math_2023_022

Міністерство освіти і науки України
Чернівецький національний університет імені Юрія Федьковича

Факультет математики та інформатики
(повна назва інституту/факультету)

Кафедра математичного моделювання
(повна назва кафедри)

Створення веб-сайту для обліку продукції та менеджменту працівників

Кваліфікаційна робота
Рівень вищої освіти - другий (магістерський)

Виконав:
студент 6 курсу, групи 607
спеціальності 124 – Системний аналіз
(назва спеціальності)
Пітей Едуард Іванович
(прізвище, ім'я, по-батькові)
Керівник д.ф.-м.н., проф. Черевко І. М.
(науковий ступінь, вчене звання, прізвище та ініціали)

До захисту допущено:
Протокол засідання кафедри № ___
від "__" _____ 2023 р.
зав. кафедри _____ проф. Черевко І.М.

Чернівці – 2023

Анотація

У кваліфікаційній роботі розроблено веб-сайт для обліку продукції та менеджменту персоналу підприємства. Основна розв’язана задача це розподіл коштів та відслідковування кількості продукції в реальному часі.

Ключові слова: Архітектура, реляційне відображення, токен, база даних, сервіс, розробка веб додатків

Annotation

In the diploma, the website was developed for product accounting and personnel management of the enterprise. The problem of fund distribution and real-time tracking of production quantities was solved.

Key words: Architecture, relational mapping, token, database, service, web application development

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

_____ Е.І. Пітей

(підпис)

ЗМІСТ

Вступ

Найбільш поширеним фруктом в Україні є яблука. Їх вживають у свіжому вигляді, переробляють на сік, варення, джеми і компоти, додають у випічку. Крім того, плоди цієї культури мають великий запас корисних компонентів, включаючи вітамін С і залізо.

Велика частка чернівецьких фермерів відома тим, що вирощує саме цей фрукт.

Ринок перероблених продуктів – один із найбільших ланок господарства, що стрімко розвиваються. За міжнародною класифікацію перероблені продукти – це продукти харчування, що перероблені згідно з умовами екологічної обробки, виважена екологічна маркетингова політика, а якість підтверджена сертифікатом.

Попит на перероблену продукцію в Європі зростає, за шість років ринок у грошах виріс більш ніж у 10 разів. Один з напрямків найбільшого приросту та великої ємності – ринок концентрованого соку. Для України має сенс розглянути можливість охоплення даного ринку через розвиток виробництва в даній сфері та наявність конкурентоспроможних умов на ринку. Світовий ринок концентрованого соку – відносно не новий на ринку соків, незважаючи на це в останні роки він динамічно розвивається. Об’єм виробництва щорічно зростає на 10–30%, а експорт збільшується в середньому на 25%.

Даний ринок ще не є глибоко дослідженим, тому існує необхідність розглянути основні тенденції. Основною проблемою такого виробництва є можливість збору великої кількості продукції для забезпечення безперервної роботи виробничих потужностей, для цього потрібно:

1. Локалізувати громади де знаходиться найбільша кількість яблуневих садів.

2. Відкрити пункти прийому та найняти працівників.
3. Налагодити логістику.
4. Формувати великі партії продукції, щоб задовольнити потреби виробничих потужностей.

Проаналізувавши проблеми пов’язані з полегшенням контролю та управління для підприємства, що займаєтьтся такою тематикою, виявлено актуальною задачею створення системи обліку продукції та менеджменту працівників. Для цих цілей розроблено веб-сайт, який має реалізований такий функціонал:

1. Створення пунктів прийому.
2. Внесення даних працівників.
3. Прикріплення працівника до пункту прийому.
4. Відслідковування коштів та кількості продукції на пунктах прийому.
5. Відслідковування історії зміни балансу.

РОЗДІЛ 1

РОЗРОБКА ТЕХНІЧНОГО ТА РОБОЧОГО ПРОЕКТУ

1.1 Вимоги до програмної документації

Кожен етап проектування завершується оформленням відповідних документів. Тому важливим елементом проектування програмних додатків є оформлення програмної документації. Винятком можуть бути прості програми з коротким життєвим циклом і низькою трудомісткістю.

Зміст програмних документів:

− специфікація – реєстр і призначення усіх файлів програмного продукту, разом із файлами документації;

− текст програми – запис програмних кодів і коментарів до них; - програма і методика тестування - перелік і опис вимог, які необхідно перевірити під час тестування програми, методи контролю.

− технічне завдання - документ, який описує мету та сферу застосування програми, вимоги до програмного продукту, етапи та терміни розробки, види тестувань.

− пояснювальна записка – опис прийнятих і використаних технічних і техніко-економічних висновків, схеми та детальний опис алгоритмів, загальний опис працюючого програмного продукту.

До програмних документів належать також документи, що забезпечують функціонування та дію програм - експлуатаційні документи:

− опис мови – містить синтаксис і семантику мови;

− керівництво з технічного підтримування – включає відомості для використання;

− випробувальних та діагностичних засобів при підтримці технічних програм.

1.2 Стадії й етапи розробки програмного проекту

Розробка продукту має бути поділена на три кроки:

1) розробка технічного завдання;

2) розробка проекту;

3) впровадження.

На першому кроці має бути завершений крок розробки: узгодження та затвердження даного технічного завдання. На кроці робочого проектування необхідно виконати наступні етапи робіт:

1) розробка продукту;

2) розробка документації;

3) випробування продукту. Зміст робіт по кроках.

На кроці розробки технічного завдання виконуються такі роботи:

1) постановка завдань;

2) визначення та уточнення вимог до технічних засобів;

3) визначення вимог до програми;

4) визначення стадій, етапів і термінів розробки програми та документації на неї;

5) вибір мов програмування;

6) узгодження та затвердження технічного завдання.

1.3 Порядок контролю і прийому

Перевірка програмної документації виконується замовником самостійно із стороннім поглядом експертів, які можуть засвідчити відповідність створеного програмного продукту всім пунктам технічної документації, включаючи технічне завдання та технічний проєкт.

Випробування програми має проводитись під час створення продукту самим розробником:

1) з використанням контрольних тестів, що перевіряють критичний функціонал програми;
2) із залученням простих користувачів чи тестерів, які повинні повідомляти про помилки, які виникли під час користування програмою.

Також тестуванням займається замовник після завершення створення програми:
1) з використанням тестів;
2) за допомогою кваліфікованих сторонніх працівників.

Розділ 2. Опис використаних технологій та архітектури

2.1 Архітектура

Для досягнення вищезазначених цілей було вирішено використовувати архітектуру побудови **REST API** [1].

REST - розшифровується як Representational State Transfer, перекладається як "передача репрезентативного стану". Це стиль розробки розподілених систем, що обмежуються форматом JSON. Основною абстракцією в REST є ресурс.

Інтерфейси програмування застосунків (API) надають платформу та середовище для програм, які дають змогу їм обмінюватись інцформацією та розуміти один одного. API визначають спосіб структурування інформації під час обміну програмами даними та інформацією.

В архітектура REST вимагається розподіл обов'язків між частинами, які несуть відповідальність за зберігання та оновлення даних (сервер), та іншими частинами, які несуть відповідальність за відображення даних в інтерфейсі кінцевого споживача та реагування на дії з цим клієнтським інтерфейсом. Що дозволяє різним частинам продукту розвиватися незалежно. REST API користується HTTP протоколом для передачі даних від сервера до клієнта і навпаки. Найчастіше для обміну інформації використовують формат - JSON.

JSON - це текстовий відкритий формат обміну даними, який базується на JavaScript. Він використовується для представлення структурованих даних та об'єктів у вигляді тексту. JSON є легким у читанні та написанні для людини, і в той же час легко обробляється мовами програмування. Основне призначення JSON - це обмін даними між сервером і клієнтом, зокрема в контексті веб-програмування. JSON використовується для передачі структурованої інформації із збереженням простоти та легкості використання.

2.2 Мова програмування Java [4-5, 12]

Мова програмування Java є об'єктно-орієнтованою мовою, розробленою компанією Sun Microsystems (згодом придбаною Oracle Corporation). Вона стала популярною завдяки своїй платформонезалежності, високій продуктивності та безпеці. Ось деякі основні характеристики, плюси та мінуси мови програмування

Плюси:

Платформонезалежність:

- Код, написаний на Java, може виконуватися на будь-якому пристрої або операційній системі, яка має відповідну віртуальну машину Java (JVM). Це дозволяє створювати крос-платформенні застосунки.

Об'єктно-орієнтована:

- Java підтримує об'єктно-орієнтований підхід, що полегшує організацію та структуру коду.

Багатопотоковість:

- Мова має вбудовану підтримку для роботи з багатьма потоками виконання, що дозволяє створювати ефективні та швидкодіючі застосунки.

Безпека:

- Java має вбудовані механізми безпеки, такі як відсічення буферу, система контролю доступу та автоматичне управління пам'яттю, що робить код менш вразливим до атак.

Широке співтовариство:

- Java має велику та активну спільноту розробників, що полегшує отримання підтримки та вирішення проблем

Мінуси:

Високі вимоги до ресурсів:

- Деякі застосунки Java можуть вимагати значних обсягів пам'яті та процесорної потужності, що може бути проблематичним для деяких вбудованих систем чи пристроїв з обмеженими ресурсами.

Помітний запуск:

- Програми на Java запускаються відносно повільно порівняно з мовами, компільованими безпосередньо в машинний код.

Важкість управління ресурсами:

- В управлінні пам'яттю використовується автоматичне сміттєзбіральник, що може призводити до несподіваних затримок в роботі програми.

Неінтуїтивний синтаксис для деяких розробників:

- Деякі програмісти можуть вважати синтаксис Java менш інтуїтивним порівняно з іншими мовами програмування.

Відсутність деяких сучасних фіч:

- Деякі нововведення в мовах програмування можуть з'являтися в Java з певним запізненням, порівняно з іншими мовами.

2.3 Spring Framework [1-2, 7]

Spring Framework - це обширний фреймворк для розробки програмних застосунків на платформі Java. Він надає комплексні рішення для різних завдань, включаючи управління бізнес-логікою, управління транзакціями, обробку винятків, доступ до даних, розробку веб-застосунків та багато іншого. Spring спрощує розробку, підтримує кращі практики програмування та сприяє створенню ефективних та легко розширюваних застосунків. Основні компоненти та концепції Spring включають:

IoC (Inversion of control - Інверсія керування):

- Spring використовує принцип Інверсії Керування, що означає, що контейнер Spring керує об'єктами життєвого циклу (бінами). Контейнер відповідає за створення, налаштування та управління залежностями бінів, а розробник визначає компоненти та їх залежності.

DI (Dependency injection - Впровадження залежностей):

- Це конкретна реалізація принципу Інверсії Керування. Spring дозволяє впроваджувати залежності між компонентами, передаючи їх через конструктор, метод чи поле об'єкта. Це полегшує тестування та забезпечує слабку залежність між компонентами.

AOP (Aspect-Oriented Programming - Програмування з орієнтацією на аспекти):

- Spring надає підтримку для аспектно-орієнтованого програмування, що дозволяє відділити аспекти, такі як журналювання, від основної бізнес-логіки. Це покращує модульність та повторне використання коду.

Spring MVC (Model-View-Controller):

- Spring надає власну реалізацію шаблону проектування Model-View-Controller для веб-застосунків. Він дозволяє ефективно розділити логіку застосунку на модель, представлення та контролер.

Spring Data:

- Цей підпроект Spring надає узагальнений доступ до різних сховищ даних, таких як бази даних, кеші та інші. Spring Data спрощує взаємодію з різними технологіями зберігання даних.

6. Spring Boot:

- Spring Boot є підпроектом Spring, який спрощує процес розробки застосунків, надаючи конфігурації за замовчуванням

та автоматизуючи рутинні задачі. Він дозволяє швидко створювати готові до використання застосунки з найменшими ускладненнями.

Плюси Spring Framework:

Модульність та Розширюваність:

- Spring дозволяє створювати застосунки, які легко розширювати та підтримувати завдяки його модульній структурі.

Широкі Можливості Конфігурації:

- Велика гнучкість у конфігурації та використання різних джерел конфігурації.

Спрощена Розробка Веб-Застосунків:

- Spring MVC дозволяє швидко створювати веб-застосунки та розділяти їх логіку.

Підтримка Транзакцій:

- Вбудована підтримка транзакцій полегшує управління конкурентністю та цілістю даних.

Мінуси Spring Framework:

Навчання та Початковий Навігаційний Поріг:

- Для новачків може бути складно вивчити всі аспекти та можливості Spring.
- Для деяких розробників може бути складно розуміти внутрішню роботу фреймворку через великий рівень абстракції.
- Деякі проекти можуть включати значний обсяг коду через велику кількість можливостей та опцій Spring.

Незважаючи на це, Spring залишається одним з найпопулярніших та використовуваних фреймворків у світі Java-розробки.

2.4 ORM Hibernate [8-10]

Hibernate - це фреймворк для роботи з реляційними базами даних в середовищі Java. Він надає зручний і високорівневий спосіб взаємодії з базами даних, використовуючи об'єктно-реляційну картографію (ORM). Основна ідея Hibernate - це уникнення прямого використання SQL-запитів та взаємодії з базою даних за допомогою роботи з об'єктами високого рівня. Основні концепції та можливості Hibernate включають:

Об'єктно-реляційна картографія (ORM):

- Hibernate дозволяє визначати об'єкти в коді, які відображають структуру таблиць в базі даних. Це полегшує роботу з базою даних, оскільки ви можете взаємодіяти з об'єктами, а не зі складними SQL-запитами.

Управління сесією:

- Hibernate використовує концепцію сесій для взаємодії з базою даних. Сесія представляє собою контекст взаємодії між програмою та базою даних, а також відповідає за відстеження стану об'єктів.

HQL (Hibernate Query Language):

- Це мова запитів, подібна до SQL, але використовується для взаємодії з об'єктами, а не з таблицями бази даних. HQL дозволяє писати запити, що враховують структуру об'єктів, а не структуру таблиць.

Ліниве та жадібне завантаження даних:

- Hibernate підтримує ліниве та жадібне завантаження даних. Це означає, що ви можете визначити, коли конкретні атрибути об'єкта повинні завантажуватися з бази даних.

Кешування:

- Hibernate підтримує різні рівні кешування для поліпшення продуктивності. Це дозволяє зберігати результати запитів або об'єктів в пам'яті для подальшого використання.

Транзакційне управління:

- Hibernate автоматично керує транзакціями бази даних, дозволяючи вам визначати границі транзакцій та здійснювати їх контроль.

Підтримка плагінів та розширення:

- Hibernate має гнучку архітектуру, яка дозволяє вам використовувати плагіни та розширювати його функціональність за необхідності.

Інтеграція з Spring та іншими фреймворками:

- Hibernate добре інтегрується з іншими фреймворками, такими як Spring, що дозволяє використовувати їх разом для створення комплексних застосунків.

Плюси Hibernate:

Простота Використання:

- Відсутність необхідності вручну писати складні SQL-запити спрощує розробку та підтримку коду.

Моделювання Об'єктів:

- Ви можете моделювати об'єкти в коді без прив'язки до деталей бази даних.

Транзакційна Підтримка:

- Забезпечує управління транзакціями, що полегшує підтримку консистентності даних.

Велика Спільнота та Документація:

- Hibernate має широку спільноту розробників і хорошу документацію.

Мінуси Hibernate:

Навантаження на Пам'ять:

- ORM може призводити до додаткового навантаження на пам'ять через об'єктно-реляційну картографію.

Запити HQL:

- Для деяких розробників використання мови запитів HQL може бути менш інтуїтивним порівняно зі звичайним SQL.

Перформанс:

- На деяких завданнях, особливо при великій кількості даних, може з'явитися проблема з продуктивністю.

Навчання та Конфігурація:

- Навчання та налаштування Hibernate може вимагати деякого часу та зусиль.

2.5 PostgreSQL [6]

PostgreSQL - це об'єктно-реляційна система управління базами даних (СУБД) з відкритим вихідним кодом. Вона є однією з найпотужніших та найбільш розширюваних реляційних баз даних та надає широкий спектр функцій для ефективного управління даними. Ось деякі ключові аспекти та можливості PostgreSQL:

Відкритий вихідний код:

- PostgreSQL має відкритий вихідний код, що дозволяє користувачам переглядати, модифікувати та розповсюджувати вихідний код безкоштовно.

Розширюваність:

- PostgreSQL підтримує розширюваність за допомогою функцій, які можна написати на різних мовах програмування, включаючи PL/pgSQL, PL/Tcl, PL/Perl, PL/Python, інші мови та можливість власноруч додавати типи даних, функції та операції.

Об'єктно-реляційна модель даних:

- PostgreSQL підтримує об'єктно-реляційну модель даних, що означає, що ви можете використовувати як реляційні, так і об'єктно-орієнтовані концепції при роботі з даними.

Мови програмування та збережені процедури:

- Підтримка вбудованих мов програмування, таких як PL/pgSQL, а також можливість використання інших мов, дозволяє розробникам вбудовувати бізнес-логіку безпосередньо в базу даних.

Підтримка JSON та інших неструктурованих даних:

- PostgreSQL має вбудовану підтримку для роботи з документ-орієнтованими данними, такими як JSON, що робить його ефективним для розробки застосунків, які використовують неструктуровані дані.

Повна транзакційність та ACID властивості:

- PostgreSQL підтримує транзакції з високим рівнем ізоляції, а також відповідає принципам ACID (Atomicity, Consistency, Isolation, Durability).

Висока продуктивність та масштабованість:

- PostgreSQL надає ефективність при роботі з великими обсягами даних та може бути легко масштабованим на високо-навантажених проектах.

Розширена підтримка запитів та індексація:

- Має розширені можливості виконання складних запитів, а також підтримує різноманітні методи індексації для покращення продуктивності пошуку.

Багатофункціональні розширення:

- Є багато розширень та доповнень для PostgreSQL, таких як PostGIS для географічної інформації, hstore для роботи з невизначеними полями та інші.

Плюси PostgreSQL:

Відкритий Вихідний Код:

- Забезпечує свободу вибору, можливість модифікації та розповсюдження вихідного коду.

Розширюваність:

- Можливість розширення функціональності системи за допомогою власних функцій та типів даних.

Об'єктно-Реляційна Модель:

- Підтримка об'єктно-реляційної моделі, що дозволяє працювати як з реляційними, так і з об'єктно-орієнтованими даними.

Масштабованість та Продуктивність:

- Висока продуктивність та можливість масштабування дозволяють використовувати PostgreSQL в різних областях.

Мінуси PostgreSQL:

Складність Установки та Налаштувань:

- У порівнянні з деякими іншими СУБД, установка та налаштування PostgreSQL може вимагати додаткових кроків.

Менша Розповсюдженість:

- У порівнянні з деякими комерційними рішеннями, PostgreSQL може мати меншу розповсюдженість у певних секторах ринку.

Оптимізація Для Запитів:

- Хоча PostgreSQL має високий рівень оптимізації запитів, деякі інші СУБД можуть бути ефективнішими в конкретних випадках.

2.6 Flyway [6, 13]

Flyway - це відкрите програмне забезпечення для контролю версій баз даних, яке дозволяє автоматизувати процес міграції схеми бази даних. Flyway дозволяє розробникам ефективно управляти структурою бази

даних протягом часу, додаючи та змінюючи об'єкти бази даних, такі як таблиці, стовпці, ключі та інші, в контрольований та зручний спосіб.

Основні концепції та функціональність Flyway:

Міграції:

- Основним поняттям в Flyway є "міграції". Це скрипти SQL або скрипти мови програмування, які описують зміни в базі даних. Міграції використовуються для введення нового функціоналу, модифікації структури або видалення об'єктів бази даних.

Версіонування:

- Кожна міграція має унікальний номер версії та опис. Flyway зберігає історію виконаних міграцій в спеціальній таблиці в базі даних, що дозволяє відстежувати, які міграції вже застосовані.

Базова Міграція:

- Перша міграція, яка встановлює початкову структуру бази даних, називається "базовою міграцією". Вона застосовується при створенні нової бази даних або встановленні Flyway на вже існуючу базу даних.

Автоматичне Визначення Міграцій:

- Flyway автоматично визначає, які міграції потрібно застосувати до бази даних на основі історії та наявних міграцій.

Підтримка Різних Джерел Міграцій:

- Flyway підтримує різні джерела для міграцій, включаючи SQL-скрипти, скрипти мов програмування (наприклад, Java або Groovy), а також віддалені ресурси, такі як URL або архіви.

Інтеграція з Збірками та Інструментами CI/CD:

- Flyway може бути легко інтегрованою в процеси збірки (build) та постачання (deployment) за допомогою різних інструментів CI/CD, таких як Jenkins, GitLab CI, чи інші.

Підтримка Різних Баз Даних:

- Flyway підтримує багато різних систем управління базами даних, включаючи PostgreSQL, MySQL, Oracle, Microsoft SQL Server, та інші.

Плюси Flyway:

Простота та Зручність Використання:

- Flyway пропонує простий та легкий у використанні спосіб управління міграціями бази даних.

Версіонування та Відстеження Історії:

- Забезпечує версіонування та детальне відстеження історії застосованих міграцій.

Автоматичне Визначення Змін:

- Автоматично визначає, які міграції повинні бути застосовані до бази даних.

Гнучкість та Розширюваність:

- Flyway гнучка та розширюється, дозволяючи використовувати різні джерела та типи міграцій.

Підтримка Різних Баз Даних:

- Flyway підтримує багато різних систем управління базами даних, що дозволяє використовувати його в різних проектах.

Мінуси Flyway:

Немає Вбудованих Механізмів ORM:

- Flyway не надає вбудованих механізмів ORM (на відміну від Hibernate), і тому ви можете використовувати його в поєднанні з іншими інструментами або ORM-фреймворками.

Специфікація Змін вручну:

- Деякі зміни в базі даних може доводитися специфікувати вручну в SQL або в скриптах мов програмування, що може вимагати додаткових зусиль від розробників.

Потребує Доступу до Бази Даних:

- Для використання Flyway, необхідно мати доступ до бази даних, що може бути викликаною проблемою у деяких випадках, наприклад, при роботі з хостинговими середовищами.

2.7 Docker [7, 11]

Docker - це платформа для розробки, доставки та виконання застосунків в контейнерах. Контейнер - це легкий та портативний пакунок програмного забезпечення, який включає в себе всі необхідні елементи для виконання: код, середовище виконання, бібліотеки, системні інструменти та інші залежності. Docker дозволяє ізолювати застосунки в контейнерах, що спрощує розгортання та управління додатками

Основні концепції та можливості Docker:

Контейнеризація:

- Docker використовує технологію контейнеризації для упаковки та поширення застосунків. Кожен контейнер запускається в ізольованому середовищі, що дозволяє їм працювати консистентно незалежно від конфігурацій хост-системи.

Docker Image:

- Зображення Docker - це легкий та переносимий пакунок, який містить все необхідне для запуску програми, включаючи код, середовище виконання, бібліотеки та інші залежності.

Docker Container:

- Контейнер - це екземпляр зображення, який запускається в ізольованому середовищі. Контейнер має свою власну файлову систему та мережеві ізоляції.

Dockerfile:

- Dockerfile - це текстовий файл, який містить інструкції для створення Docker-зображення. Він визначає, як буде побудовано та налаштовано зображення.

Docker Hub:

- Docker Hub - це реєстр образів Docker, де розробники можуть публікувати та обмінюватися своїми зображеннями. Ви також можете використовувати зображення, опубліковані іншими користувачами.

Docker Compose:

- Docker Compose - це інструмент для визначення та управління багатоконтейнерними застосунками. Він дозволяє визначити всю конфігурацію застосунку в одному файлі.

Мережі та Збірки:

- Docker надає можливість створювати власні мережі для контейнерів та використовувати їх для забезпечення ізоляції та зручності спілкування. Docker також дозволяє автоматизувати збірку застосунків через інтеграцію з CI/CD інструментами.

Системи Оркестрації:

- Docker може бути інтегрований з системами оркестрації, такими як Kubernetes або Docker Swarm, для управління та автоматизації роботи контейнерів в розподіленому середовищі.

Плюси Docker:

Портативність:

- Контейнери Docker можуть бути запущені на будь-якій системі, що підтримує Docker, незалежно від конфігурації хост-системи.

Легкість та Швидкість Розгортання:

- Зображення Docker легкі та швидко запускаються, що робить процес розгортання та масштабування додатків швидким та ефективним.

Ізоляція та Безпека:

- Контейнери ізольовані один від одного, що забезпечує безпеку та стабільність застосунків.

Спрощення Управління Залежностями:

- Docker дозволяє визначити та управляти всіма залежностями застосунку в одному місці.

Масштабованість та Оркестрація:

- Легко масштабувати застосунки та використовувати системи оркестрації для автоматизації управління контейнерами в розподіленому середовищі.

Мінуси Docker:

Використання Ресурсів:

- Docker може використовувати значну кількість ресурсів, особливо при великій кількості контейнерів.

Комплексність Налаштувань:

- Деякі налаштування та конфігурації можуть бути складними, особливо для новачків.

Необхідність Ядра Linux для Linux-Контейнерів:

- Для використання Linux-контейнерів потрібне ядро Linux, що може бути обмеженням на деяких платформах.

Можливість Конфліктів Мережі:

- При використанні багатьох контейнерів може виникнути проблеми з конфліктами мережі та портами.

Збірка Образів:

- Збірка зображень може вимагати додаткового часу та ресурсів на початковому етапі розробки.

2.8 Середовище розробки IntelliJ IDEA [7, 13]

IntelliJ IDEA - це інтегроване середовище для розробки (IDE), спеціально призначене для мов програмування Java, але також підтримує ряд інших мов, таких як Kotlin, Groovy, Scala, JavaScript, і багато інших. Розроблена компанією JetBrains, IntelliJ IDEA надає розширений набір інструментів для покращення продуктивності розробників та забезпечення зручності при роботі з проектами.

Основні функції та можливості IntelliJ IDEA:

Редактор Коду:

- Має потужний та інтелектуальний редактор коду з високопродуктивною системою автодоповнення, підтримкою рефакторингу та багатьма іншими функціями для полегшення написання та редагування коду.

Аналіз Коду:

- Забезпечує вбудований аналіз коду, підсвічуючи потенційні помилки та надаючи рекомендації щодо вдосконалення коду за допомогою інспекцій.

Система Керування Проектами:

- Підтримує різні системи керування версіями, такі як Git, SVN, Mercurial, та інші. Забезпечує легке інтегрування з популярними платформами для хостингу коду.

Підтримка Мов та Фреймворків:

- IntelliJ IDEA підтримує не тільки Java, але і ряд інших мов програмування, включаючи Kotlin, Groovy, Scala, JavaScript,

TypeScript та інші. Він також інтегрується з різними фреймворками, такими як Spring, Hibernate, Android SDK, і багато інших.

Вбудовані Інструменти для Розробки Web:

- Надає вбудовані інструменти для розробки веб-додатків, включаючи підтримку HTML, CSS, JavaScript, і вбудований сервер для тестування та налагодження.

Система Плагінів:

- Інтеграція з системою плагінів, що дозволяє розширити функціональність IDE, додавши нові інструменти та можливості за допомогою власних або сторонніх розширень.

Вбудовані Інструменти для Тестування:

- Підтримка вбудованих інструментів для тестування, включаючи JUnit, TestNG, і інші. Забезпечує зручний інтерфейс для виконання та аналізу тестів.

Система Автоматизації Збірки:

- Інтеграція з системами автоматизації збірки, такими як Maven, Gradle, та Ant, що дозволяє легко керувати процесами збірки та розгортання проектів.

Розуміння Контексту:

- Забезпечує розуміння контексту розробки та інтеграцію з іншими інструментами JetBrains, такими як ReSharper для .NET розробки.

Інтеграція з Серверами Додатків:

- Підтримка інтеграції з різними серверами додатків, такими як Tomcat, JBoss, і інші, для зручного розгортання та тестування додатків.

Плюси IntelliJ IDEA:

Висока Продуктивність Розробки:

- Забезпечує потужний та продуктивний інструментарій для розробки, що дозволяє розробникам швидше створювати високоякісний код.

Інтелектуальне Автодоповнення та Аналіз Коду:

- Має інтелектуальне автодоповнення та аналіз коду, що підвищує якість та стабільність кодової бази.

Підтримка Багатьох Мов та Фреймворків:

- Підтримує широкий спектр мов та фреймворків, що дозволяє розробникам працювати з різноманітними технологіями.

Зручна Інтеграція з Іншими Інструментами JetBrains:

- Інтегрується з іншими продуктами JetBrains, що полегшує роботу розробників, які використовують інші мови програмування.

Активна Спільнота та Підтримка Плагінів:

- Має активну спільноту користувачів та розробників, а також велику кількість плагінів для розширення функціональності.

Мінуси IntelliJ IDEA:

Потребує Ресурсів Системи:

- IntelliJ IDEA може вимагати значних ресурсів системи, особливо при великих проектах.

Вартість:

- Хоча є безкоштовна версія Community Edition, повна версія IntelliJ IDEA Ultimate є платною.

Навчання:

- Для новачків може знадобитися деякий час для ознайомлення та засвоєння всіх можливостей та інструментів IDE.

2.9 Angular [16]

Angular - це платформа та фреймворк для створення веб-додатків. Розроблений і підтримуваний компанією Google, Angular дозволяє розробникам створювати потужні та масштабовані односторінкові додатки (Single Page Applications, SPA) з використанням TypeScript. Нижче наведено обширний огляд основних концепцій та можливостей Angular.

Основні концепції та можливості Angular:

Компоненти:

- Angular використовує концепцію компонентів для побудови користувацького інтерфейсу. Кожен компонент об'єднує HTML-код, логіку та стилі, і може бути використаний як будівельний блок для створення ієрархії додатків.

Шаблони та Директиви:

- Angular використовує шаблони для визначення вигляду компонентів та директив для вставки динамічного поведінки в HTML. Директиви також використовуються для маніпуляції DOM та створення реюзабельних компонентів.

Сервіси та Залежності:

- Сервіси в Angular використовуються для організації та обміну загальною логікою та функціональністю між компонентами. Залежності внедрюються для забезпечення компонентам доступу до сервісів.

Реактивне Програмування:

- Angular підтримує реактивне програмування за допомогою RxJS (Reactive Extensions for JavaScript), що дозволяє працювати з асинхронними подіями та данними.

Маршрутизація:

- Angular має вбудований механізм маршрутизації для управління навігацією в односторінкових додатках та надання структури та організації додатку.

Форми:

- Angular надає потужні засоби для роботи з формами, включаючи валідацію, відстеження стану, інтеграцію з директивами та інше.

Інтерсептори та Перехоплення Запитів:

- Angular дозволяє використовувати інтерсептори для перехоплення та обробки HTTP-запитів, що надає можливість додатково обробляти дані до відправки чи після отримання.

Модулі:

- Додатки Angular організовані у модулі, які визначають контекст та область видимості компонентів та сервісів.

Зібрання та Розгортання:

- Angular надає інструменти для зібрання та оптимізації коду перед розгортанням. Можна генерувати продакшн-готовий код для підвищення продуктивності додатка.

Тестування:

- Angular сприяє тестуванню застосунків, надаючи інструменти для одиниць, інтеграційних та e2e (end-to-end) тестів.

Плюси Angular:

Масштабованість:

- Angular підходить для розробки великих та складних додатків, забезпечуючи потужні інструменти для організації та структурування коду.

Строга Система Типізації:

- Використання TypeScript дозволяє створювати надійний та читабельний код завдяки статичній типізації.

Велика Активна Спільнота:

- Angular має велику та активну спільноту розробників, що сприяє обміну досвідом та вирішенню проблем.

Вбудовані Механізми:

- Angular має вбудовані рішення для багатьох аспектів розробки, таких як маршрутизація, HTTP-запити, форми та інше.

Розвиток та Підтримка Google:

- Розробка та підтримка від компанії Google надає впевненість в стабільності та актуальності фреймворка.

Мінуси Angular:

Великий Об'єм Коду:

- Проекти на Angular можуть мати значний об'єм коду, що може впливати на продуктивність та завантаження додатків.

Строга Крива Навчання:

- Angular має строгу криву навчання, особливо для розробників, які тільки починають вивчати фреймворк.

Затримки у Випусках:

- Іноді випуски нових версій Angular можуть затримуватися, що може впливати на використання новітніх можливостей.

2.10 Система контролю версій Git [13, 14]

Git - це система керування версіями (VCS), яка дозволяє відстежувати зміни в програмному коді та координувати роботу розробників над спільним проектом. Розроблена Лінусом Торвальдсом у 2005 році, Git швидко став однією з найпопулярніших систем керування версіями через

свою ефективність та розподілену природу. Нижче подано обширний огляд концепцій та можливостей Git.

Основні концепції та можливості Git:

Репозиторій:

- Репозиторій в Git - це сховище для вашого проекту. Може бути локальним (на вашому комп'ютері) або віддаленим (на сервері).

Коміт:

- Коміт представляє собою зміни в коді, які ви зберігаєте в репозиторії. Він включає в себе змінені, додані або видалені файли, а також повідомлення про коміт, що описує зміни.

Бранчі:

- Бранчі дозволяють вам відокремити робочу копію коду для розвитку нової функціональності чи виправлення помилок без впливу на основну гілку.

Злиття (Merge) та Ребейз (Rebase):

- Злиття об'єднує зміни з одного бранчу в інший. Ребейз також дозволяє об'єднувати зміни, але це робить історію комітів більш чистою та легшою для розуміння.

Віддалені Репозиторії:

- Git підтримує розподілені репозиторії, що дозволяє кільком розробникам працювати з одним та тим же проектом та взаємодіяти через віддалені репозиторії.

Відгалуження та Злиття (Branching and Merging):

- Git дозволяє створювати гілки для роботи над конкретними функціональностями чи виправленнями. Злиття потім дозволяє об'єднувати ці гілки.

Історія Комітів:

- Git веде докладну історію комітів, що дозволяє переглядати та відстежувати зміни протягом часу.

Ігнорування Файлів:

- Git дозволяє створювати файл .gitignore**,** в якому перераховуються файли та теки, які не потрібно включати в репозиторій (наприклад, файли з налаштуваннями, файли, створені системою тощо).

Вітки Віддалених Репозиторіїв:

- За допомогою віток віддалених репозиторіїв, таких як origin/master, розробники можуть взаємодіяти з віддаленим репозиторієм та синхронізувати свій локальний код.

Стеження Змін та Відновлення:

- Git дозволяє вам відстежувати та зберігати зміни, а також відновлювати код на певному етапі в минулому.

Плюси Git:

Розподілена Система Керування Версіями:

- Git є розподіленою системою керування версіями, що дозволяє робити коміти та працювати навіть без підключення до мережі.

Швидкість та Ефективність:

- Git є дуже швидким та ефективним, завдяки чому великі проекти легко керуються та великі обсяги даних швидко обробляються.

Гнучкість та Відкритий Загальний Інтерфейс (API):

- Git дозволяє розробникам використовувати різні стратегії роботи, надає відкритий API для розширень та інтеграції з іншими інструментами.

Велика Спільнота та Підтримка:

- Git має велику та активну спільноту, яка надає підтримку та вирішує проблеми.

Гітхаб та Інші Платформи Спільного Робочого Простору:

- Git інтегрований з популярними хмарними платформами, такими як GitHub, GitLab, Bitbucket, що полегшує спільну роботу над проектами.

Стеження та Відновлення:

- Можливість стежити за змінами та відновлювати код на певних етапах дозволяє легко управляти історією розробки.

Мінуси Git:

Складність та Навчання:

- Git може бути складним для новачків, особливо коли йдеться про роботу з гілками та ребейзом.

Можливість Конфліктів При Злитті:

- При злитті гілок можуть виникати конфлікти, які потрібно вирішувати вручну.

Історія Комітів Може Бути Складною:

- У великих проектах історія комітів може стати складною та заплутаною.

Потребує Додаткового Програмного Забезпечення:

- Для повноцінної роботи Git, може знадобитися додаткове програмне забезпечення для відображення графічного інтерфейсу та інших зручностей.

РОЗДІЛ 3. РОЗРОБКА ВЕБ-СЕРВІСУ

3.1 Загальний опис системи

Для виконання вимог в системі реалізовано:

Spring Security, який на першому етапі перевіряє що за введеними логіном та паролем існує користувач в системі. На другому кроці перевіряється роль користувача в системі.

В системі може бути дві ролі:

1. Admin - власник підприємства або поручена особа
2. User - працівник

Власник: після того як власник увійшов в систему він потрапляє на сторінку аналітики всього підприємства, вверху веб-сайту знаходиться хедер із назвою веб-сайту та меню в якому є опція вийти з системи. Зліва знаходиться меню з наступними сторінками:

- Аналітика - відображення кількості продукції та коштів на кожному з пунктів прийому.
- Пункти прийому -. відображення пунктів прийому, кожна з них з них доступна для редагування та кнопка “Створити”.
- Працівники - відображення всіх працівників підприємства та пункти прийому до яких вони прикріплені.

Працівник: після того як працівник увійшов в систему він потрапляє на сторінку аналітики пункту прийому до якого він прикріплений, вверху веб-сайту знаходиться хедер із назвою веб-сайту та меню в якому є опція вийти з системи. Зліва знаходиться меню з наступними сторінками:

- Аналітика - відображення кількості продукції та коштів на своєму пункті прийому.
- Працівники - відображення всіх працівників, які відносяться до того самого пункту прийому, що і працівник.

- Баланс - відображення історії зміни балансу. Та кнопка “Прихід”, яка дає можливість здійснити прихід продукції.

3.2 Проектування та створення бази даних

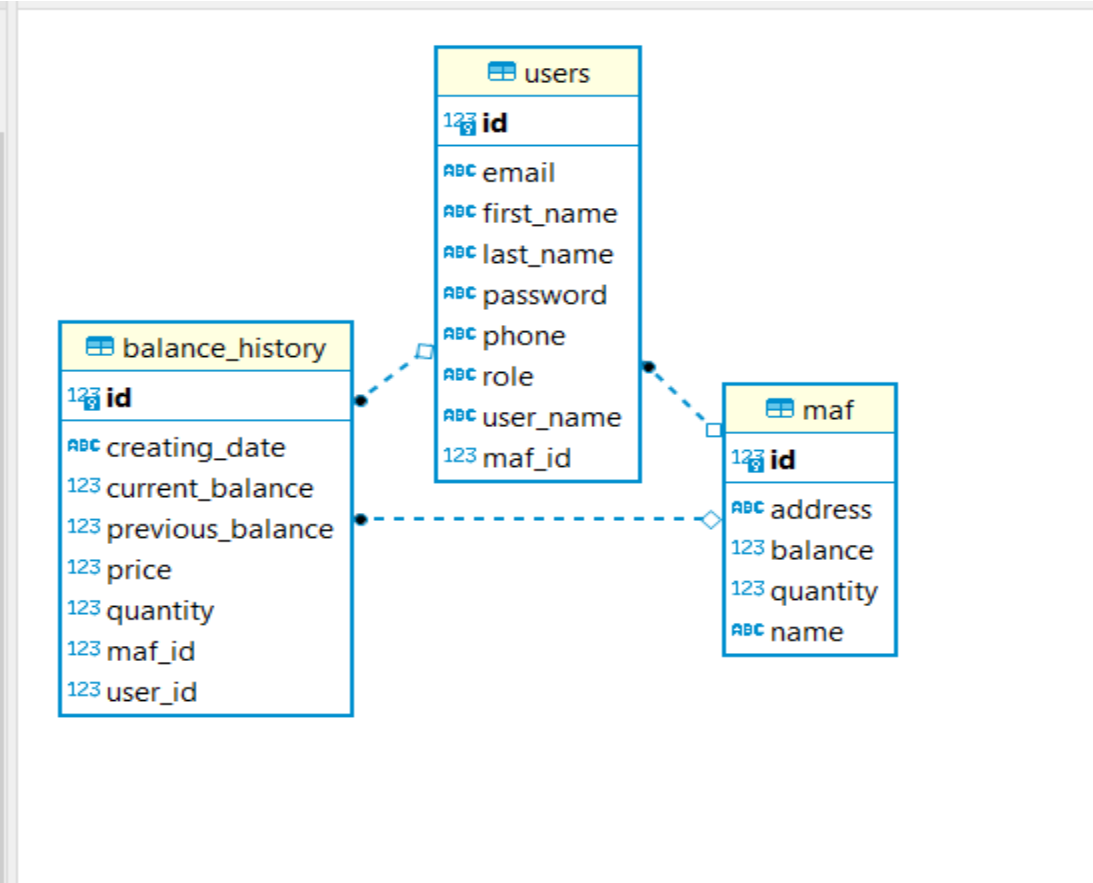
База даних є найважливішим елементом будь якої інформаційної системи. Сервером бази даних для веб-додатку обрана система керування базами даних – Postgresql.

Перерахуємо сутності (Entity):

- users - сутність, яка визначає користувачів системи, включає в себе такі поля:
  - id - унікальний ідентифікатор
  - email - електронна адреса користувача
  - first_name - ім’я користувача
  - last_name - прізвище
  - password - пароль до облікового запису користувача
  - phone - номер мобільного користувача
  - role - вказує на роль користувача в системі
  - user_name - логін для входу в обліковий запис
  - maf_id - зовнішній ключ для зв’язку з таблицею maf
- maf - сутність, яка представляє пункти прийому, включає в себе такі поля:
  - id - унікальний ідентифікатор
  - name - назва пункту прийому
  - address - адреса
  - quantity - кількість продукції, яка знаходиться на пункті
  - balance - баланс коштів на пункті

- balance_history - сутність, яка представляє історію зроблених приходів працівником
    - id - унікальний ідентифікатор
    - creating_date - дата приходу
    - current_balance - баланс пункту прийому після здійснення приходу
    - previous_balance - баланс пункту прийому до здійснення приходу
    - price - ціна по якій був здійснений прихід
    - quantity - кількість продукції у приході
    - maf_id - зовнішній ключ для зв’язку з таблицею maf
    - user_id - зовнішній ключ для зв’язку з таблицею users

Схема бази даних



3.3 Оголошення сутностей Hibernate [3, 8, 9]

Для зручної і швидкої роботи з базою даних, було вирішено використовувати Hibernate. Hibernate дозволяє нам відображати таблиці у вигляді моделей (Entity) та працювати з ними, без використання sql запитів напряму.

У даному проєкті є такі моделі:

- **User**

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

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    @Column(unique = true)
    private String userName;
    @Column(unique = true)
    private String phone;
    @Column(unique = true)
    private String email;
    private String password;
    @Enumerated(EnumType.STRING)
    private Role role;
    @JsonBackReference
    @ManyToOne
    @JoinColumn(name = "maf_id")
    private Maf maf;

}
```

- **Maf (**пункт прийому**)**

```
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
@ToString
public class Maf {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String address;
    @Column(unique = true)
    private String name;
    private BigDecimal balance;
    @Positive
    private BigDecimal quantity;
    @OneToMany(mappedBy = "maf")
    @JsonBackReference
    private Set<User> users;
}
```

- **BalanceHistory**

```
33 usages
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
public class BalanceHistory {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private BigDecimal quantity;
    private BigDecimal price;
    private BigDecimal previousBalance;
    private BigDecimal currentBalance;
    private String creatingDate = LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm"));
    @ManyToOne
    @JoinColumn(name="user_id")
    @JsonBackReference
    private User user;
    @ManyToOne
    @JoinColumn(name="maf_id")
    @JsonBackReference
    private Maf maf;
}
```

При створенні моделей були використані анотації для представлення метаданих:

1. **Entity -** вказує спрінгу, що цей клас відображає сутність в базі даних
2. **Table -** вказує ім’я таблиці
3. **Id -** вказує на те, що дане поле є унікальним ідентифікатором
4. **GeneratedValue -** стратегія генерування унікального ідентифікатора
5. **Column -** вказує ім’я стовпця в таблиці, та дозволяє вказувати додаткові атрибути поля
6. **DateTimeFormat -** формат часу
7. **Enumerated -** зберігає енам в базі даних
8. **JoinColumn -** вказує на поле, по якому таблиці реляційно зв’язані
9. **OneToOne, ManyToOne, OneToMany -** вказують на тип реляційного зв’язку

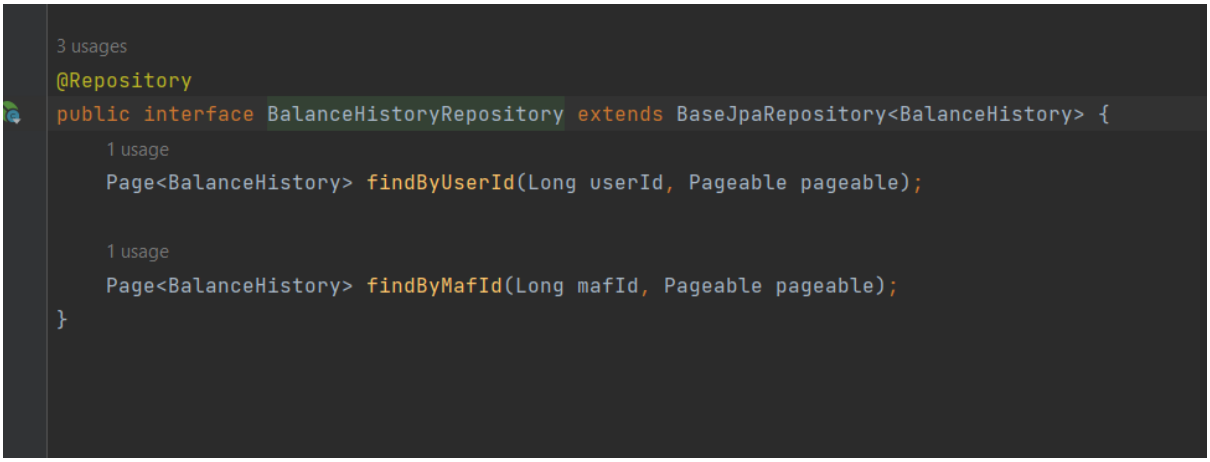
3.4 Створення репозиторіїв

Рівень абстракції репозиторію визначається для ефективного відокремлення функцій, пов'язаних з базою даних. Spring Data спрощує формування запитів, дозволяючи використовувати camelCase для автоматичного визначення запитів за назвами методів. Для більш специфічних випадків можна використовувати анотацію @Query для точного визначення запитів.

- **Репозиторій User-a**

- **Репозиторій Maf-a**

- **Репозиторій BalanceHistory**



```
3 usages
@Repository
public interface BalanceHistoryRepository extends BaseJpaRepository<BalanceHistory> {
    1 usage
    Page<BalanceHistory> findByUserId(Long userId, Pageable pageable);

    1 usage
    Page<BalanceHistory> findByMafId(Long mafId, Pageable pageable);
}
```

3.5 Створення сервісів

Рівень сервісів використовується для забезпечення взаємодії між рівнем репозиторію та контролерами. Він відповідає за отримання даних з репозиторію та виконання основної бізнес-логіки проекту. Сервіс також форматує дані у вигляд, необхідний для контролерів, забезпечуючи ефективну передачу інформації між різними рівнями додатку.

- **Сервіс UserServiceUmpl**

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

    10 usages
    private UserRepository repository;
    3 usages
    private MafService mafService;

    2 usages
    private AuthService authService;
    2 usages
    private JwtTokenProvider jwtTokenProvider;
    2 usages
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    public UserServiceImpl(UserRepository repository, BCryptPasswordEncoder bCryptPasswordEncoder,
                           MafService mafService, AuthService authService, JwtTokenProvider jwtTokenProvider) {
        this.repository = repository;
        this.mafService = mafService;
        this.authService = authService;
        this.jwtTokenProvider = jwtTokenProvider;
        this.bCryptPasswordEncoder = bCryptPasswordEncoder;
    }
```

- **Методи UserService**

```
1 usages  1 implementation
public interface UserService  {
    1 implementation
    List<UserDto> getAll();

    1 implementation
    User findById(Long id);

    1 implementation
    User create(UserDto item);

    1 implementation
    void delete(Long id);

    1 usage  1 implementation
    User updateUser(Long id, UserUpdateDto userUpdateDto);

    1 usage  1 implementation
    TokenDto login(AuthDto authDtoa);

    1 usage  1 implementation
    Page<User> getAllPages(int page, int pageSize, String sort, Long mafId);
}
```

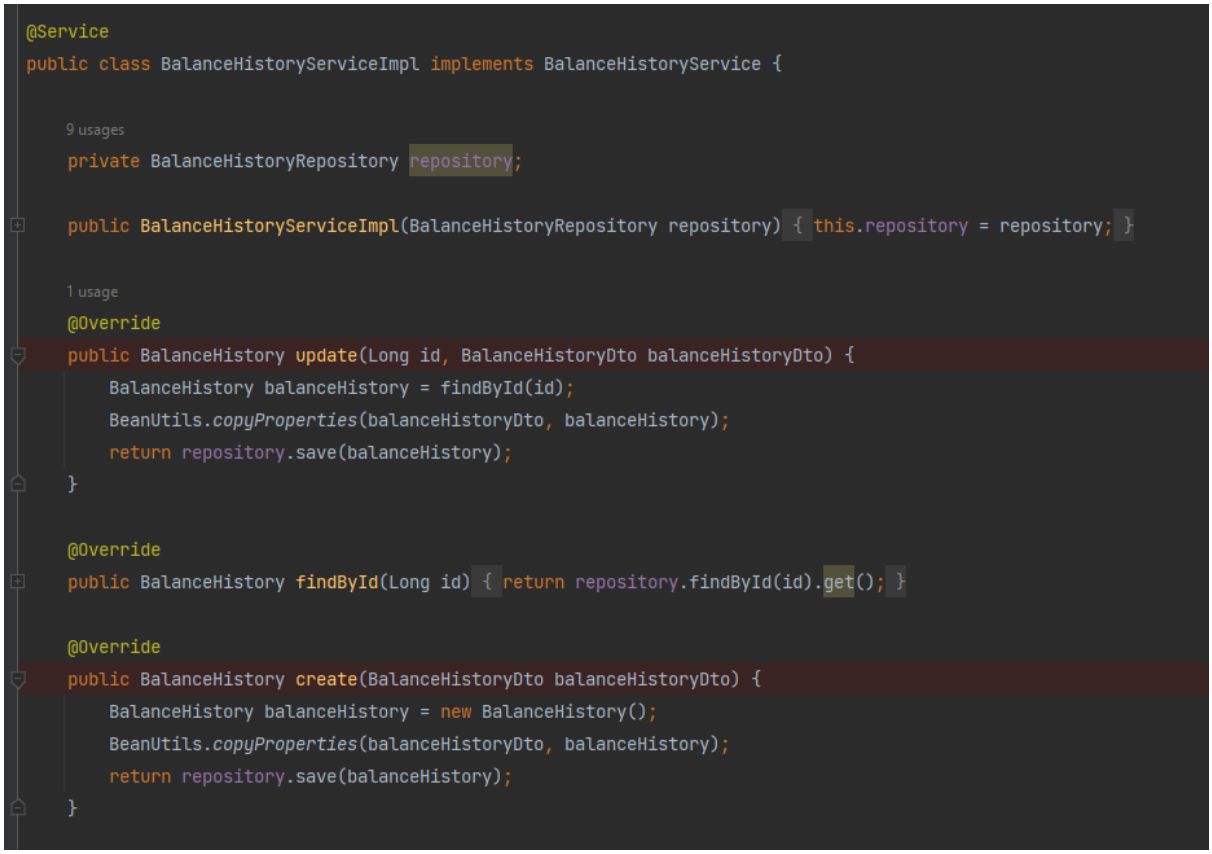
- **Сервіс MafServiceImpl**

```
@Service
public class MafServiceImpl implements MafService {

    8 usages
    private MafRepository repository;
    2 usages
    private UserRepository userRepository;

    2 usages
    private TwilioService twilioService;

    public MafServiceImpl(MafRepository repository, UserRepository userRepository, TwilioService twilioService
    ) {
        this.repository = repository;
        this.userRepository = userRepository;
        this.twilioService = twilioService;
    }
```

- **Методи MafService**

```
15 usages   1 implementation
public interface MafService {

    1 implementation
    Maf create(MafDto mafDto);

    1 usage   1 implementation
    Maf update(Long id, MafDto mafDto);

    1 implementation
    Maf findById(Long id);

    1 implementation
    void delete(Long id);

    1 usage   1 implementation
    Page<Maf> getAllMafs(int page, int pageSize, String sort);

    1 usage   1 implementation
    void update(Maf maf);

    1 usage   1 implementation
    List<Maf> getAll();
}
```

- **Сервіс BalanceHistoryServiceImpl**



```java
@Service
public class BalanceHistoryServiceImpl implements BalanceHistoryService {

    9 usages
    private BalanceHistoryRepository repository;

    public BalanceHistoryServiceImpl(BalanceHistoryRepository repository) { this.repository = repository; }

    1 usage
    @Override
    public BalanceHistory update(Long id, BalanceHistoryDto balanceHistoryDto) {
        BalanceHistory balanceHistory = findById(id);
        BeanUtils.copyProperties(balanceHistoryDto, balanceHistory);
        return repository.save(balanceHistory);
    }

    @Override
    public BalanceHistory findById(Long id) { return repository.findById(id).get(); }

    @Override
    public BalanceHistory create(BalanceHistoryDto balanceHistoryDto) {
        BalanceHistory balanceHistory = new BalanceHistory();
        BeanUtils.copyProperties(balanceHistoryDto, balanceHistory);
        return repository.save(balanceHistory);
    }
```

- **Методи BalanceHistory**

```java
import ...

5 usages  1 implementation
public interface BalanceHistoryService {
    1 usage  1 implementation
    BalanceHistory update(Long id, BalanceHistoryDto balanceHistoryDto);

    1 implementation
    BalanceHistory findById(Long id);

    1 implementation
    BalanceHistory create(BalanceHistoryDto balanceHistoryDto);

    1 implementation
    BalanceHistory create(BalanceHistory balanceHistory);

    1 implementation
    void delete(Long id);

    1 usage  1 implementation
    Page<BalanceHistory> getAllBalanceHistories(int page, int pageSize, String sort, Long userId, Long mafId);
}
```

3.6 Створення контролерів

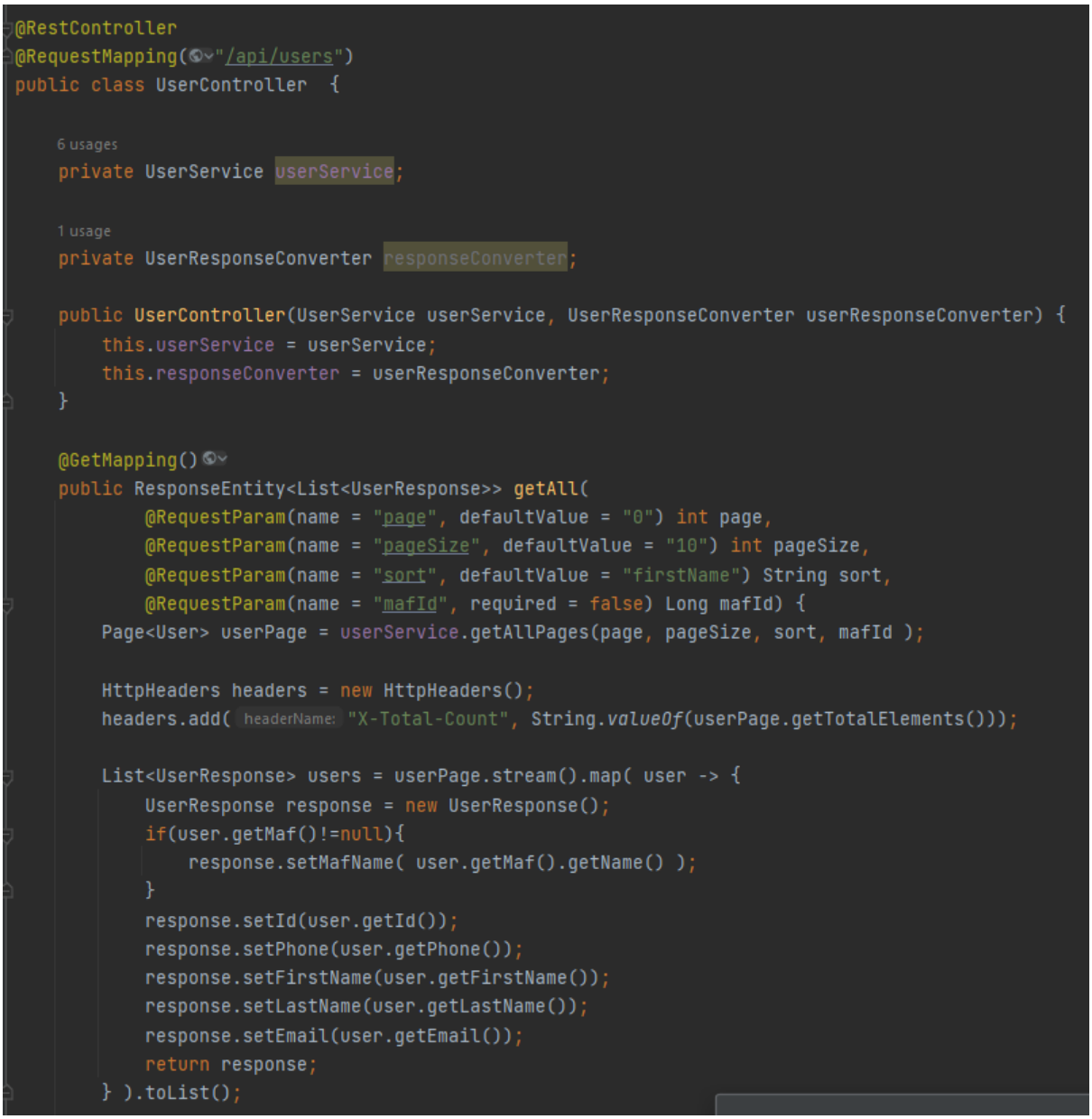
Контролери використовуються для отримання та відображення даних у графічному інтерфейсі. Вони передають моделі для ефективного відображення та відповідають на URL-запити, використовуючи методи, які відображають їхню функціональність, завдяки механізму Spring MVC.

В проєкті реалізовано наступні контролери та їх основні методи:

- **AuthenticationController -** контролер відповідає за автентифікацію та авторизацію користувачів в системі.

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

    2 usages
    private UserService userService;
    2 usages
    private AuthService authService;

    public AuthenticationController(UserService userService, AuthService authService) {
        this.userService = userService;
        this.authService = authService;
    }

    @PostMapping("/login")
    public TokenDto login(@RequestBody AuthDto authDto) { return userService.login(authDto); }

    @GetMapping("/current-user")
    public UserDto getCurrentUser() {
        User user = authService.getUserFromContext();
        UserDto userDto = new UserDto();
        BeanUtils.copyProperties(user, userDto);
        if (user.getMaf() != null) {
            userDto.setMafId(user.getMaf().getId());
        }
        return userDto;
    }
}
```

- **UserController** - контролер, який відповідає за операції над користувачами.



```
@RestController
@RequestMapping("/api/users")
public class UserController  {

    6 usages
    private UserService userService;

    1 usage
    private UserResponseConverter responseConverter;

    public UserController(UserService userService, UserResponseConverter userResponseConverter) {
        this.userService = userService;
        this.responseConverter = userResponseConverter;
    }

    @GetMapping()
    public ResponseEntity<List<UserResponse>> getAll(
            @RequestParam(name = "page", defaultValue = "0") int page,
            @RequestParam(name = "pageSize", defaultValue = "10") int pageSize,
            @RequestParam(name = "sort", defaultValue = "firstName") String sort,
            @RequestParam(name = "mafId", required = false) Long mafId) {
        Page<User> userPage = userService.getAllPages(page, pageSize, sort, mafId );

        HttpHeaders headers = new HttpHeaders();
        headers.add( headerName: "X-Total-Count", String.valueOf(userPage.getTotalElements()));

        List<UserResponse> users = userPage.stream().map( user -> {
            UserResponse response = new UserResponse();
            if(user.getMaf()!=null){
                response.setMafName( user.getMaf().getName() );
            }
            response.setId(user.getId());
            response.setPhone(user.getPhone());
            response.setFirstName(user.getFirstName());
            response.setLastName(user.getLastName());
            response.setEmail(user.getEmail());
            return response;
        } ).toList();
```

- **MafController -** контролер, який відповідає за операції над пунктами прийому.

```
@RestController
@RequestMapping(path = "/api/maf")
public class MafController {

    7 usages
    private MafService mafService;

    public MafController(MafService mafService) { this.mafService = mafService; }

    @GetMapping
    public ResponseEntity<List<Maf>> getMafs(@RequestParam(name = "page", defaultValue = "0") int page,
                                             @RequestParam(name = "pageSize", defaultValue = "10") int pageSize,
                                             @RequestParam(name = "sort", defaultValue = "balance") String sort) {
        Page<Maf> mafsPage = mafService.getAllMafs(page, pageSize, sort);

        HttpHeaders headers = new HttpHeaders();
        headers.add( headerName: "X-Total-Count", String.valueOf(mafsPage.getTotalElements()));
        List<Maf> mafs = mafsPage.stream().collect(Collectors.toList());

        return new ResponseEntity<>(mafs, headers, HttpStatus.OK);
    }
```

- **BalanceHistoryController** - контролер, який відповідає за операції історії приходів.

```java
@RestController
@RequestMapping(path = "/api/balance-history")
public class BalanceHistoryController {

    7 usages
    private BalanceHistoryService balanceHistoryService;
    2 usages
    private AuthService authService;
    2 usages
    private MafService mafService;

    public BalanceHistoryController(BalanceHistoryService balanceHistoryService, AuthService authService, MafService mafService) {
        this.balanceHistoryService = balanceHistoryService;
        this.authService = authService;
        this.mafService = mafService;
    }
```

```java
@PostMapping("/arrival")
public BalanceHistoryResponse makeArrival(@RequestBody ArrivalDto arrivalDto) {
    User user = authService.getUserFromContext();
    Maf maf = user.getMaf();

    BigDecimal previousBalance = maf.getBalance();
    BigDecimal actualBalance = maf.getBalance().subtract(arrivalDto.getQuantity().multiply(arrivalDto.getPrice()));

    BalanceHistory balanceHistory = new BalanceHistory();
    balanceHistory.setPreviousBalance(previousBalance);
    balanceHistory.setCurrentBalance(actualBalance);
    balanceHistory.setMaf(maf);
    balanceHistory.setUser(user);
    balanceHistory.setPrice(arrivalDto.getPrice());
    balanceHistory.setQuantity(arrivalDto.getQuantity());

    maf.setBalance(actualBalance);
    maf.setQuantity(maf.getQuantity().add(arrivalDto.getQuantity()));
    mafService.update(maf);

    BalanceHistoryResponse response = new BalanceHistoryResponse();
    BeanUtils.copyProperties(balanceHistory, response);
    response.setMafName(maf.getName());
    response.setUserName(user.getFirstName() == null ? " " : user.getFirstName() + " " + user.getLastName() == null ? " " : user.getLastName());
    balanceHistoryService.create(balanceHistory);
    return response;
```

При розробці контролерів було використано такі анотації:

1. **RestController** - полегшує створення класу - контролера, вказуючи spring-у його роль.
2. **RequestMapping** - вказує адрес по якому доступний данний метод.
3. **ResponseStatus** - вказує код запиту, після його обробки
4. **ResponseBody** - Автоматично записує відповідь у тіло відповіді.
5. **GetMapping, PostMapping** - вказують на HTTP метод, яким можна звернутися до даного методу.

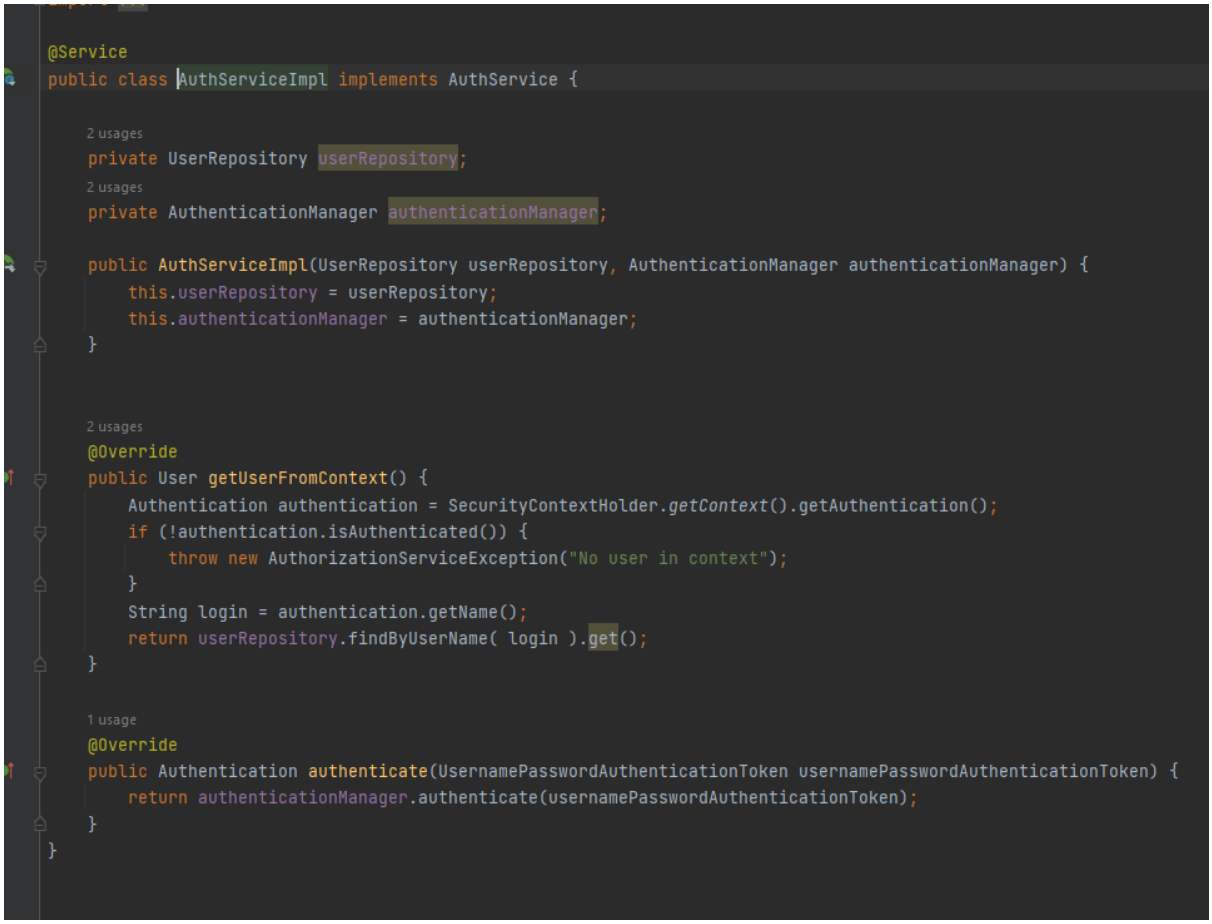
6. **RequestParam -** використовується для доступу до параметрів, в тілі URL
7. **RequestBody -** використовується для передачі тіла запиту.

3.7 **Розробка Spring Security та JWT токена** [2, 8]

Система використовує Spring Security для забезпечення безпеки, перевіряючи існування користувача за введеним номером телефона та паролем. Spring Security є Java фреймворком, який надає засоби для побудови систем автентифікації та авторизації, а також забезпечення безпеки корпоративних додатків на основі Spring Framework. Основні компоненти Spring Security включають SecurityConfiguration, який обмежує доступ до визначених енд-поінтів відповідно до ролей користувача.

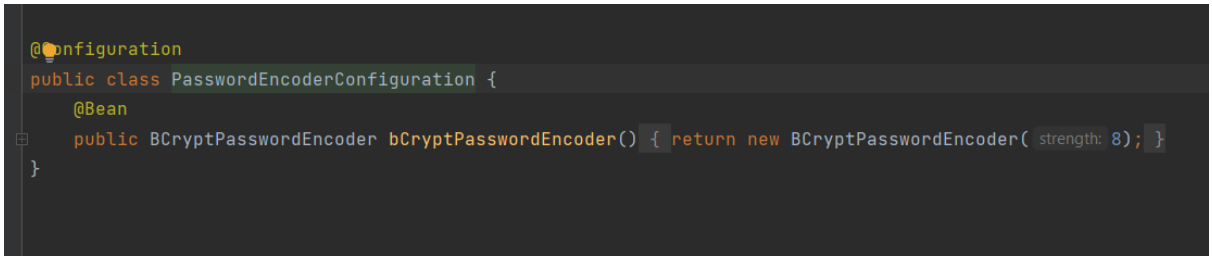
Та **SecurityContext** - де зберігається авторизований користувач.

- **AuthService**



```
@Service
public class AuthServiceImpl implements AuthService {

    2 usages
    private UserRepository userRepository;
    2 usages
    private AuthenticationManager authenticationManager;

    public AuthServiceImpl(UserRepository userRepository, AuthenticationManager authenticationManager) {
        this.userRepository = userRepository;
        this.authenticationManager = authenticationManager;
    }


    2 usages
    @Override
    public User getUserFromContext() {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        if (!authentication.isAuthenticated()) {
            throw new AuthorizationServiceException("No user in context");
        }
        String login = authentication.getName();
        return userRepository.findByUserName( login ).get();
    }

    1 usage
    @Override
    public Authentication authenticate(UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken) {
        return authenticationManager.authenticate(usernamePasswordAuthenticationToken);
    }
}
```

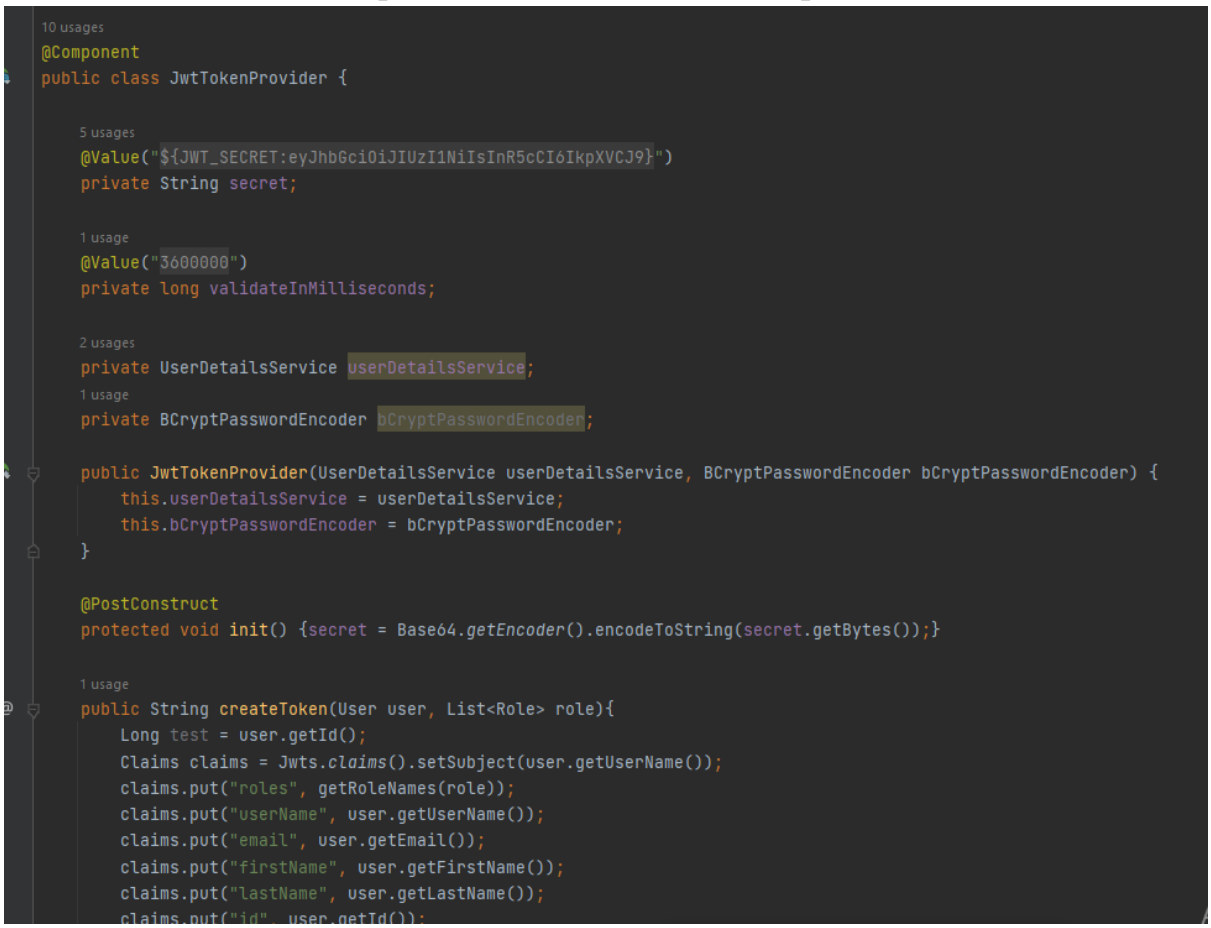
- **PasswordEncoder -** відповідає за безпечне збереження паролю користувача в базі даних



```
@Configuration
public class PasswordEncoderConfiguration {
    @Bean
    public BCryptPasswordEncoder bCryptPasswordEncoder() { return new BCryptPasswordEncoder( strength: 8); }
}
```

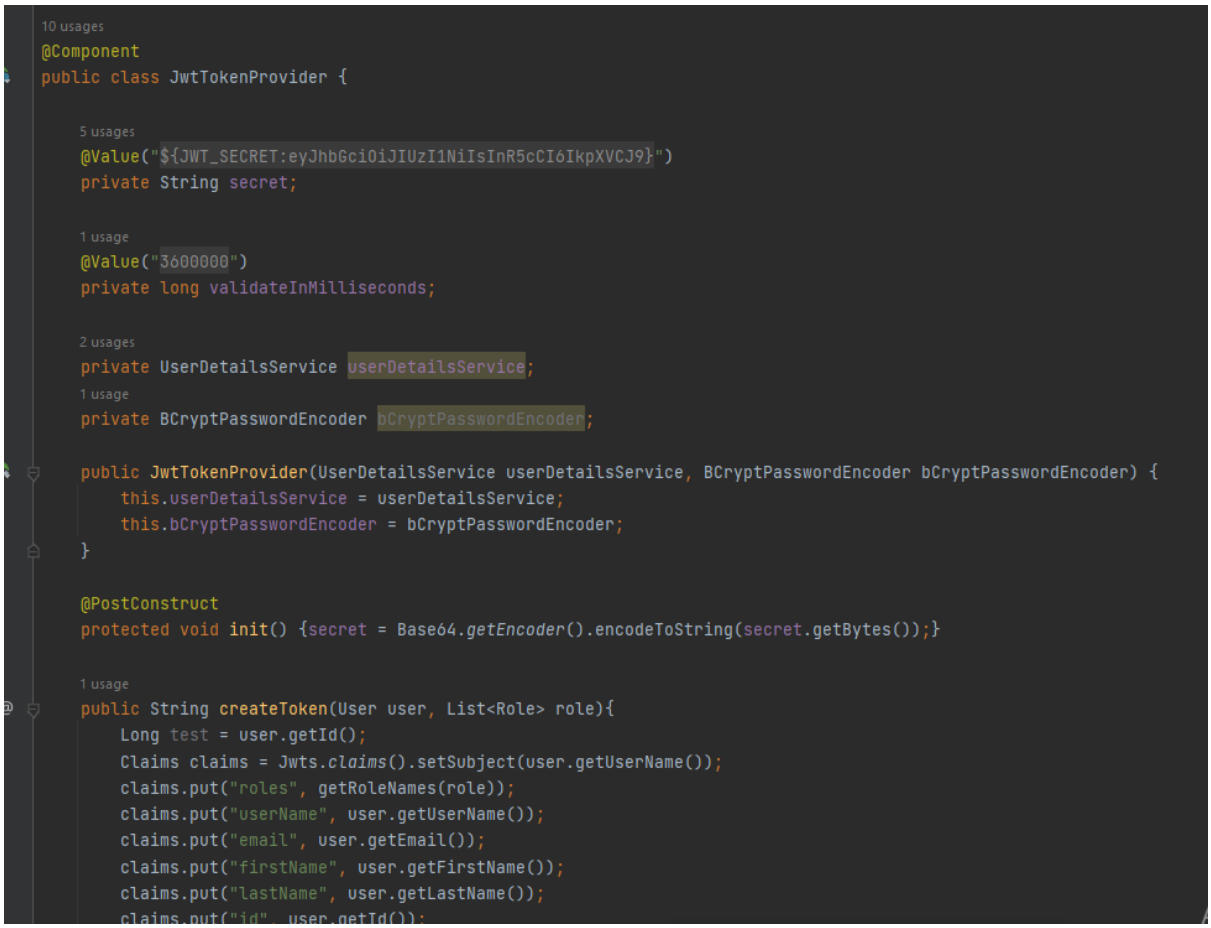
JWT (JSON Web Token) - це відкритий стандарт для створення токенів доступу, побудованих на основі JSON формату. Сервер генерує ці токени, підписує їх з використанням секретного ключа і передає клієнту. Клієнт може підтвердити свою особу та отримати доступ до захищених ресурсів сервера, використовуючи отриманий токен.

- **JwtTokenProvider -** Сервіс, що відповідає за створення Токена



```
10 usages
@Component
public class JwtTokenProvider {

    5 usages
    @Value("${JWT_SECRET:eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9}")
    private String secret;

    1 usage
    @Value("3600000")
    private long validateInMilliseconds;

    2 usages
    private UserDetailsService userDetailsService;
    1 usage
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    public JwtTokenProvider(UserDetailsService userDetailsService, BCryptPasswordEncoder bCryptPasswordEncoder) {
        this.userDetailsService = userDetailsService;
        this.bCryptPasswordEncoder = bCryptPasswordEncoder;
    }

    @PostConstruct
    protected void init() {secret = Base64.getEncoder().encodeToString(secret.getBytes());}

    1 usage
    public String createToken(User user, List<Role> role){
        Long test = user.getId();
        Claims claims = Jwts.claims().setSubject(user.getUserName());
        claims.put("roles", getRoleNames(role));
        claims.put("userName", user.getUserName());
        claims.put("email", user.getEmail());
        claims.put("firstName", user.getFirstName());
        claims.put("lastName", user.getLastName());
        claims.put("id", user.getId());
```

- **JwtTokenFilter -** Сервіс, що відповідає за обмеження доступу запитів на сервер, що не мають валідного токена



```
10 usages
@Component
public class JwtTokenProvider {

    5 usages
    @Value("${JWT_SECRET:eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9}")
    private String secret;

    1 usage
    @Value("3600000")
    private long validateInMilliseconds;

    2 usages
    private UserDetailsService userDetailsService;
    1 usage
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    public JwtTokenProvider(UserDetailsService userDetailsService, BCryptPasswordEncoder bCryptPasswordEncoder) {
        this.userDetailsService = userDetailsService;
        this.bCryptPasswordEncoder = bCryptPasswordEncoder;
    }

    @PostConstruct
    protected void init() {secret = Base64.getEncoder().encodeToString(secret.getBytes());}

    1 usage
    public String createToken(User user, List<Role> role){
        Long test = user.getId();
        Claims claims = Jwts.claims().setSubject(user.getUserName());
        claims.put("roles", getRoleNames(role));
        claims.put("userName", user.getUserName());
        claims.put("email", user.getEmail());
        claims.put("firstName", user.getFirstName());
        claims.put("lastName", user.getLastName());
        claims.put("id", user.getId());
```

3.8 Підключення TWILIO сервіса

Twilio - це платформа, призначена для створення повідомлень SMS, дзвінків та інших форм оповіщень. У проекті Twilio використовується для надсилання унікального коду користувачу після реєстрації. Користувач може використати отриманий код для активації свого облікового запису.

```
@Service
public class TwilioServiceImpl implements TwilioService {

    1 usage
    private static final String TWILIO_SID = "ACbf559e8c64b15a95a9b132d7947248db";

    1 usage
    private static final String AUTH_TOKEN = "6b9f38b87bbe341170460bf73808aa28";

    1 usage
    private static final String TWILIO_NUMBER = "+19495317234";

    1 usage
    private static final String OWNER_PHONE = "+380660888770";

    1 usage
    public void sendMessage(Maf maf){
        String message = String.format(
                "На пункті прийому: %s, за адресом %s закінчуються кошти, баланс: %s грн",
                maf.getName(),
                maf.getAddress(),
                maf.getBalance()
        );
        createMessage(OWNER_PHONE, message);
    }

    1 usage
    private void createMessage(String phone, String message) {
        Twilio.init(TWILIO_SID, AUTH_TOKEN);
        Message.creator(new PhoneNumber(phone), new PhoneNumber(TWILIO_NUMBER), message).create();
    }
}
```

4 РОЗДІЛ. РОЗРОБКА UI ЧАСТИНИ

4.1 Сторінка авторизації

Pitey

Ввійти

- HTML ресурс сторінки

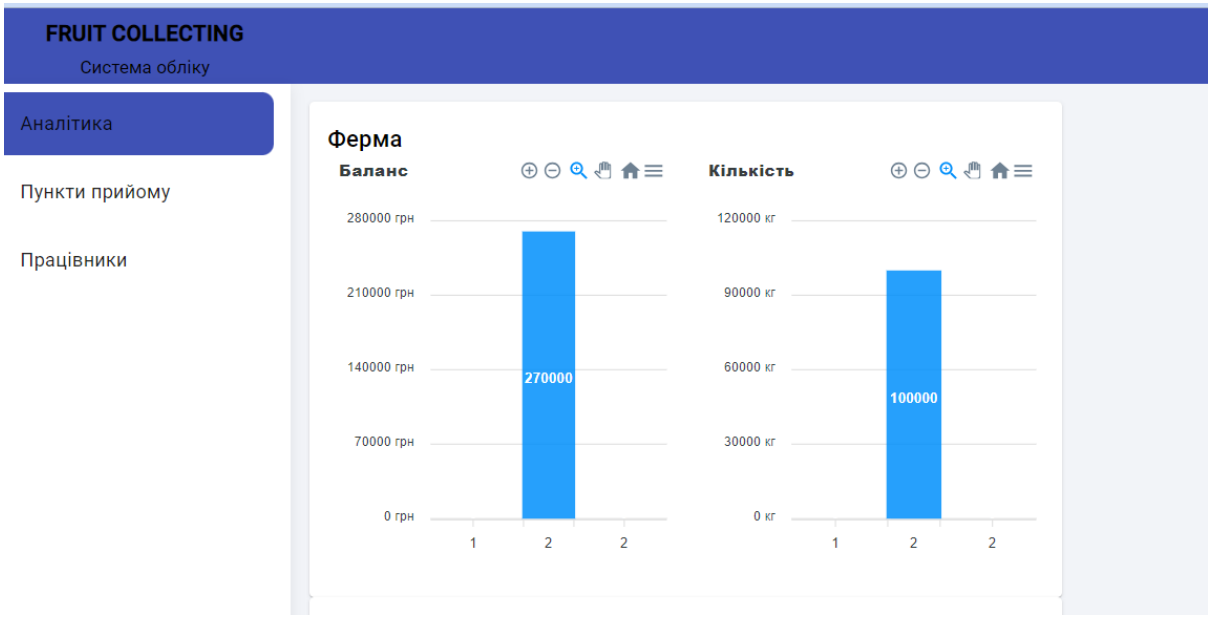
```
<main id="login-page">
  <section id="login">
    <span>
      <img src="../../../assets/icons/gala-apple-1_1.png" alt="Logo" width="400" height="300">
    </span>

    <form id="login-form" [formGroup]="loginForm" (ngSubmit)="login()" (keyup.enter)="login()">
      <mat-form-field appearance="outline">
        <input type="text" matInput formControlName="login" placeholder="Логін" required>
        <mat-error *ngIf="loginForm.hasError( errorCode: 'required', path: 'login')">
          {{ 'Введіть логін' }}
        </mat-error>
      </mat-form-field>

      <mat-form-field appearance="outline">
        <input type="password" matInput formControlName="password" placeholder="Пароль" required>

        @if(loginForm.hasError('required', 'password')) {
          <mat-error>
            Введіть пароль
          </mat-error>
        }
      </mat-form-field>
      @if(isAuthenticationError) {
        <mat-error>
          Такого користувча немає в системі
        </mat-error>
      }
```

- TypeScript частина

```
login(): void {
  this.loginForm.markAllAsTouched();

  if (this.loginForm.valid) {
    this.authenticationService.login(this.loginForm.controls['login'].value!, this.loginForm.controls['password'].value!)
      .subscribe( next: response => {
          this.isAuthenticationError = response;
          let role = this.authenticationService.getCurrentUser()?.role;

          if (role) {
            this.router.navigate( commands: ['/main/analytics']).then(() => {
              this.cd.detectChanges();
            });
          }
        }
      );
  }
```

4.2 Аналітика

Сторінка на якій відображається скільки коштів та продукції на пункті прийому.



- HTML ресурс сторінки

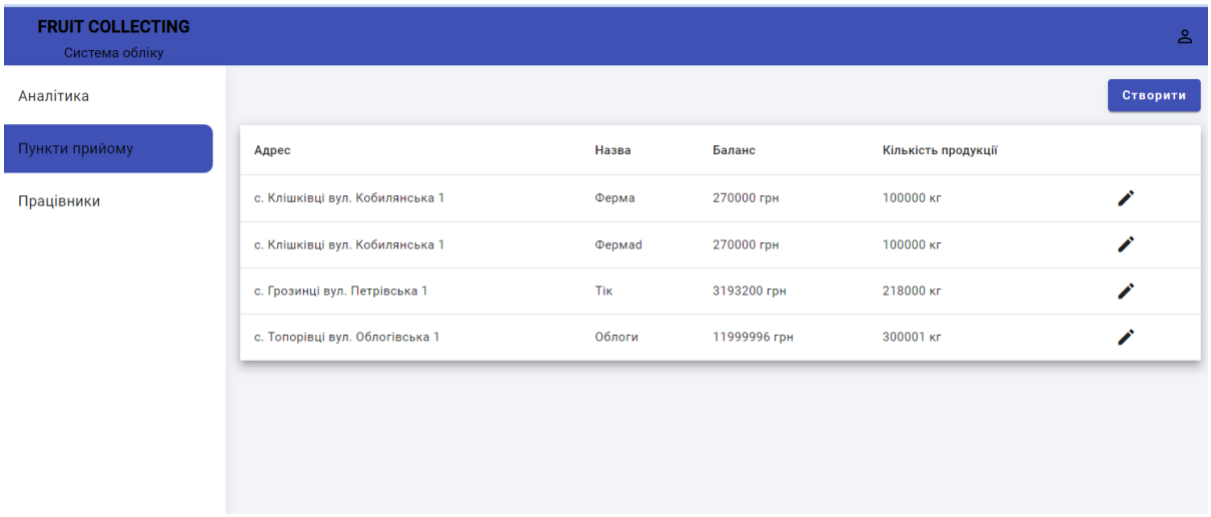
```
<div class="flex flex-row flex-wrap justify-between p-4">
  @for(i of mafs; track i.id) {
    <app-analytics-item [maf]="i"></app-analytics-item>
  }
</div>
```

- TypeScript частина

```
@Component({
  selector: 'app-analytics',
  templateUrl: './analytics.component.html',
  styleUrls: ['./analytics.component.css']
})
export class AnalyticsComponent implements OnInit {
  private authService = inject(AuthenticationService);
  private mafService = inject(MafService);

  mafs: any[] = [];

  ngOnInit(): void {
    const observer = this.authService.getUser().role !== "EMPLOYEE" ? this.mafService.getAll() : this.mafService.getById(
      map( project: (response: any) => [response])
    );

    observer.subscribe( next: (response: any) => {
      this.mafs = response;
    });
  }
}
```

4.3 Пункти прийому

Сторінка на якій відображаються всі пункти прийому та їх інформація.



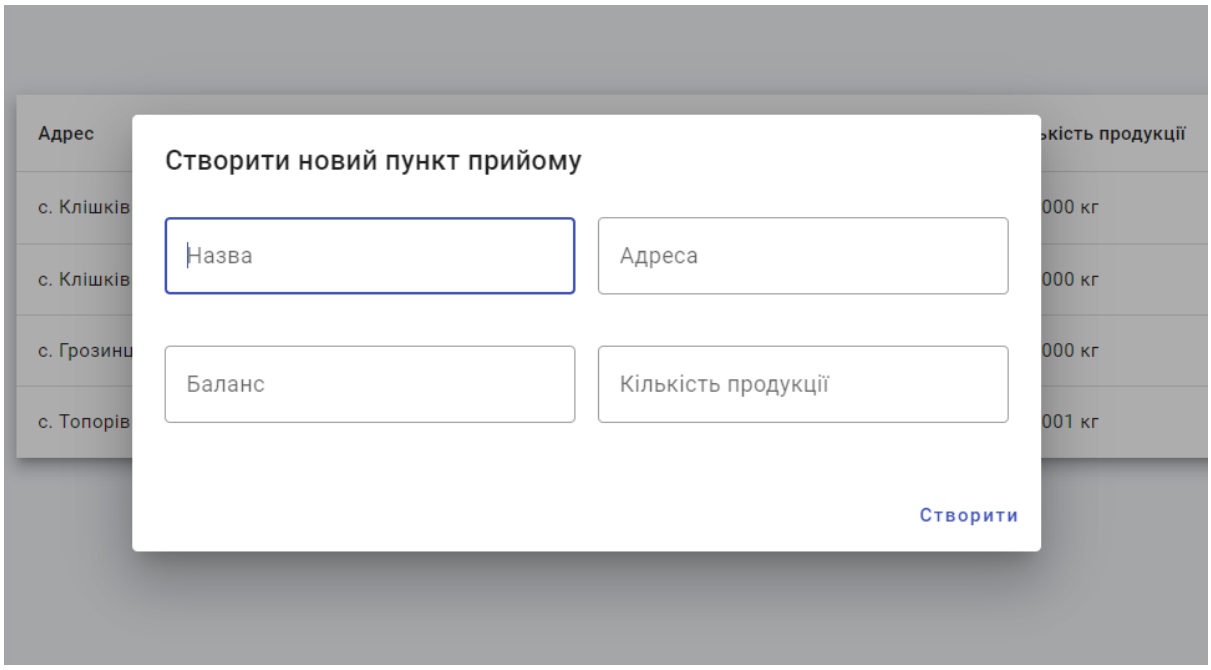
- HTML ресурс сторінки

```
<div>
  <table mat-table [dataSource]="mafs" class="mat-elevation-z8">
    <ng-container matColumnDef="address">
      <th mat-header-cell *matHeaderCellDef> Адрес </th>
      <td mat-cell *matCellDef="let element"> {{element?.address}} </td>
    </ng-container>

    <ng-container matColumnDef="name">
      <th mat-header-cell *matHeaderCellDef> Назва </th>
      <td mat-cell *matCellDef="let element"> {{ element?.name }} </td>
    </ng-container>

    <ng-container matColumnDef="balance">
      <th mat-header-cell *matHeaderCellDef> Баланс </th>
      <td mat-cell *matCellDef="let element"> {{element?.balance}} грн </td>
    </ng-container>

    <ng-container matColumnDef="quantity">
      <th mat-header-cell *matHeaderCellDef> Кількість продукції </th>
      <td mat-cell *matCellDef="let element"> {{element?.quantity}} кг </td>
    </ng-container>
```

- TypeScript частина

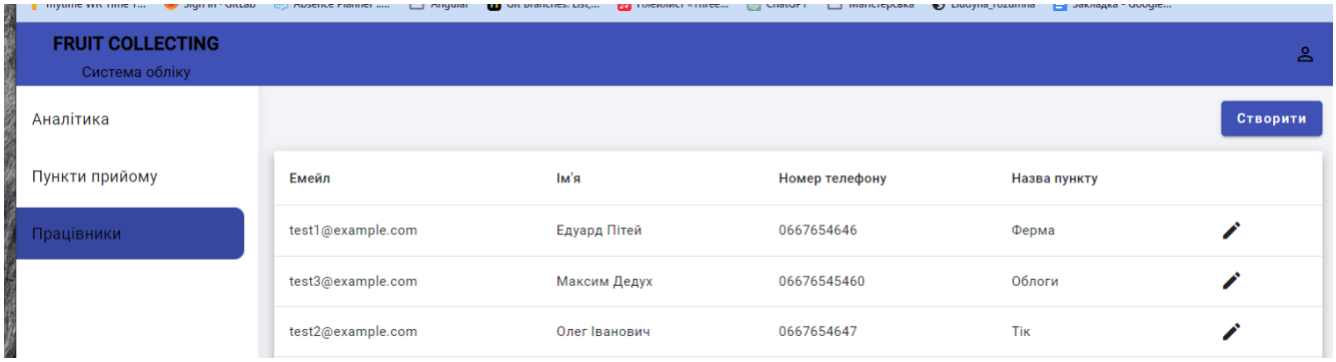
```typescript
@Component({
  selector: 'app-maf',
  templateUrl: './maf.component.html',
  styleUrls: ['./maf.component.css']
})
export class MafComponent implements OnInit {
  public dialog = inject(MatDialog);
  public cd = inject(ChangeDetectorRef);
  public mafService = inject(MafService);

  displayedColumns: string[] = ['address', 'name', 'balance', 'quantity', 'star'];
  mafs: any[] = [];

  ngOnInit(): void {
    this.getData();
  }
```

Форма для створення та редагування



4.4 Працівники

Сторінка на якій відображається список працівників на підприємстві з можливістю їх створення або редагування.



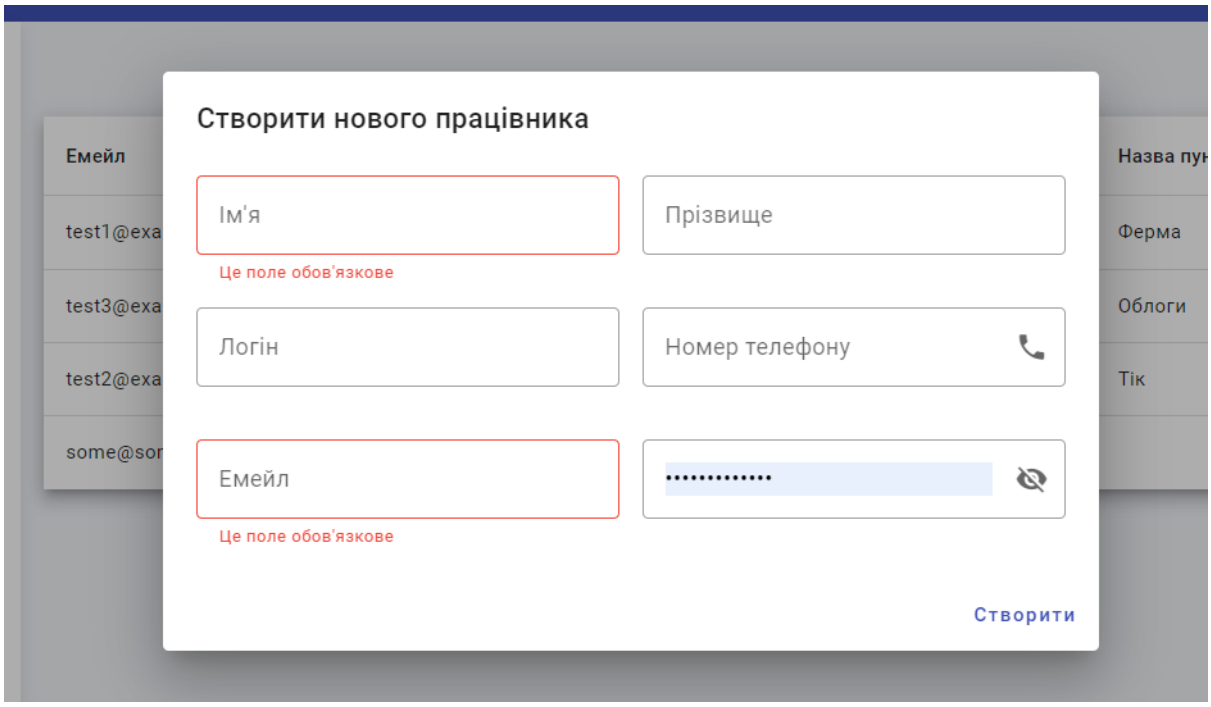
- HTML ресурс сторінки

```
<table mat-table [dataSource]="employee" class="mat-elevation-z8">
  <ng-container matColumnDef="email">
    <th mat-header-cell *matHeaderCellDef> Емейл </th>
    <td mat-cell *matCellDef="let element"> {{element.email}} </td>
  </ng-container>

  <ng-container matColumnDef="name">
    <th mat-header-cell *matHeaderCellDef> Ім'я </th>
    <td mat-cell *matCellDef="let element"> {{ element.firstName + " " + element.lastName }} </td>
  </ng-container>

  <ng-container matColumnDef="phone">
    <th mat-header-cell *matHeaderCellDef> Номер телефону </th>
    <td mat-cell *matCellDef="let element"> {{element.phone}} </td>
  </ng-container>

  <ng-container matColumnDef="mafName">
    <th mat-header-cell *matHeaderCellDef> Назва пункту </th>
    <td mat-cell *matCellDef="let element"> {{element.mafName}} </td>
  </ng-container>
```

- TypeScript частина

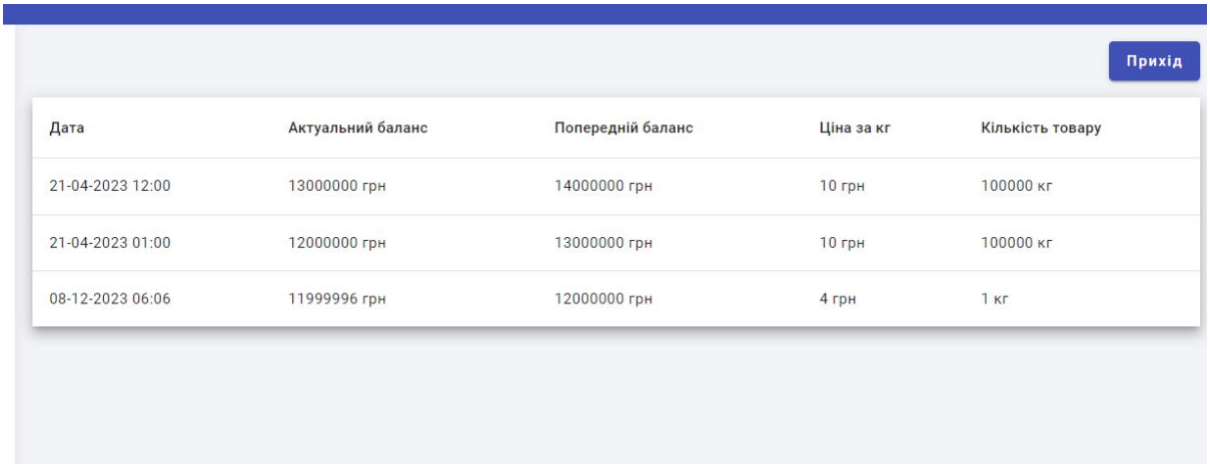
```typescript
@Component({
  selector: 'app-employee',
  templateUrl: './employee.component.html',
  styleUrls: ['./employee.component.css']
})
export class EmployeeComponent implements OnInit {
  public dialog = inject(MatDialog);
  public employeeService = inject(EmployeeService);
  public authService = inject(AuthenticationService);

  displayedColumns: string[] = ['email', 'name', 'phone', 'mafName', 'star'];
  employee: any[] = [];

  ngOnInit(): void {
    if (this.authService.getUser().role === "EMPLOYEE") {
      this.displayedColumns.pop();
    }

    this.getData();
  }
```

Форма для створення та редагування



4.5 Баланс

Сторінка на якій звичайний працівник робить прихід продукції, та бачить історію цих приходів.

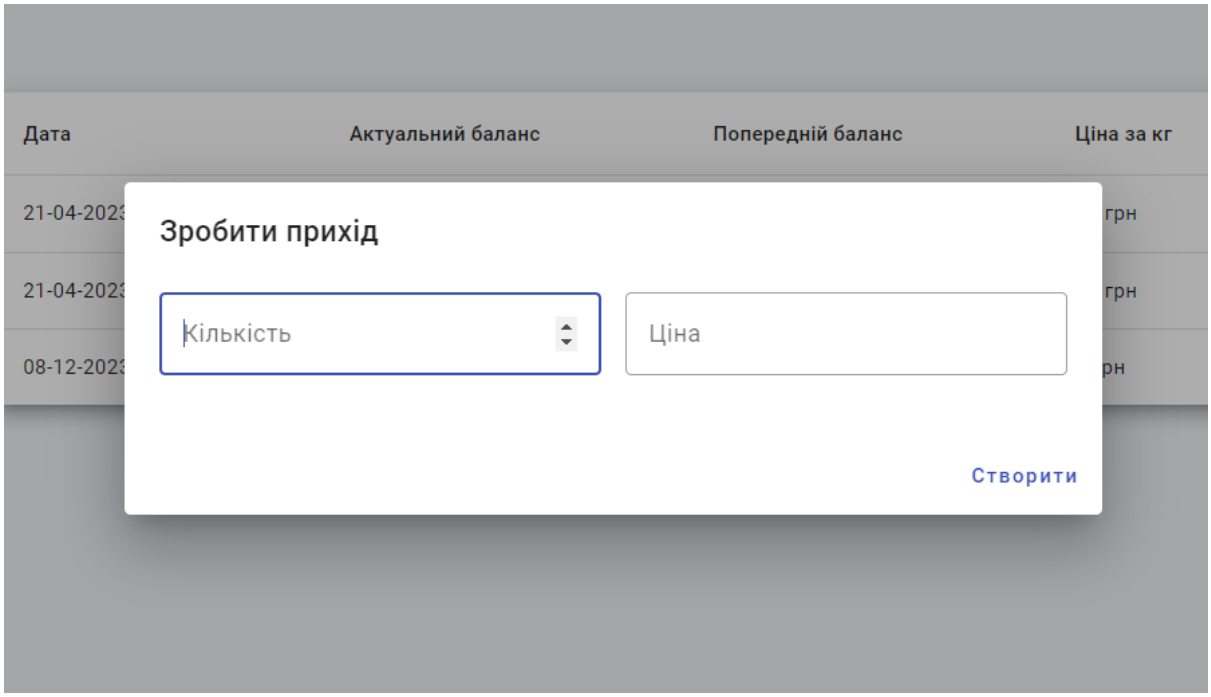


- HTML ресурс сторінки

```
<table mat-table [dataSource]="balances" class="mat-elevation-z8">
    <ng-container matColumnDef="creatingDate">
        <th mat-header-cell *matHeaderCellDef> Дата </th>
        <td mat-cell *matCellDef="let element"> {{element.creatingDate | date: 'dd-MM-YYYY hh:mm' }} </td>
    </ng-container>

    <ng-container matColumnDef="currentBalance">
        <th mat-header-cell *matHeaderCellDef> Актуальний баланс </th>
        <td mat-cell *matCellDef="let element"> {{ element.currentBalance }} грн </td>
    </ng-container>

    <ng-container matColumnDef="previousBalance">
        <th mat-header-cell *matHeaderCellDef> Попередній баланс </th>
        <td mat-cell *matCellDef="let element"> {{element.previousBalance}} грн </td>
    </ng-container>

    <ng-container matColumnDef="price">
        <th mat-header-cell *matHeaderCellDef> Ціна за кг </th>
        <td mat-cell *matCellDef="let element"> {{element.price}} грн </td>
    </ng-container>

    <ng-container matColumnDef="quantity">
        <th mat-header-cell *matHeaderCellDef> Кількість товару </th>
        <td mat-cell *matCellDef="let element"> {{element.quantity}} кг </td>
```

- TypeScript частина

```
export class BalanceComponent implements OnInit {
  public dialog = inject(MatDialog);
  public cd = inject(ChangeDetectorRef);
  private balanceService = inject(BalanceService);
  balances: any[] = [];
  displayedColumns: string[] = ['creatingDate', 'currentBalance', 'previousBalance', 'price', 'quantity'];

  ngOnInit(): void {
    this.getData();
  }

  private getData(): void {
    this.balanceService.getAll().subscribe( next: (response: any) => {
      this.balances = response;
      this.cd.markForCheck();
    });
  }

  openCreateDialog(): void {
```



- HTML ресурс сторінки

```
<h2 mat-dialog-title>Зробити прихід</h2>

<mat-dialog-content>
    <div class="grid grid-cols-2 gap-4 form-wrap" [formGroup]="form">
        <mat-form-field appearance="outline" class="form-input">
            <input type="number" formControlName="quantity" matInput placeholder="Кількість">

            @if (form.controls.quantity.invalid) {
                <mat-error>Це поле обов'язкове</mat-error>
            }
        </mat-form-field>

        <mat-form-field appearance="outline" class="form-input">
            <input type="number" formControlName="price" matInput placeholder="Ціна">
            @if (form.controls.price.invalid) {
                <mat-error>Це поле обов'язкове</mat-error>
            }
        </mat-form-field>
    </div>
</mat-dialog-content>
```

- TypeScript частина

```
})
export class BalanceDialogComponent {
  private fb = inject(FormBuilder);
  private balanceService = inject(BalanceService);
  private dialogRef = inject(MatDialogRef<BalanceDialogComponent>);

  form = this.fb.group( controls: {
    quantity: ['', Validators.required],
    price: ['', Validators.required]
  });

  save(): void {
    if (this.form.valid) {
      this.balanceService.create(this.form.value).subscribe( next: () => {
        this.dialogRef.close( dialogResult: true);
      });
    } else {
      this.form.markAllAsTouched();
    }
  }
}
```

ВИСНОВКИ

1. Проведений аналіз веб-сервісів та пов'язаних технологій свідчить, що вони ґрунтуються на відкритих та широко поширених протоколах Інтернету. Це, з одного боку, визначає їхню головну силу, але, з іншого боку, може вважатися істотною слабкістю. Веб-сервіси надають численні переваги, і хоча вони не ідеальні у всьому, це категорія програмних продуктів, яка стрімко розвивається та має перспективи для майбутнього.

2. На основі проведеного аналізу рекомендуємо при розробці веб-додатків використовувати архітектуру RESTful, що допомагає покращити якість програмного забезпечення. Планування колекцій та ресурсів RESTful від самого початку сприяє не лише покращенню архітектури програми, але й полегшує створення API для інтеграції з іншими службами у майбутньому.

3. В роботі було використано ряд технологій, таких як Docker для швидкого та легкого розгортання додатків, Spring Security для реалізації авторизації та реєстрації, а також фреймворк Angular для швидкої та динамічної фронтенд-розробки. TypeScript та RxJS допомагають вирішити завдання асинхронного програмування та підвищити ефективність веб-додатку.

4. Описана методологія розробки веб-додатку застосована для створенню гнучкого та легко розширюваного проекту для обліку продукції та менеджменту персоналу підприємства, який планується до впровадження.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Spring Data, Reference Documentation / Електронний ресурс. – Режим доступу: https://docs.spring.io/springdata/jpa/docs/current/reference/html/#reference
2. Spring Security, Reference Documentation / Електронний ресурс. – Режим доступу: https://spring.io/projects/spring-security
3. Hibernate, Reference Documentation / Електронний ресурс. – Режим доступу: https://hibernate.org/orm/documentation/6.0/
4. Head First Java / К. Сьерра Б. Бейтс, 2017. – 417с.
5. Effective Java, Джошуа Блох 2016. – 768с.
6. Maven – introduction /Електронний ресурс – Режим доступу до ресурсу: https://maven.apache.org/what-is-maven.html
7. Раджпут Д. Spring. Design Patterns / Master efficient application development with patterns such as proxy, singleton, the template method, and more, 2019. — 320с
8. К. Бауэр, Г. Грегорі, Г. Кінг. / Java Persistence with Hibernate, 2017. — 632с. Електронний ресурс. – Режим доступу: https://docs.oracle.com/en/java/javase/16/
9. Elliott, James. / Hibernate: A Developer's Notebook, 2004. — 190с.
10. Linwood, Jeff, Minter, Dave / Beginning Hibernate, 2010. — 400с.
11. Джейсон Мак-Колм Смит / Elemental Design Patterns, 2012. — 304с.
12. Марк Гранд / Patterns in Java, Volume 1. A Catalog of Reusable Design Patterns Illustrated with UML, 2004. — 560с.
13. Стів Макконнелл / Code complete 2005. — 896 с.

14. Петюшкин А. HTML. Экспрес-курс. – Київ:Фенікс-паблішинг, 2016. - 192 с.
15. Джеремі К. HTML5 для веб-дизайнерів. – Київ:Віват, 2015. - 410 с.
16. Фрісбі М. Angular 2 Cookbook: Discover over 70 recipes that provide the solutions you need to know to face every challenge in Angular 2, – Харків: Фенікс Пабліш, 2017: – 388 с.