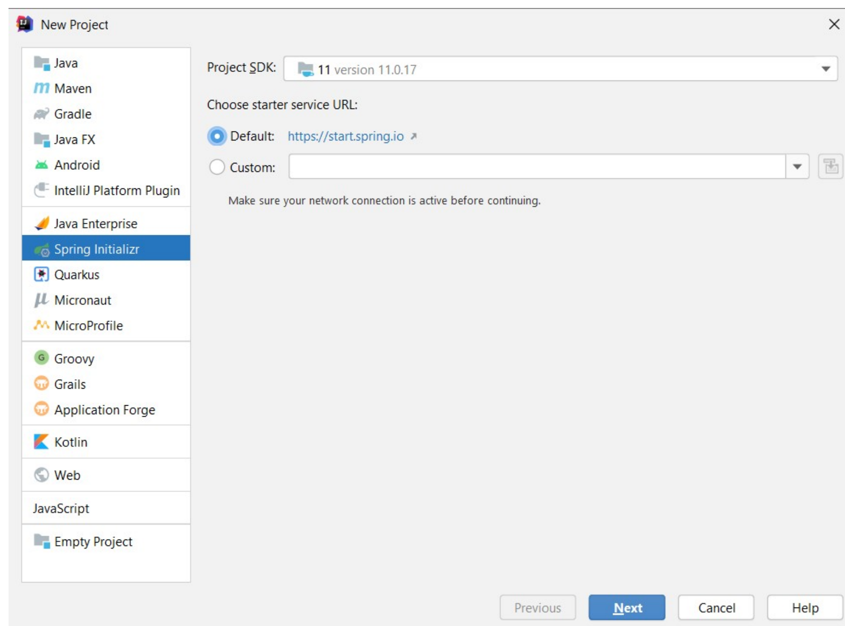


Рис. 1.4. Модальне вікно створення проекту

Таким чином середовище розробки пропонує два варіанти: завантажити готову шаблон з готовою конфігурацією або використати API вищеописаного сервісу start.spring.io.

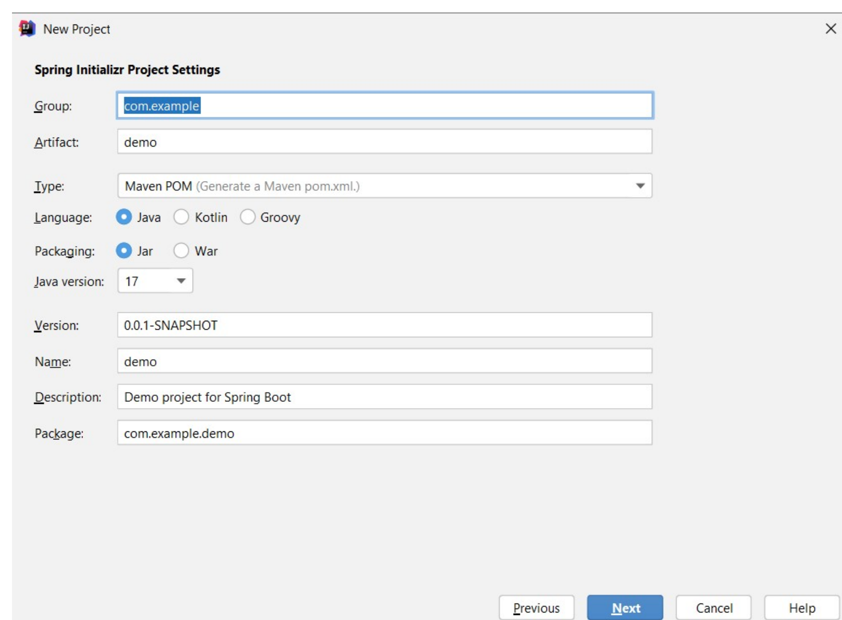


Рис. 1.4. Модальне вікно налаштування загальних параметрів проекту

Після того як користувач заповнить всі загальні дані про проект та підтвердить їх шляхом натискання на кнопку «Next», його буде перенаправлено на модальне вікно управління залежностями.

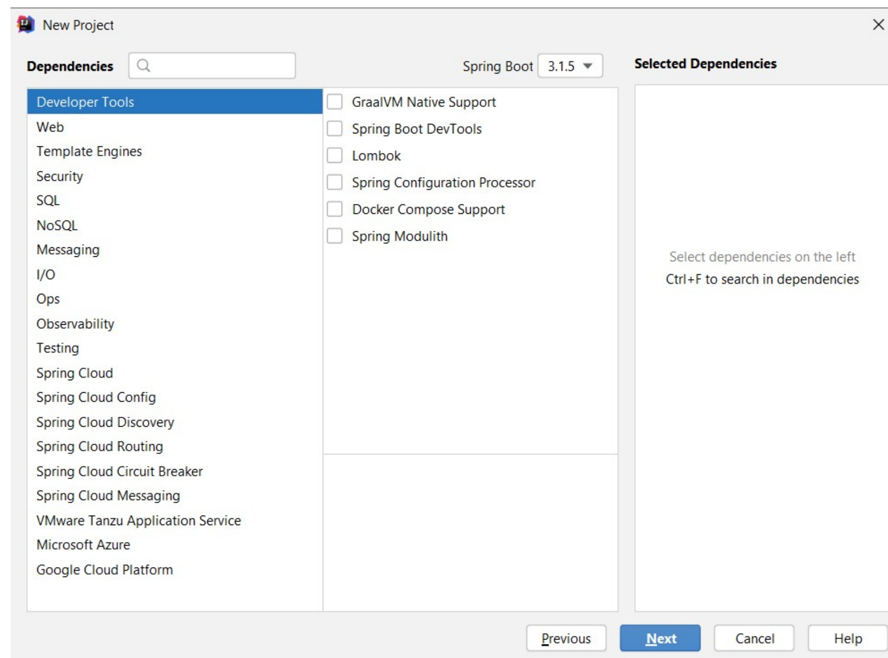


Рис. 1.4. Модальне вікно управління залежностями

Вибравши всі необхідні залежності, користувач здійснює перехід на наступний крок, де йому залишається тільки вказати назву для проекту та завершити генерацію.

1.3. СУБД PostgreSQL

Важливим аспектом розробки програмного продукту націленого на роботу з великою кількістю даних є використання реляційних баз даних. Вони служать основою для багатьох програм та забезпечуючи цілісність даних завдяки зв'язкам та обмеженням. Одною з таких СУБД є PostgreSQL, яку і було обрано для використання при розробці програмного продукту.

PostgreSQL — це надзвичайно потужна об'єктно-реляційна база даних з відкритим вихідним кодом, що використовує та розширює структуровану мову запитів SQL у поєднанні з багатьма функціями, котрі безпечно зберігають і масштабують найскладніші дані. PostgreSQL була розроблена в 1986 році як частина проекту POSTGRES у Каліфорнійському університеті в місті Берклі та активно розвивалась понад 35 років на основній платформі.

PostgreSQL володіє міцно репутацією, яку здобув завдяки своїй надійності, перевіреній архітектурі, цілісності даних, набору функцій, можливості розширюватись та відданості спільноти з відкритим кодом, котра стоїть за програмним забезпеченням та працює над тим, щоб постійно надавати користувачам ефективні та інноваційні рішення. PostgreSQL працює на всіх операційних системах та сумісний з ACID з 2001 року. Також PostgreSQL має потужні додаткові модулі як, наприклад, популярний розширювач геопросторових баз даних PostGIS. Все це сприяло вибору PostgreSQL як реляційної бази даних з відкритим вихідним кодом для багатьма проектами та організаціями.

PostgreSQL має велику кількість функцій, які допомагають розробникам створювати програмні продукти, адміністраторам — зберігати цілісність даних і створювати надійне та відмовостійке середовище, а також допомагають керувати даними, незалежно від обсягу набору даних. PostgreSQL є безкоштовним програмним забезпеченням з відкритим вихідним кодом та володіє надзвичайною здатністю до розширення, наприклад, дає можливість визначати власні типи даних, створювати власні функції та навіть писати код різними мовами програмування без перекомпіляції бази даних.

PostgreSQL намагається відповідати стандарту SQL та підтримує багато функцій, необхідних стандарту, хоч іноді з дещо відмінним синтаксисом або функціями. Починаючи з випуску версії 16 у вересні 2023 року, PostgreSQL реалізовує принаймні 170 із 179 обов’язкових функцій для відповідності стандарту SQL:2023 Core.

1.4. Фреймворк Angular

TypeScript — це безкоштовна високорівнева мова програмування з відкритим вихідним кодом, розроблена компанією Microsoft, яка додає до JavaScript статичну типізацію з додатковими анотаціями типу. Він створений для розробки великих додатків і компілюється в JavaScript. Оскільки TypeScript є розширенням JavaScript, то усі програми написані на JavaScript є синтаксично

дійсними і для TypeScript, але вони можуть не перевіряти тип даних з міркувань безпеки.

TypeScript можна використовувати для розробки програмних продуктів як для виконання на стороні клієнта, так і серверні додатки (як у випадку з Node.js або Deno). Для компіляції в JavaScript код доступні кілька варіантів. Для цього можна використовувати типовий компілятор TypeScript або викликати компілятор Babel для перетворення TypeScript коду в JavaScript.

TypeScript підтримує файли визначення, котрі можуть містити інформацію про типи існуючих бібліотек JavaScript. Це дозволяє іншим програмним продуктам використовувати значення, які визначені у файлах, так само, як якщо б вони були статично типізованими сутностями TypeScript.

Таким чином TypeScript став невід'ємною частиною сучасної веб-розробки, інтегруючись з популярними фреймворками для спрощення процесу розробки програмних продуктів. У таких фреймворках, як Angular чи React, TypeScript надає переваги статичної типізації, що забезпечує раннє виявлення помилок і покращення якості коду.

Одним з таких фреймворків є Angular, використання якого разом з TypeScript значно полегшує розробку, надаючи потужні інструменти для написання масштабованих, надійних та ефективних програмних продуктів. Все це сприяло вибору фреймворку Angular для написання фронтенд частини програмного продукту.

Angular – це потужний фреймворк розроблений компанією Google, який призначений для написання браузерних та мобільних додатків. Однією з ключових особливостей Angular є те, що він використовує я якості мови програмування TypeScript, що дозволяє значно покращити читабельність програмного коду і допускати меншу кількість помилок підчас розробки.

Даний фреймворк володіє великою кількістю переваг:

- MVC архітектура: використання архітектури «модель-представлення-контролер» додає цінності фреймворку та закладає базу для інших

функцій, як от область застосування та прив'язка даних. Використання цієї архітектури дозволяє відділити логіку додатку від UI частини;

- Розширена архітектура дизайну: багато сучасних вебдодатків складаються з великої кількості компонентів і з ростом проекту стає важко ними керувати. Архітектура фреймворку дозволяє значно спростити підтримку проекту;
- Модульність: Angular використовує підхід модульного програмування, згідно з яким програмний продукт складається з декількох модулів. Таким чином кожний модуль певну конкретну роль, що забезпечує кращий розподіл задач;
- Прив'язка даних: це означає, що дані можуть відображатись в різних представленнях, проте змінивши їх в одному з представлень, то зміни відобразяться і в решті місць без написання додаткового коду. Angular забезпечує двосторонню прив'язку даних. Це означає, що і представлення і модель оновлюються, коли здійснюються зміни в моделі та представленні відповідно;
- Сервіси та впровадження залежностей: сервіси – це функції JavaScript, які відповідають за виконання тільки певних конкретних задач. Вони представляють собою класи, що реалізовують шаблон проектування «Одинак». Сервіси поєднуються між собою за допомогою механізму впровадження залежностей;
- Директиви – Angular дозволяє розширити HTML з допомогою нових атрибутів, які і називаються директивами. Директива ng-app застосовується для позначення програми Angular, ng-model - прив'язує значення HTML елементів (input, select, text) до даних програми, а ng-bind прив'язує дані програми до подання HTML.

РОЗДІЛ 2. ПРАКТИЧНА ЧАСТИНА

2.1. Проектування програмного продуту

2.1.1. Історії користувачів

Важливим моментом перед початком розробки програмного продукту є повноцінне формулювання бачення того, якими ролями можуть володіти користувачі та який функціонал повинен бути доступний для них. Відмінним інструментом для цього є історії користувачів (user stories).

Історії користувачів представляють собою короткий опис функціональної можливості програмного забезпечення, що пишеться простими загальними словами і відображає бачення продукту з точки зору кінцевого користувача. Вони визначаються з ціллю описати те, як саме виконання певної конкретної задачі призведе до деякої конкретної цінності для клієнта. При цьому, «Клієнтами» не обов'язково мають бути кінцеві клієнти, їх роль можуть виконувати також і внутрішні клієнти. Історії користувачів є частиною гнучких методологій, таких як Scrum і Kanban, де вони використовуються для визначення можливостей продукту та встановлення їх пріоритизація. Вони допомагають командам розробників зрозуміти функціонал додатку з точки зору користувача та визначати пріоритети на основі потреб і цінності користувача.

Як правило, історії користувачів складаються з декількох частин, які просто і в загальних рисах описують певні дії для отримання необхідного результату. При цьому вони повинні бути чіткими, лаконічними та не повинні містити в собі дрібних деталей. Історії користувачів часто записуються у вигляді простого речення наступного виду:

«Як [тип користувача], [хочу щось], [щоб робити щось]».

Отже, здійснивши попередній аналіз вимог до проекту, було написано такі історії користувачів:

- як незареєстрований користувач, я хочу мати можливість створити обліковий запис, щоб мати можливіть ввійти в нього;
- як неавторизований користувач, я хочу мати можливість здійснити вхід в особистий обліковий запис, щоб розпізнаватись системою і мати доступ до персоналізованого вмісту;
- як користувач з правами клієнта, я хочу мати можливість переглядати свої кавові апарати, щоб мати доступ до інформації про них;
- як користувач з правами клієнта, я хочу мати можливість додавати придбані кавові апарати, щоб вони відображались серед моїх апаратів;
- як користувач з правами майстра, я хочу мати можливість переглядати список кавових апаратів, щоб бачити інформації про них;
- як користувач з правами майстра, я хочу мати можливість створювати шаблони апаратів, щоб полегшити процедуру додавання нових апаратів;
- як користувач з правами майстра, я хочу мати можливість додавати нові кавові апарати, щоб додати їх в реєстр майстерні;
- як користувач з правами майстра, я хочу мати можливість додавати записи стосовно ремонту та огляду, щоб вести журнал обслуговування;
- як користувач з правами адміністратора, я хочу мати можливість видаляти кавові апарати, щоб позбуватися неактуальної інформації;
- як користувач з правами адміністратора, я хочу мати можливість блокувати користувачів, щоб відключити порушників правил сайту;
- як користувач з правами адміністратора, я хочу мати можливість керувати параметрами апаратів, щоб забезпечувати актуальність даних для майстрів.

Після написання історій користувачів було вирішено візуалізувати відношення ролей користувачів та функціоналу, що їх відповідає. Для цього відмінно підходить UML-діаграма прецедентів.

Діаграма прецедентів – це UML-діаграма, яка відображає відношення між учасниками та прецедентами. Учасниками в діаграмі прецедентів можуть бути люди, їх роль в системі або навіть інша система, що представляє щось поза прецедентом. Прецедентом називається опис окремого аспекту поведінки системи. При цьому, прецедент не повинен описувати реалізацію певного функціоналу, а тільки виконуваний процес.

Рис. 2.1. Діаграма прецедентів, побудована засобами Diagrams.net

2.1.2. Концептуальна модель «сутність-зв’язок»

Після того, як було отримано загальне бачення предметної області, визначення необхідних типів користувачів, та необхідної для програмного продукту функціональності, вдалося сформувати предметну область яка б відповідала суті продукту. Чітке бачення цього дозволило розпочати роботу над проектуванням моделі даних.

Насамперед було визначено необхідні сутності, такі як користувач, кавовий апарат, бренд, модель, завдання, деталь, та інші. Після цього було визначено необхідні поля для цих сутностей. Наступним кроком було знаходження таких полів, значення яких дублюються і їх можна було б винести в окремі сутності. Цей процес надзвичайно схожий до нормалізації баз даних, оскільки він забезпечує мінімізацію дублювання даних, структурну цілісність та ефективність операцій при роботі з даними, що пов'язано з їх кращою організацією. Таким чином цей процес виконувався до тих пір, поки сутності не почали відповідати нормальній формі Бойса-Кодда, а запис інформації про ці сутності було записано у вигляді схеми «сутність-зв'язок».

Схема «сутність-зв'язок» (Entity–relationship diagram або ER-діаграма) – це різновид блок-схем, де відображаються різні сутності (моделі, класи, об'єкти) пов'язані між собою та які типи зв'язків вони мають. ER-діаграми часто використовуються для проектування реляційних баз даних.

Для візуалізації результату виконання цього процесу було використано веб-сервіс для роботи з діаграмами, а саме Diagrams.net.

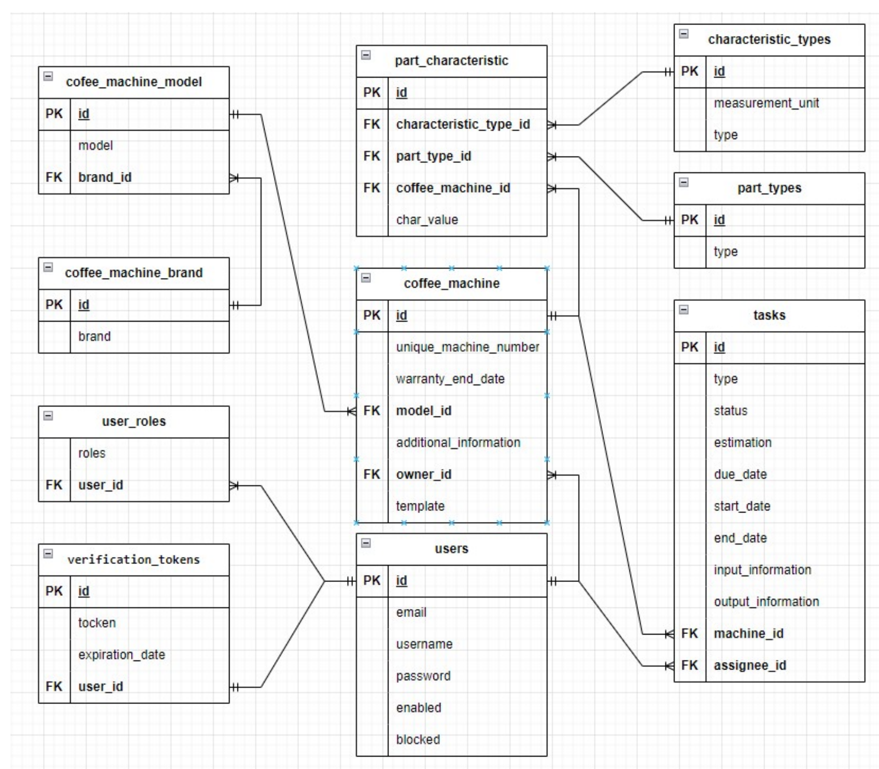


Рис. 2.2. Візуалізація ER-діаграми засобами diagrams.net

2.2. Розробка програмного продуту

2.2.1. Класи моделей

Одразу після того, як був завершений процес проектування, стало зрозуміло, які класи-моделей необхідно створити та якими властивостями вони повинні володіти. Загалом було створено дев'ять класів-моделей: User, VerificationToken, Task, PartType, PartCharacteristic, CharacteristicType, CoffeeMachine, CoffeeMachineBrand, CoffeeMachineModel. Всі вищезгадані моделі володіють властивостями, які є відповідними до моделі «сутність-зв'язок», за винятками моментів агрегації, оскільки в таких випадках потрібно описувати поля згідно з вимогами Spring Data JPA.

```
7   @Entity
8   @Table(name = "coffee_machines")
9   public class CoffeeMachine {
10
11      @Id
12      @GeneratedValue(strategy = GenerationType.IDENTITY)
13      private Long id;
14
15      @Column(name = "template")
16      private boolean template;
17
18      @Column(name = "unique_machine_number")
19      private String uniqueMachineNumber;
20
21      @ManyToOne(cascade = CascadeType.ALL)
22      @JoinColumn(name = "model_id")
23      private CoffeeMachineModel model;
24
25      @Column(name = "warranty_end_date")
26      private LocalDate warrantyEndDate;
27
28      @Column(name = "additional_information")
29      private String additionalInformation;
30
31      @OneToMany(mappedBy = "coffeeMachine")
32      private List<PartCharacteristic> parts;
33
34      @ManyToOne(cascade = CascadeType.ALL)
35      @JoinColumn(name = "owner_id")
36      private User owner;
```

Рис. 2.3. Приклад класу CoffeeMachine з урахуванням вимог Spring Boot JPA

Згідно зі стандартом Spring Data JPA, до програмного коду встановлюються певні вимоги, а саме – необхідність вказати певні анотації щодо конфігурації.

Таким чином для позначення того, що цей клас є моделлю слід використати анотацію @Entity. Ця анотація встановлює чіткі обмеження на промаркований клас, а саме:

- клас повинен мати хоча б одне поле;
- хоча б одне поле класу повинно бути первинним ключем;
- клас повинен містити конструктор без параметрів;
- клас не повинен бути визначений як **final**.

Анотація @Entity також має параметр name, який не є обов'язковим. Цей параметр відповідає за встановлення назви моделі. Оскільки даний атрибут є необов'язковим, то у випадку коли він не буде вказаний модель буде мати таку ж саму назву, як і промаркований клас.

Для позначення того, що модель відповідає таблиці бази даних використовується анотація @Table. Ця анотація також має атрибут name, який також не є обов'язковим. Призначення цього атрибуту є аналогічним до однойменного з вищеописаної анотації @Entity, а саме встановлення назви для таблиці в базі даних. Таким чином, якщо позначити клас анотацією @Table, то буде вважатися, що він має відповідну таблицю в БД з назвою ідентичною до назви самого класу, а при явно прописаному атрибуті nameназва таблиці буде повністю ідентичною до значення цього атрибута.

Наступною по значимості анотацією є @Id, яка слугує для позначення полів класу, котрі будуть відповідати первинним ключам. Ця анотація встановлює на поле класу повністю ідентичні вимоги, що і база даних:

- поле має бути обов'язково заповнене;
- поле повинно володіти унікальними значеннями;

У випадках, коли необхідно, щоб значення первинного ключа було згенеровано автоматично можна застосувати анотацію @GeneratedValue. Ця анотація відповідає за стратегію генерації значення для первинного ключа і має

обов’язковий параметр **strategy**. Це параметр перелічуваного типу GenerationType і може набувати наступних значень:

- AUTO – автоматичний вибір одного з підходів, котрий найкраще підходить до типу ключа;
- TABLE – підхід, що використовує основні таблиці бази даних, що містять частини згенерованих унікальних значень;
- SEQUENCE – підхід, що базується на використання послідовностей для генерації первинного ключа та використовується з базами даних, які підтримують такий підхід, а в іншому випадку використовується TABLE підхід;
- IDENTITY – підхід до генерації ключа, що є реалізацією автоматичного інкременту.

Для позначення того що поле повинно відповідати колонці з таблиці використовується анотація @Column. Насправді ця анотація є необов’язковою та використовується для позначення зв’язку поля класу з колонкою таблиці. Проте ця анотація також має необов’язкове поле name, за допомогою якого можна встановити конкретну назву для колонки. За допомогою цієї анотації також можна визначати різні обмеження для колонки, такі як максимальна довжина, унікальність значень та інші.

При використанні в класі поля перелічуваного типу прийнято використовувати анотація @Enumerated. Єдиний її параметр **value** дозволяє встановити тип даних, що використовуватиметься в базі даних для позначеної колонки. В якості значення цього параметра також використовується перелічуваний тип, що має назву EnumType та може набувати двох значень:

- ORDINAL – використання індексу значення в перелічуваному типу;
- STRING – використання стрічкового значення за допомогою функції name().

Не менш важливими анотаціями є ті, що відповідають за налаштування зв’язків між об’єктами. До них належать анотації @OneToOne, @OneToMany,

@ManyToMany, а також ті, що використовуються з ними для забезпечення цілісності даних.

Анотація @OneToOne використовується для позначення зв’язку між двома сутностями, при якому одному екземпляру першої сутності відповідає один і тільки один екземпляр другої сутності. Дана анотація була використана для прив’язки токена верифікації до користувача. Даний вид зв’язку гарантує, що в базі даних буде тільки один токен для конкретного користувача, що унеможливить існування ситуацій при яких можливе виникнення колізій. Ця анотація використовується разом з @JoinColumn, яка слугує для того, щоб вказати яким чином зв’язувати колонки таблиць. Вона володіє наступними параметрами:

- name – використовується для задання назви колонки для поточної сутності;
- referencedColumnName – використовується для позначення колонки сутності зв’язаної з поточною, з якої повинно братись значення.

Окрім зазначених вище полів анотація @OneToOne володіє і іншими параметрами, які є притаманними для всіх анотацій, що позначають колонку в базі даних (unique, nullable, insertable, updatable та інші).

```
@OneToOne
@JoinColumn(name = "user_id", referencedColumnName = "id")
private User user;
```

Рис. 2.4. Приклад використання анотацій @OneToOne та @JoinColumn

Анотація @ManyToOne використовується для позначення типу зв’язку, при якому безліч об’єктів поточної сутності відповідати одному об’єкту іншої сутності, але не навпаки. Вона, як і @OneToOne використовується в парі з анотацією @JoinColumn.

```
@ManyToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "owner_id")
private User owner;
```

Рис. 2.5. Приклад використання анотації @ManyToOne

Для забезпечення двонаправленості зв'язків між класами двох сутностей можна використовувати в парі з @ManyToOne анотацію @OneToMany. Вона дозволяє зберігати в об'єкті однієї сутності колекцію об'єктів іншої, що посилаються на неї. Це дозволяє полегшити роботу з агрегованими об'єктами, котрі повинні опрацьовуватись логікою «класу-контейнера». При цьому, якщо для виконання певних операцій не потрібно завантажувати в оперативну пам'ять об'єкти цієї колекції, то цей зв'язок можна налаштувати належним чином. Для цього достатньо задати необхідне значення для одного з параметрів, а саме, **fetch**. Цей параметр приймає значення перелічуваного типу FetchType та може набувати наступних значень:

- EAGER (значення параметра по замовчуванню) – використовується для позначення зв'язків між сутностями, коли завантаження агрегованих об'єктів в оперативну пам'ять є обов'язковим;
- LAZY – використовується для позначення зв'язків між сутностями, коли завантаження даних про агреговані об'єкти не завжди необхідне. У випадках, коли ці значення все ж будуть необхідні їх можна буде завантажити мануально за допомогою статичного методу Hibernate.initialize(…).

```
@OneToMany(mappedBy = "owner", fetch = FetchType.LAZY)
private Set<CoffeeMachine> coffeeMachines;
```

Рис. 2.6. Приклад використання анотації @OneToMany

2.2.2. Рівень доступу до баз даних.

Для роботи з даними було використано модуль Spring Data JPA, що значно полегшує роботу з реляційними базами даних та написання запитів. В основному це досягається за рахунок високорівневої абстракції. Spring Data JPA дозволяє автоматично генерувати запити до бази даних опираючись на назву методів та тип даних, що повинні ними повертатись. Це досягається за рахунок використання JPQL.

JPQL (Java Persistence Query Language) – це запитова мова для роботи з об’єктно-реляційними базами даних в контексті JPA. Вона надає можливість виразити запити до бази даних, використовуючи об’єктну модель замість табличної. Це дозволяє звертатись до об’єктів таблиці не за назвами колонок, а за назвами полів та властивостей сутностей без прив’язки до конкретної системи управління базами даних.

Таким чином для реалізації доступу до бази даних достатньо створити інтерфейс для певної сутності успадкувавши його від загального JpaRepository, вказавши тип об’єкта-сутності та тип його первинного ключа, а також позначити цей інтерфейс анотацією @Repository.

Інтерфейс JpaRepository вже надає декілька загальних методів для роботи з даними, оскільки він розширює інтерфейс CrudRepository. Це дозволяє не витрачати час на написання рутинного шаблонного коду та зосередитися на написанні логіки додатку або ж специфічних запитів.

2.2.3. Сервісний рівень.

Сервісний рівень – це рівень, на якому повинна бути реалізована вся необхідна логіка додатку. Він є посередником між контролерами та репозиторіями. На цьому рівні визначаються інтерфейси, що описують функціонал програмного продукту та його поведінку, а таж створюється їх реалізація.

Розбиття на інтерфейси та їх реалізації прийнято здійснювати з метою забезпечення гнучкості програмного коду, оскільки програми на основі Spring та Spring Boot фреймворків активно використовують принцип інверсії залежностей. Це дозволяє при необхідності легко замінити одну реалізацію на іншу без додаткової зміни програмного коду.

```java
public interface CharacteristicTypeService {
    List<CharacteristicType> getAll();

    CharacteristicType getById(Long id) throws NoSuchElementException;

    CharacteristicType save(SaveCharacteristicTypeDto characteristicType);

    CharacteristicType editCharacteristicType(EditCharacteristicTypeDto editCharacteristicTypeDto)
            throws NoSuchElementException;

    void deleteCharacteristicTypeWithId(Long characteristicTypeId) throws NoSuchElementException;

    CharacteristicType getCharacteristicTypeByNameAndMeasurementUnit(String characteristicType, String measurementUnit)
            throws ElementNotFoundException;
}
```

Рис. 2.7. Приклад інтерфейсу, що описує сервіс.

Після опису інтерфейсів, що визначають необхідні сервіси було створено їх реалізації. Реалізаціями сервісів є звичайні класи, що імплементують відповідні інтерфейси. При цьому реалізації необхідно позначати анотацією @Service. Це необхідно робити для того, щоб фреймворк зміг відрізнити їх від інших класів та створити відповідні об’єкти при запуску програми. Також таке маркування відіграє важливу роль в роботі механізму впровадження залежностей.

Однією з основних переваг сервісів для написання логіки додатку є підтримка транзакційності. Для цього Spring та Spring Boot фреймворки надають вже готовий функціонал. Все що потрібно зробити – це просто додати анотацію @Transactional. При цьому, якщо позначити цією анотацією конкретний метод, то транзакційним буде тільки він, а якщо позначити нею клас, то транзакційними будуть всі методи, що в ньому знаходяться.

2.2.4. Рівень контролерів.

Цей рівень містить класи, котрі відповідають за спілкування з клієнтською частиною та саме він визначає інтерфейс додатку, та функціонал доступний користувачам. Для позначення класів-контролерів у Spring Boot фреймворку використовуються дві анотації: @Controller та @RestController.

Анотація @Controller використовується при написанні програмних продуктів, що містять як бекенд, так і фронтенд частини. Методи таких класів повинні повертати назву відображення, що відповідатиме конкретному запиту від користувача. Такий підхід зручний для невеликих додатків, що орієнтовані тільки під роботу з браузерами.

Альтернативним підходом є використання анотації @RestController. Ця анотація є аналогічною до @Controller, проте використовується для позначення класів-контролерів, що працюють за REST підходом. Це дозволяє розробляти додатки з певним прикладним програмним інтерфейсом не обмежуючись якоюсь однією реалізацією фронтенд частини. Це, в свою чергу, надає можливість реалізувати зв'язок з користувачами різними способами, як от веб-клієнт, мобільний додаток, комп'ютерна програма з віконним інтерфейсом, тощо. Даний підхід також дозволяє іншим клієнт-серверним додаткам надсилати запити до даного програмного продукту та використовувати його функціонал.

При створенні даного програмного продукту було реалізовано сім класів контролерів, що відповідали за певні частини додатку, а саме:

- CharacteristicTypeController – робота з характеристиками та одиницям їх вимірювання;
- CoffeeMachineBrandController – робота з брендами кавових апаратів;
- CoffeeMachineController – робота з кавовими апаратами;
- CoffeeMachineModelController – робота з моделями кавових апаратів;
- PartTypeController – робота з деталями до кавових апаратів;
- SecurityController – робота з реєстрацією, авторизацією та автентифікацією користувачів;
- TaskController – робота з операціями над кавовими апаратами.

Кожен з цих класів прив'язаний до певної url-адреси та володіє специфічними методами, що необхідні для роботи з конкретними сутностями.

Таким чином клас CharacteristicTypeController володіє наступними методами:

- getAllCharacteristicTypes() – отримання всіх записів, використовується GET метод та адреса /machine/characteristic_type;

- saveCharacteristicType(saveCharacteristicTypeDto) – додавання нового запису, використовується POST метод та адреса /machine/characteristic_type;
- updateCharacteristic(editCharacteristicTypeDto) – оновлення існуючого запису, використовується PUT метод та адреса /machine/characteristic_type;
- deleteCharacteristicType(characteristicTypeId) – видалення існуючого запису за його ідентифікатором, використовується DELETE метод та адреса /machine/characteristic_type.

Для класу CoffeeMachineBrandController було визначено наступні методи:

- getAllCoffeeMachineBrands() – отримання всіх записів брендів, використовується GET метод та адреса /machine/brand;
- saveBrand(brandDto) – збереження нового запису, використовується POST метод та адреса /machine/brand;
- updateBrandName(editBrandDto) – оновлення існуючого запису, використовується PUT метод та адреса /machine/brand;
- deleteBrand(brandId) – видалення запису за ідентифікатором, використовується DELETE метод та адреса /machine/brand.

CoffeeMachineController клас реалізовує наступні методи:

- getAllMachines() – отримання всіх записів кавових апаратів, використовується GET метод та адреса /machines;
- getCoffeeMachineById(machineId) – отримання повної інформації про кавовий апарат за його ідентифікатором, використовується GET метод та адреса /machines/{machineId};
- isMachineWithUniqueNumberExist(uniqueMachineNumber) – перевірка наявності апарату з вказаним ідентифікатором, використовується GET метод та адреса /machines/exist/{uniqMachineNumber};
- getAllTemplates() – отримання всіх шаблонів, використовується GET метод та адреса /machines/template;

- getTemplateById(templateId) – отримання повної інформації про конкретний шаблон за його ідентифікатором, використовується GET метод та адреса /machines/template/{templateId};
- getAllMachinesForLoggedUser() – отримання записів про кавові апарати авторизованого користувача, використовується GET метод та адреа /machines/my;
- save(saveMachineDto) – додавання нового кавового апарата, використовується POST метод та адреса /machines;
- saveTemplate(saveMachineDto) – додавання нового шаблону кавового апарата, використовується POST метод та адреса /machines/template;
- addMachine(uniqNumber) – додавання апарату до списку придбаних користувачем за його ідентифікатором, використовується POST метод та адреса /machines/add.

Для забезпечення можливості роботи з моделями апаратів було реалізовано такі методи:

- getAllCoffeeMachineModels() – отримання всіх записів моделей, використовується метод GET та адреса /machine/models;
- getAllCoffeeMachineModelsForBrandWithId(brandId) – отримання всіх моделей, що належать до бренду з вказаним ідентифікатором, використовується GET метод та адреса /machine/models/for_brand/{brandId};
- saveModel(modelDto) – додавання запису, використовується метод POST та адреса /machine/models;
- updateModel(editModelDto) – оновлення існуючого запису, використовується PUT метод та адреса /machine/models;
- deleteModel(modelId) – видалення запису за його ідентифікатором, використовується DELETE метод та адреса /machine/models.

Щоб забезпечити програмний продукт можливістю роботи з запчастинами було розроблено клас PartTypeController та наділено його наступними методами:

- getAllPartTypes() – отримання всіх записів пов'язаних з запчастинами, використовується GET метод та адреса /machine/part_type;
- savePartType(savePartTypeDto) – додавання нового запису, використовується метод POST та адреса /machine/part_type;
- updatePartTypeName(editPartTypeDto) – редагування існуючого запису, використовується PUT метод та адреса /machine/part_type;
- deletePartType(partTypeId) – видалення запису за його ідентифікатором, використовується DELETE метод та адреса /machine/part_type/{partTypeId}.

Забезпечення програмного продукту можливостями реєстрації, авторизації та інших операцій напрямленим на верифікацію користувача було створено SecurityController, що володіє такими методами:

- login(userLoginDto) – вхід користувача в систему, використовується POST метод та адреса /security/login;
- logout(session) – вихід користувача з системи, використовується POST метод та адреса /security/logout;
- registration(userRegistrationDto) – реєстрація користувача в системі, використовується POST метод та адреса /security/registration;
- tokenVerification(token) – підтвердження адреси електронної пошти користувача з використанням унікального ключа надісланого на неї, використовується метод GET та адреса /security/token/verification;
- updateToken(token) – оновлення унікального ключа для підтвердження адреси електронної пошти користувача на випадок протермінування дій, використовується GET метод та адреса /security/token/update.

Ведення записів по обслуговуванню апаратів здійснюється за допомогою методів класу TaskController, а саме:

- getTaskTypes() – отримання переліку типів операцій, використовується GET метод та адреса /task/types;
- getTaskStatuses() – отримання переліку статусів операцій, використовується GET метод та адреса /task/statuses;
- getTaskMasters() – отримання переліку майстрів, використовується GET метод та адреса /task/masters;
- getTaskById(taskId) – отримання всієї інформації про операцію за її ідентифікатором, використовується GET метод та адреса /task/{taskId};
- getAllTasksForMachineWithId(machineId) – отримання переліку операцій для кавового апарату з вказаним ідентифікатором, використовується GET метод та адреса /task/for_machine/{machineId};
- saveRecord(saveDto) – додавання нового запису щодо обслуговування, використовується POST метод та адреса /task;
- editRecord(editDto) – редагування записів щодо обслуговування, використовується PUT метод та адреса /task.

2.3. Використані шаблони проектування

Підчас реалізації програмного продукту, так чи інакше, було використано ряд шаблонів проектування, що дозволило покращити якість коду, його гнучкість та читабельність. Це сприяло формуванню структури коду, що легко піддається масштабуванню. Нижче наведено опис декількох з таких шаблонів.

2.3.1. Одинак

Одним із основних шаблонів проектування є Одинак. Цей шаблон використовується за замовчуванням фреймворком Spring Boot для інфраструктурних класів таких як репозиторії, сервіси, контролери та інші. Його основна ідея полягає в тому, щоб підчас роботи програми було створено тільки один екземпляр класу. Це забезпечується наявністю тільки приватного конструктору та статичного методу, що повертає цей єдиний екземпляр.

2.3.2. Замісник

Шаблон проектування «Замісник» є структурним шаблоном, що надає об'єкт-замінник. Цей об'єкт контролює доступ до оригінального об'єкту і, при цьому, дозволяє виконувати певні дії перед або ж після виклику оригінальних методів. Його основна ідея полягає в створенні нового об'єкту, що буде замінником оригінального, є схожим на нього та реалізовує той самий інтерфейс. Саме це дозволяє виконувати додаткові операції перед або після логіки методів оригінального об'єкту. Цей шаблон широко використовується фреймворком Spring Boot для імплементації інфраструктурної логіки, такої як аспектно-орієнтоване програмування, транзакційний менеджмент та інші.

2.3.3. Стратегія

Стратегія є поведінковим шаблоном проектування. Його основна ідея полягає в винесенні подібної логіки в окремі класи, що реалізовують спільний інтерфейс замість безпосередньої інтеграції цієї логіки в один клас. Такий підхід робить ці алгоритми взаємозамінними та використовувати різні підходи без зміни програмного коду.

Використання цього шаблону проектування значно покращує читабельність коду, робить його гнучкішим та більш лінійним. Яскравим прикладом використання стратегії є механізм відправки транзакційних електронних листів. Таким чином визначається загальний інтерфейс, що описує всі необхідні методи для формування електронного листа. Після цього для кожного з можливих сценаріїв створюється окремий клас, що відповідає цьому інтерфейсу та реалізовує описані методи. Це дозволяє позбутись розгалужень в коді та лінійно викликати методи конкретної реалізації.

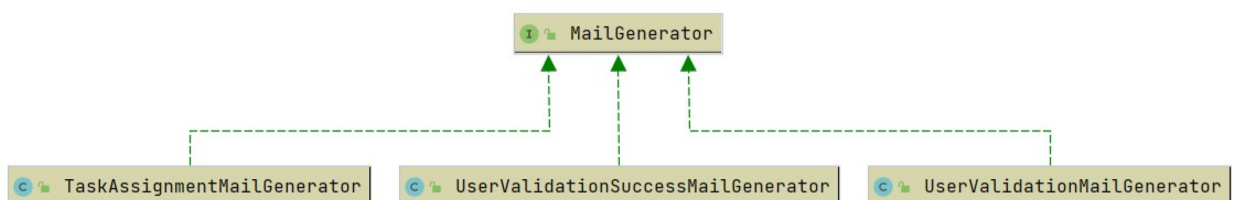


Рис. 2.8. Схема-ілюстрація шаблону «Стратегія»

В поєднанні з механізмом впровадження залежностей, що надається фреймворком Spring Boot, всі імплементації стратегії можна помістити в колекцію Map, де ключем буде виступати певний ідентифікатор, а значенням – сама реалізація.

```java
private final Environment environment;
private final Map<EmailType, MailGenerator> mailGenerators;

@Autowired
EmailServiceImpl(Environment environment, List<MailGenerator> mailGenerators) {
    this.environment = environment;
    this.mailGenerators = mailGenerators.stream()
            .collect(Collectors.toMap(
                    MailGenerator::getEmailType,
                    value -> value));
}
```

Рис. 2.9. Приклад впровадження колекції стратегій

Завдяки такому впровадженню залежностей можна використати конкретну реалізацію використовуючи ключ-ідентифікатор.

```java
public void sendEmail(EmailType type, String emailTo, Map<String, Object> parameters) {
    MailGenerator mailGenerator = mailGenerators.get(type);
    if (mailGenerator == null) {
        throw new UnsupportedOperationException(
                String.format("Emails with type <%s> are not supported", type.name()));
    }
    EmailInstance emailInstance = mailGenerator.generate(parameters);
```

Рис. 2.10. Приклад використання шаблону «Стратегія»

РОЗДІЛ 3. ІНСТРУКЦІЯ КОРИСТУВАЧА ТА ОПИС ДОДАТКУ

3.1. Реєстрація нових користувачів

Оскільки розроблений продукт орієнтований на роботу з багатьма користувачами та їх даними, то необхідно було забезпечити можливість реєстрації нових користувачів. Для цього було створено просту форму реєстрації, що складається з чотирьох полів: логіну, адреси електронної пошти, пароля та його підтвердження.

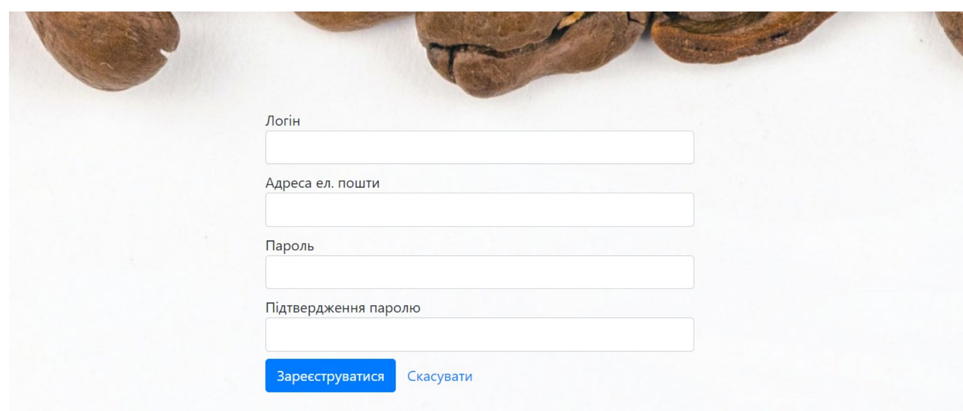


Рис. 3.1. Форма реєстрації

На цю форму накладено певні обмеження, а саме:

- всі поля повинні бути заповненими;
- адреса електронної пошти повинна відповідати відповідному шаблону;
- адреса електронної пошти повинна бути унікальною;
- поля пароля та його підтвердження повинні мати однакові значення.

Якщо користувач заповнив всі поля згідно до вимог та натиснув кнопку «Зареєструватися», то на вказану адресу електронної пошти буде надіслано листа для підтвердження її справжності та можливості доступу до неї користувача, що реєструється, а сам користувач отримає відповідне повідомлення.

Дякуємо за реєстрацію!
На Вашу пошту було відправлено листа для підтвердження реєстрації.

Рис. 3.2. Повідомлення про необхідність перевірки поштової скриньки

3.2. Підтвердження адреси електронної пошти

Після надсилання форми реєстрації на вказану адресу надсилається лист-підтвердження з унікальним посиланням для активації облікового запису.

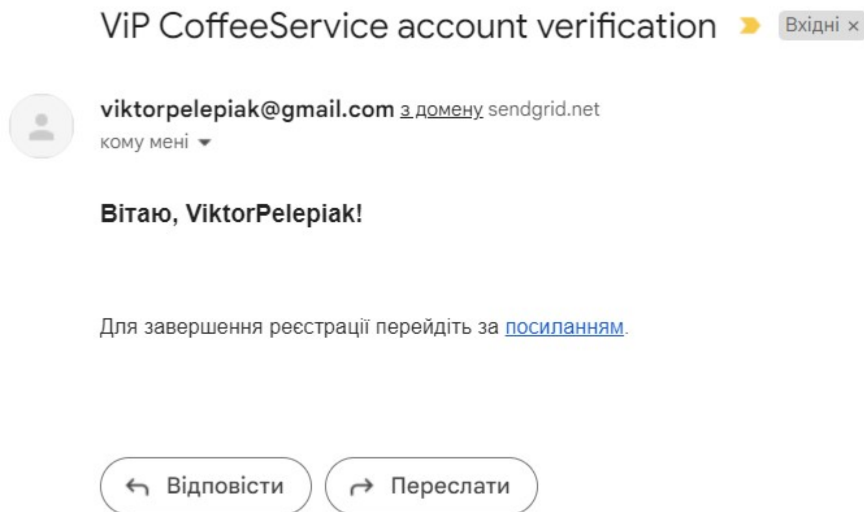


Рис. 3.3. Приклад листа-підтвердження

Перейшовши за цим вказаним посиланням користувач потрапить на сторінку перевірки ключа підтвердження та побачить відповідне повідомлення. У випадку, коли все пройшло добре, ключ валідний та не протермінований буде відображено повідомлення про успішну активацію облікового запису.

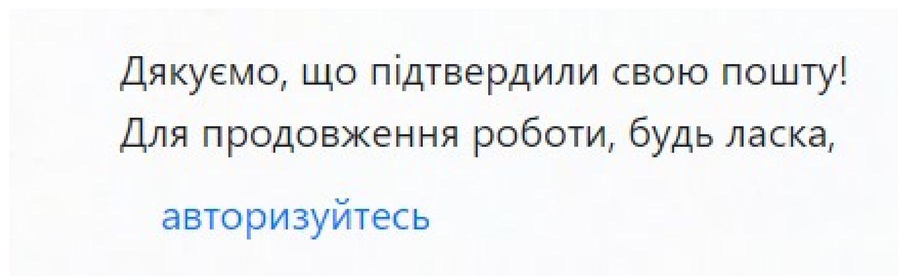


Рис. 3.4. Повідомлення про успішну активацію

Проте якщо з моменту отримання листа до переходу по посиланню пройде занадто багато часу, то ключ активації облікового запису втратить свою актуальність та більше не буде дійним і повідомлення з цією інформацією буде показано користувачеві. Хоч ключ і не є вже дійсним проте адреса електронної пошти є дійсно валідною і все ж може використовуватись. Спеціально для таких випадків був реалізований механізм повторної відправки посилання для

активації облікового запису. Для здійснення повторної відправки ключа достатньо натиснути на відповідну кнопку.

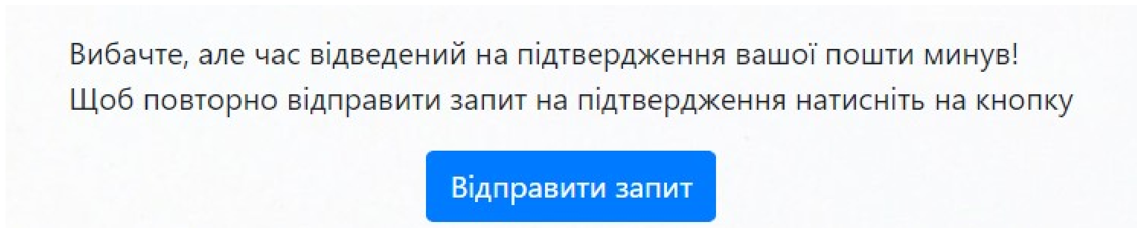


Рис. 3.5. Повідомлення про протермінування ключа

На випадок повторного переходу по вже використаному або ж неіснуючому посиланню було передбачено виведення відпоівдного поідомлення.

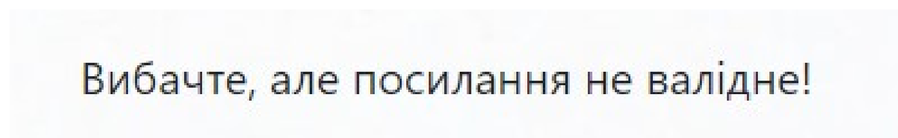


Рис. 3.6. Повідомлення про недійсне посилання

3.3. Вхід в обліковий запис

Вхід в обліковий запис користувача здійснюється через просту форму, що складається з двох полів: імені користувача та паролю. При цьому, в якості імені користувача можна використовувати як відповідне значення, що вказувалось при реєстрації, так і адресу електронної пошти.

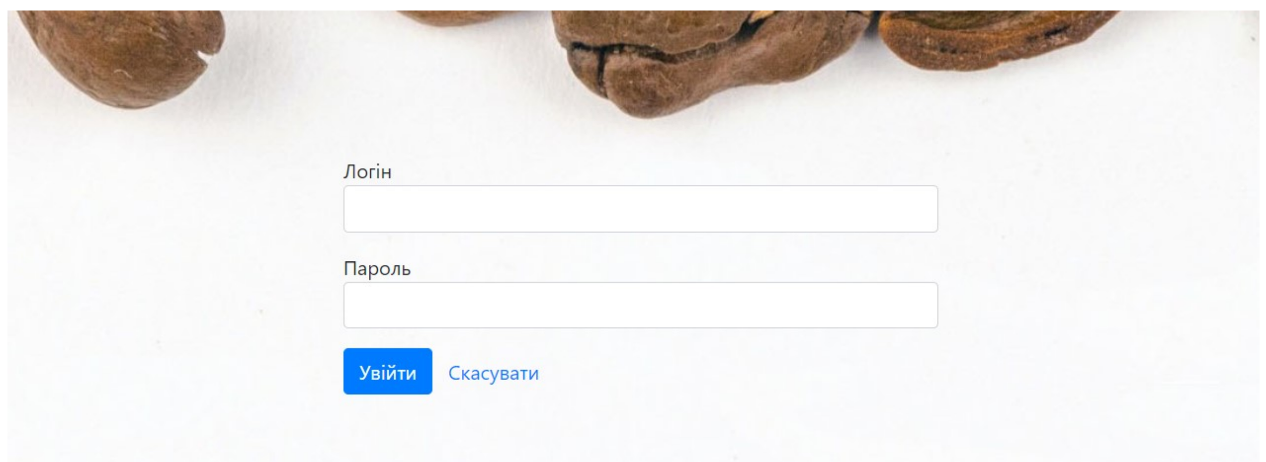


Рис. 3.7. Форма входу в обліковий запис

У разі успішної автентифікації користувача буде переадресовано на головну сторінку, а в протилежному випадку – нотифіковано про помилку.

3.4. Адміністрування

Невід’ємною частиною функціоналу розробленого додатку є адміністрування. Таким чино було реалізовано інструменти для управління брендами, моделями, деталями та характеристиками кавових апаратів. Доступ до цього функціоналу є тільки в користувачів з правами адміністратора.

Щоб перейти на відповідні сторінки необхідно і достатньо натиснути на кнопку «Адміністрування» та вибрати потрібну сторінку.

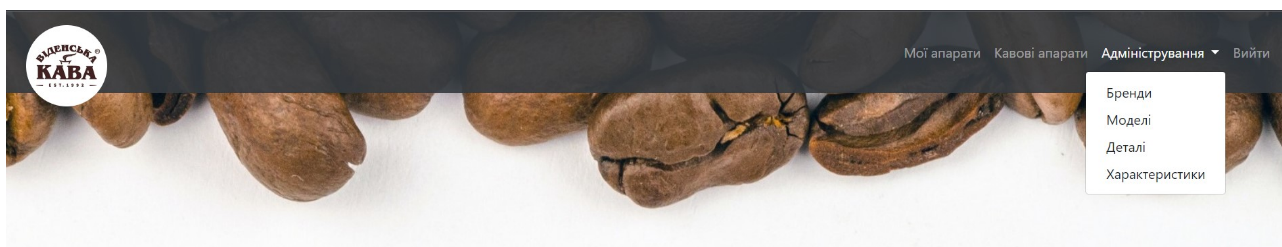


Рис. 3.8. Контекстне меню адміністрування

Всі сторінки адміністрування подібні між собою та надають однакові можливості по адмініструванню відповідних параметрів. В якості прикладу розглянемо сторінку адміністрування моделей кавових апаратів.

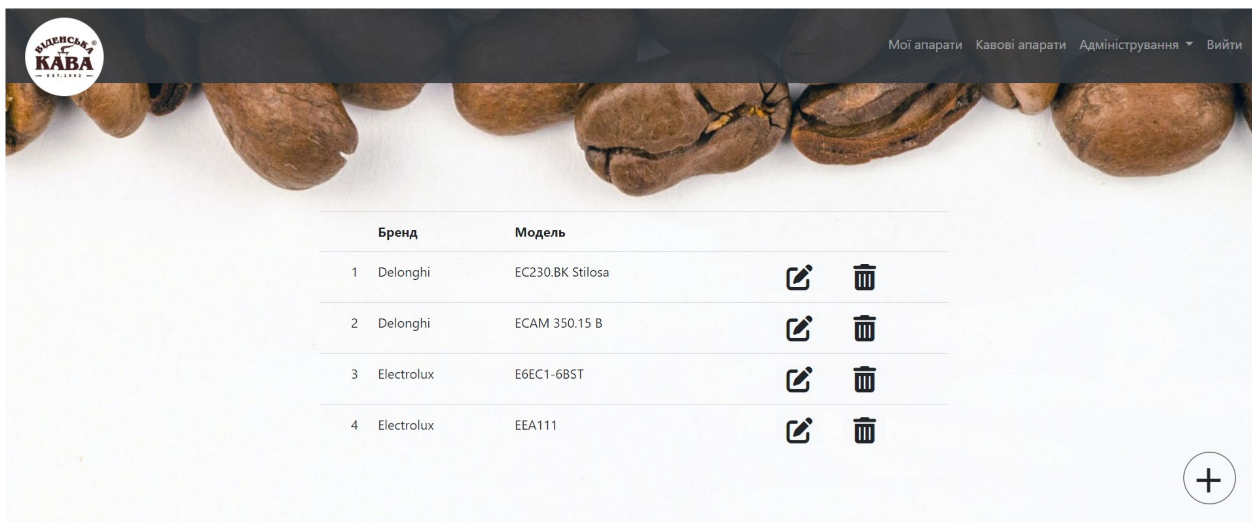


Рис. 3.9. Сторінка адміністрування моделей

Сторінки адміністрування володіють надзвичайно простим та інтуїтивно зрозумілим інтерфейсом, що дозволяє легко та швидко оперувати записами. Таким чином для додавання нового запису достатньо натиснути на кнопку «+» в нижній правій частині екрану, для видалення існуючого запису на кнопку зі значком кошика навпроти відповідного рядка, а для редагування – олівця. При

цьому будь-які дії на цій сторінці супроводжуються появою модальних вікон, що сприяє мінімізації шансів виникнення помилкових операцій.

Рис. 3.10. Приклад модального вікна видалення запису

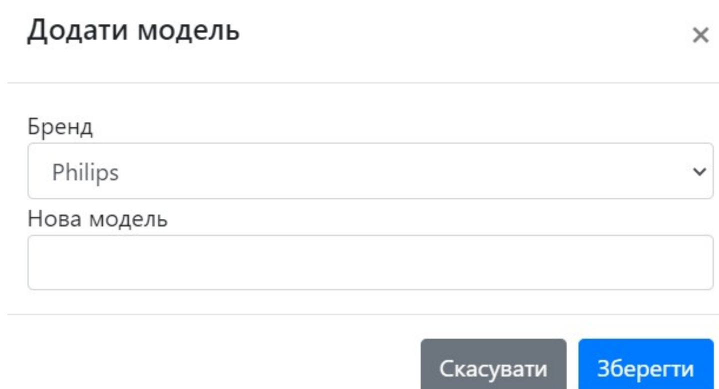


Рис. 3.11. Приклад модального вікна додавання запису

3.5. Додавання кавових апаратів до реєстру та перегляд записів

Чи не найважливішою частиною розробленого програмного продукту є функціонал присвячений створенню записів про кавові апарати та їх систематизація.

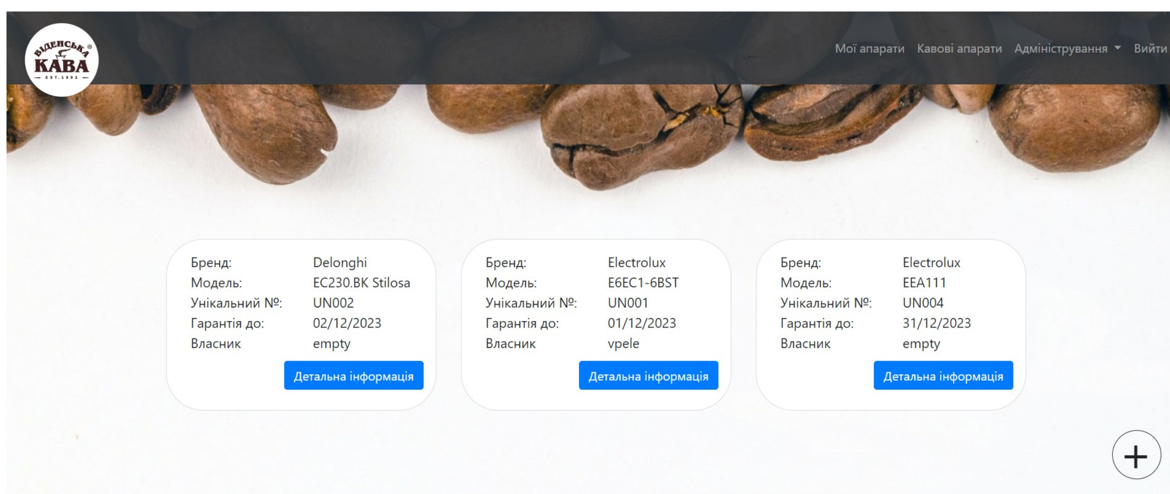


Рис. 3.12. Сторінка перегляду всіх кавових апаратів

Щоб потрапити на сторінку перегляду всіх кавових апаратів необхідно натиснути на кнопку «Кавові апарати», що знаходиться у верхній правій частині. На цій сторінці відображається коротка інформація про кожен кавовий апарат, що проходив ремонт або обслуговування в майстерні. Для отримання детальної інформації про конкретний апарат необхідно натиснути на кнопку «Детальна інформація» на відповідній картці.

Бренд: Electrolux
Модель: E6EC1-6BST
Унікальний номер апарату: UN001
Гарантія до: 01/12/2023

Деталь	Характеристика	Значення	Одиниці виміру
Бойлер	Макс.температура	80	°C
	Потужність	200	Вт

Додаткова інформація
Twst machine Mk1

Дані про обслуговування

№	Тип операції	Статус	Дата початку	Дата завершення	Зробити до	Відповідальний
1	Діагностика	Готово	2023-11-18T09:03	2023-11-18T09:04	2023-11-17T12:00	vpele

Рис. 3.13. Приклад сторінки з детальною інформацією про апарат

На цій сторінці можна отримати інформацію про бренд, модель, унікальний номер-ідентифікатор, деталі кавового апарату та їх характеристики, гарантійний термін, додаткову інформацію, а також дані про обслуговування в короткому форматі.

Для перегляду детальної інформації про обслуговування можна натиснути на значок збільшувального скла навпроти відповідного запису.

Тип операції: Діагностика
Статус: Готово
Майстер: vpele
Оцінка тривалості: 1h
Зробити до: 17.11.2023 12:00
Дата початку: 18.11.2023 09:03
Дата завершення: 18.11.2023 09:04
Вхідні дані: Підчас роботи млина чути посторонній шум.
Робочі записи: Було проведено чистку та перевірку млина. Виявлено незначне зміщення робочої частини млина. Необхідно виставити робочу частину в правильне положення.

Рис. 3.14. Приклад сторінки з повною інформацією про обслуговування

Щоб додати новий кавовий апарат або його шаблон необхідно натиснути на кнопку «+» в правому нижньому кутку сторінки перегляду всіх кавових апаратів.

Рис. 3.15. Приклад створення шаблону апарату

Рис. 3.15. Приклад створення апарату

Для створення нового шаблону кавового апарату та його детального запису використовується одна і та ж сторінка. Для позначення того, що новий запис буде шаблоном достатньо виставити істинне значення для прапорця «Зберегти як шаблон». При цьому з форми буде прибрано декілька полів, що не є загальними. Враховуючи простоту розробленого інтерфейсу, заповнення форми є зручним та інтуїтивно зрозумілим. Звернути увагу варто тільки на частину, що відповідає за деталі та їх характеристики, яка реалізована через вкладені таблиці. Таким чином спочатку необхідно додати деталь, натиснувши на кнопку «+» внизу цієї секції. Після додавання нових деталей до апарату

навпроти них з’являються додаткові кнопки «+», які слугують для додавання їх характеристик.

Видалення деталей та характеристик відбувається шляхом натискання на кнопку зі зображенням кошика навпроти відповідного компоненту. Варто відзначити, що в цьому випадку модальні вікна для підтвердження дій не використовуються, тому потрібно бути обережним.

3.6. Додавання придбаних кавових апаратів

Користувачі програмного продукту з правами клієнта можуть переглядати інформацію про придбані ними апарати та поновлювати цей список. Щоб переглянути цю інформацію достатньо перейти на сторінку «Мої апарати».

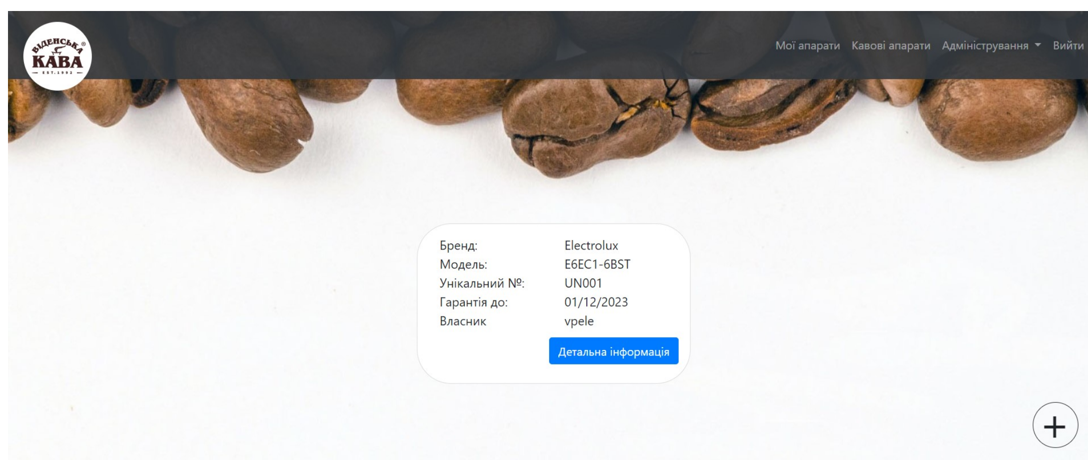


Рис. 3.16. Сторінка перегляду придбаних апаратів

Якщо придбаний користувачем апарат не додався до списку автоматично, то його можна додати самостійно. Для цього передбачена кнопка «+» в правому нижньому кутку сторінки. Натиснувши на неї користувач побачить модальне вікно з полем для вводу унікального номера апарату.

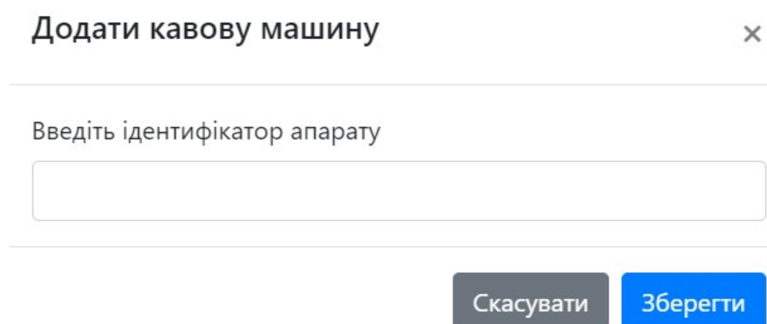


Рис. 3.17. Модальне вікно додавання придбаного апарату

ВИСНОВКИ

У кваліфікаційній роботі «Розробка вебдодатку для майстерні по ремонту та обслуговуванню кавових апаратів» засобами мов програмування Java та Typescript, фреймворків Spring Boot та Angulr успішно реалізовано відповідний вебдодаток, що є завершеним продуктом, який поєднанує ефективну MVC-архітектуру, високу швидкодію та зручний і зрозумілий інтерфейс користувача.

При виконанні кваліфіувційної роботи було отримано такі результати:

- покращено навички роботи з сучасними технологіями розробки вебдодатків;
- здійснено проектування програмного продукту для майстерні по ремонту та обслуговуванню кавових апаратів;
- здійснено програмну реалізацію відповідного продукту.

В кваліфікаційній роботі описані:

- використані технології та їх переваги;
- генерація та структура проекту;
- етапи проектування та реалізації програмного продукту;
- інструкція користувача.

Використання розробленого програмного продукту сприятиме полегшенню та спрощенню ведення обліку кавових апаратів та записів щодо їх обслуговування майстернею по ремонту та обслуговуванню.

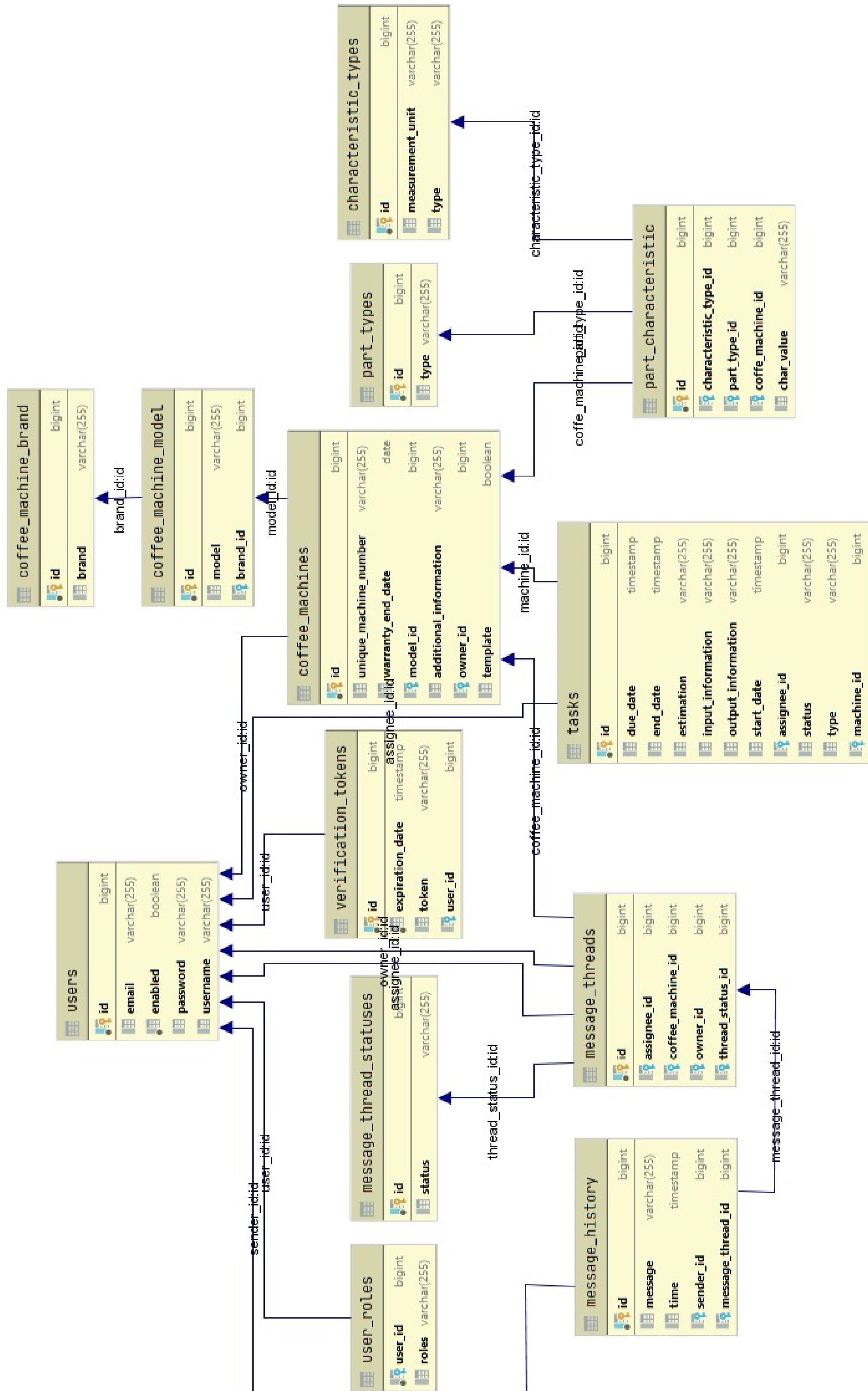
Додаток можна легко модифікувати, впроваджуючи новий функціонал або редагуючи існуючий, у випадку виникнення такої потреби.

ПЕРЕЛІК ПОСИЛАНЬ

1. Cay S. Horstmann Core Java. Volume I: Fundamentals 12th Edition. – Oracle Press, 2021.
2. Musib Somnath. Spring Boot in Practice. – Manning Publications, 2022. – 584 p.
3. Jeff Langr, Andy Hunt, Dave Thomas. Pragmatic Unit Testing in Java 8 with JUnit. – Pragmatic Bookshelf, 2015. – 236 p.
4. Cătălin Tudose. Java Persistence with Spring Data and Hibernate – Manning, 2023. – 616p.
5. Greg Lim. Beginning Angular with Typescript. – Sticky notes, 2017. – 222p.
6. Проект додатку на Github (бекенд) [Електронний ресурс]. – Режим доступу : URL : **https://github.com/ViktorPelepiak/vip_coffe_service_back**
7. Проект додатку на Github (фронтенд) [Електронний ресурс]. – Режим доступу : URL : **https://github.com/ViktorPelepiak/vip_coffe_service_front**

ДОДАТКИ

Додаток А. Схема Бази даних.



Додаток Б. Програмний код класу EmailServiceImpl

```
package com.vip.coffee.service.services.impl;

import com.sendgrid.Method;
import com.sendgrid.Request;
import com.sendgrid.Response;
import com.sendgrid.SendGrid;
import com.sendgrid.helpers.mail.Mail;
import com.sendgrid.helpers.mail.objects.Email;
import com.vip.coffee.service.enums.EmailType;
import com.vip.coffee.service.exceptions.EmailException;
import com.vip.coffee.service.mail.EmailInstance;
import com.vip.coffee.service.mail.MailGenerator;
import com.vip.coffee.service.services.EmailService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.stereotype.Service;

import java.io.IOException;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

@Service
@PropertySource("/email.properties")
public class EmailServiceImpl implements EmailService {

    private final Environment environment;
    private final Map<EmailType, MailGenerator> mailGenerators;

    @Autowired
    EmailServiceImpl(Environment environment, List<MailGenerator>
mailGenerators) {
        this.environment = environment;
        this.mailGenerators = mailGenerators.stream()
                .collect(Collectors.toMap(
                        MailGenerator::getEmailType,
                        value -> value));
    }

    public void sendEmail(EmailType type, String emailTo, Map<String, Object>
parameters) {
        MailGenerator mailGenerator = mailGenerators.get(type);
        if (mailGenerator == null) {
            throw new UnsupportedOperationException(
                    String.format("Emails with type <%s> are not supported",
type.name()));
        }
        EmailInstance emailInstance = mailGenerator.generate(parameters);

        Mail mail = new Mail(
                emailInstance.getFrom(),
                emailInstance.getSubject(),
                new Email(emailTo),
                emailInstance.getContent());

        SendGrid sendGrid = new
SendGrid(environment.getProperty("sendgrid.api.key"));
        Request request = new Request();
```

```
        try{
            request.setMethod(Method.POST);
            request.setEndpoint("mail/send");
            request.setBody(mail.build());
            Response response = sendGrid.api(request);
        } catch (IOException e) {
            throw new EmailException("Error during email sending");
        }
    }
}
```

Додаток В. Програмний код класу CoffeeMachineController

```
package com.vip.coffee.service.controller;

import com.vip.coffee.service.dto.*;
import com.vip.coffee.service.exceptions.ElementNotFoundException;
import com.vip.coffee.service.model.CoffeeMachine;
import com.vip.coffee.service.rest.GenericResponse;
import com.vip.coffee.service.services.CoffeeMachineService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping("machines")
public class CoffeeMachineController {

    private final CoffeeMachineService coffeeMachineService;

    @Autowired
    public CoffeeMachineController(CoffeeMachineService coffeeMachineService) {
        this.coffeeMachineService = coffeeMachineService;
    }

    @GetMapping
    public GenericResponse<List<CoffeeMachineShortDto>> getAllMachines() {
        return GenericResponse.of(
                coffeeMachineService.getAllMachines().stream()
                        .map(CoffeeMachineShortDto::toDto)
                        .collect(Collectors.toList())
        );
    }

    @GetMapping("{machineId}")
    public GenericResponse<CoffeeMachineDto> getCoffeeMachineById(@PathVariable
Long machineId) {
        try {
            return
GenericResponse.of(CoffeeMachineDto.toDto(coffeeMachineService.getById(machineId
)));
        } catch (ElementNotFoundException e) {
            return GenericResponse.error(e.getMessage());
        }
    }

    @GetMapping("exist/{uniqMachineNumber}")
    public GenericResponse<Boolean> isMachineWithUniqueNumberExist(@PathVariable
String uniqueMachineNumber) {
        return
GenericResponse.of(coffeeMachineService.isMachineWithUniqNumberExist(uniqueMachi
neNumber));
    }

    @GetMapping("template")
    public GenericResponse<List<CoffeeMachineTemplateShortDto>>
getAllTemplates() {
        return GenericResponse.of(
                coffeeMachineService.getAllTemplates().stream()
                        .map(CoffeeMachineTemplateShortDto::toDto)
```

```
                        .collect(Collectors.toList())
        );
    }

    @GetMapping("template/{templateId}")
    public GenericResponse<CoffeeMachineTemplateDto>
getTemplateById(@PathVariable Long templateId) {
        try {
            return
GenericResponse.of(CoffeeMachineTemplateDto.toDto(coffeeMachineService.getTempla
teById(templateId)));
        } catch (ElementNotFoundException e) {
            return GenericResponse.error(e.getMessage());
        }
    }

    @GetMapping("my")
    GenericResponse<List<CoffeeMachineShortDto>> getAllMachinesForLoggedUser() {
        try {
            return GenericResponse.of(
                    coffeeMachineService.getAllMachinesForLoggedUser().stream()
                            .map(CoffeeMachineShortDto::toDto)
                    .collect(Collectors.toList())
            );
        } catch (ElementNotFoundException e) {
            return GenericResponse.error(e.getMessage());
        }
    }

    @PostMapping
    public GenericResponse<CoffeeMachine> save(@RequestBody SaveMachineDto
saveMachineDto) {
        try {
            return
GenericResponse.of(coffeeMachineService.saveMachine(saveMachineDto));
        } catch (ElementNotFoundException e) {
            return GenericResponse.error(e.getMessage());
        }
    }

    @PostMapping("template")
    public GenericResponse<CoffeeMachine> saveTemplate(@RequestBody
SaveMachineTemplateDto saveMachineDto) {
        try {
            return
GenericResponse.of(coffeeMachineService.saveMachineTemplate(saveMachineDto));
        } catch (ElementNotFoundException e) {
            return GenericResponse.error(e.getMessage());
        }
    }

    @PostMapping("add")
    public GenericResponse<CoffeeMachineShortDto> addMachine(@RequestBody String
uniqNumber) {
        try {
            return
GenericResponse.of(CoffeeMachineShortDto.toDto(coffeeMachineService.addMachineWi
thUniqNumber(uniqNumber)));
        } catch (ElementNotFoundException e) {
            return GenericResponse.error(e.getMessage());
        }
    }
}
```

Додаток Г. Програмний код класу CoffeeMachineServiceImpl

```
package com.vip.coffee.service.services.impl;

import com.vip.coffee.service.dto.PartTypesWithCharacteristicsDto;
import com.vip.coffee.service.dto.SaveMachineDto;
import com.vip.coffee.service.dto.SaveMachineTemplateDto;
import com.vip.coffee.service.exceptions.ElementNotFoundException;
import com.vip.coffee.service.model.CoffeeMachine;
import com.vip.coffee.service.model.User;
import com.vip.coffee.service.repository.CoffeeMachineRepository;
import com.vip.coffee.service.services.CoffeeMachineModelService;
import com.vip.coffee.service.services.CoffeeMachineService;
import com.vip.coffee.service.services.PartCharacteristicService;
import com.vip.coffee.service.services.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Service;

import javax.transaction.Transactional;
import java.time.LocalDate;
import java.util.LinkedList;
import java.util.List;

@Service
@Transactional
public class CoffeeMachineServiceImpl implements CoffeeMachineService {

    private final CoffeeMachineRepository coffeeMachineRepository;
    private final CoffeeMachineModelService modelService;
    private final PartCharacteristicService partCharacteristicService;
    private final UserService userService;

    @Autowired
    public CoffeeMachineServiceImpl(CoffeeMachineRepository
coffeeMachineRepository,
                                    CoffeeMachineModelService modelService,
                                    PartCharacteristicService
partCharacteristicService, UserService userService) {
        this.coffeeMachineRepository = coffeeMachineRepository;
        this.modelService = modelService;
        this.partCharacteristicService = partCharacteristicService;
        this.userService = userService;
    }

    @Override
    public List<CoffeeMachine> getAllMachines() {
        return coffeeMachineRepository.findAllMachines();
    }

    @Override
    public List<CoffeeMachine> getAllTemplates() {
        return coffeeMachineRepository.findAllTemplates();
    }

    @Override
    public CoffeeMachine saveMachine(SaveMachineDto saveMachineDto) throws
ElementNotFoundException {
        String[] date = saveMachineDto.getWarrantyEndDate().split("-");
        return coffeeMachineRepository.save(
                saveMachineTemplate(saveMachineDto)
```

```
                        .setTemplate(false)

.setUniqueMachineNumber(saveMachineDto.getUniqMachineNumber())

.setAdditionalInformation(saveMachineDto.getAdditionalInformation())
                        .setWarrantyEndDate(LocalDate.of(
                                Integer.parseInt(date[0]),
                                Integer.parseInt(date[1]),
                                Integer.parseInt(date[2])
                        ))
        );
    }

    @Override
    public CoffeeMachine saveMachineTemplate(SaveMachineTemplateDto
saveMachineTemplateDto) throws ElementNotFoundException {
        CoffeeMachine coffeeMachine = coffeeMachineRepository.save(new
CoffeeMachine()

.setModel(modelService.getById(saveMachineTemplateDto.getModelId()))
                .setTemplate(true)
        );
        if (coffeeMachine.getParts() == null) {
            coffeeMachine.setParts(new LinkedList<>());
        }

        for (PartTypesWithCharacteristicsDto ptwc :
saveMachineTemplateDto.getPartTypesWithCharacteristics()) {

coffeeMachine.getParts().addAll(partCharacteristicService.saveAll(coffeeMachine,
ptwc));
        }

        return coffeeMachineRepository.save(coffeeMachine);
    }

    @Override
    public CoffeeMachine getTemplateById(Long templateId) throws
ElementNotFoundException {
        return
coffeeMachineRepository.findFirstTemplateById(templateId).orElseThrow(ElementNot
FoundException::new);
    }

    @Override
    public boolean isMachineWithUniqNumberExist(String uniqMachineNumber) {
        try {

coffeeMachineRepository.findFirstByUniqueMachineNumber(uniqMachineNumber)
                    .orElseThrow(ElementNotFoundException::new);
            return true;
        } catch (ElementNotFoundException e) {
            return false;
        }
    }

    @Override
    public CoffeeMachine getById(Long machineId) throws ElementNotFoundException
{
        return coffeeMachineRepository.findById(machineId).orElseThrow(() -> new
ElementNotFoundException(
                String.format("Can't find machine wth id=\"%s\"", machineId)
        ));
```

```
    }

    @Override
    public List<CoffeeMachine> getAllMachinesForLoggedUser() throws
ElementNotFoundException {
        return new LinkedList<>(

userService.findByUsername(SecurityContextHolder.getContext().getAuthentication(
).getName())
                        .getCoffeeMachines()
        );
    }

    @Override
    public CoffeeMachine addMachineWithUniqNumber(String uniqNumber) throws
ElementNotFoundException {
        User loggedUser =
userService.findByUsername(SecurityContextHolder.getContext().getAuthentication(
).getName());
        CoffeeMachine machine =
coffeeMachineRepository.findFirstByUniqueMachineNumber(uniqNumber).orElseThrow((
) ->
                new ElementNotFoundException(
                        String.format("Can't find machine with uniq number
\"%s\"", uniqNumber)
                )
        );
        machine.setOwner(loggedUser);
        loggedUser.getCoffeeMachines().add(machine);
        coffeeMachineRepository.save(machine);
        userService.update(loggedUser);

        return machine;
    }
}
```