Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: «**ТЕХНОЛОГІЇ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ З ВИКОРИСТАННЯМ СТЕКУ ФРЕЙМВОРКІВ SPRING НА БАЗІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ**»

Виконав: студент 4-го року навчання,

Освітньої програми «Комп'ютерні науки», 122

Вавдійчик Віктор Олександрович

Керівник старший викладач

Борозенний С.О.

Рецензент _____

(прізвище та ініціали)

Кваліфікаційна робота захищена

з оцінкою _____

Секретар ЕК _____

«____» _____ 20_24_ р.

Київ – 2024

Тема: «Технології реалізації застосунку з використанням стеку фреймворків Spring на базі мікросервісної архітектури»

План виконання роботи:

№ з/п	ЕТАПИ РОБОТИ	ТЕРМІН ВИКОНАННЯ
1.	Вибір теми кваліфікаційної роботи	жовтень 2023
2.	Складання графіка роботи над темою	листопад 2023
3.	Збирання матеріалу. Опрацювання літератури.	листопад – грудень 2023
4.	Вибір програмного забезпечення	січень 2024
5.	Теоретичний огляд засобів розробки	лютий-березень 2024
6.	Написання практичної частини	квітень 2024
7.	Надання кваліфікаційної роботи на перевірку навчальному керівнику	травень 2024
8.	Остаточне оформлення текстової частини і презентації	травень 2024
9.	Захист кваліфікаційної роботи	30 травня 2024

Рецензент – Сініцина Р. Б.

ЗМІСТ

Стор.

Вступ

В останні роки спостерігається значне зростання популярності мікросервісної архітектури серед розробників програмного забезпечення, що пояснюється необхідністю створення масштабованих, гнучких та стійких до відмов додатків. Мікросервісний підхід до розробки засосунків дозволяє розбити великі монолітні системи на набір дрібних, незалежних сервісів, кожен з яких виконує конкретну функцію і може розроблятися, тестуватися та розгортатися окремо. Стек фреймворків Spring, включаючи Spring Boot та Spring Cloud, забезпечує потужні інструменти та бібліотеки для побудови мікросервісних додатків на Java, що робить його одним з найпопулярніших рішень у цій галузі.

Однак, на сьогоднішній день деякі методи розробки мікросервісних додатків на Java недостатньо вивчені. Тому виникає необхідність провести власне комплексне дослідження для встановлення ефективних методів і підходів до реалізації мікросервісної архітектури з використанням стеку фреймворків Spring, а також показати практичне їх застосування.

Тому, метою даної роботи є аналіз необхідних технологій для ефективної реалізації мікросервісного застосунку з використанням стеку фреймворків Spring, а також використання цих технологій на практиці.

Мета роботи зумовила наступне ***наукове завдання:***

- Проаналізувати основні модулі стеку фреймворків Spring та принципи роботи з ними.
- Дослідити задачі та принципи мікросервісної архітектури, а також розглянути її недоліки.

- Розробити та реалізувати мікросервісний додаток.
- Забезпечити контейнеризацію та моніторинг мікросервісів.

Робота складається з чотирьох розділів.
Перший розділ присвячено аналізу теоретичних основ роботи зі стеком фреймворків Spring. Також у цьому розділі розглядаються основні інструменти Spring Cloud.

В другому розділі розглядаються основні принципи та задачі мікросервісної архітектури, а також інструмент контейнеризації Docker.

Третій розділ присвячено реалізації мікросервісного застосунку Scheduler API на мові Java використовуючи інструменти Spring Cloud.
В четвертому розділі розглядається контейнеризація мікросервісного застосунку Scheduler API та моніторинг контейнерів.

Запропоновані методи та інструменти можуть бути впроваджені в реальні проекти для підвищення ефективності розробки, розгортання та експлуатації мікросервісів. Було розроблено мікросервісний застосунок та проведено його контейнеризацію.

Розділ 1.
Теоретичні аспекти роботи зі стеком фреймворків Spring.

1.1 Загальний огляд модулів Spring

Spring - це набір взаємопов'язаних міні-фреймворків, створених для роботи над різними частинами програми. Підключати їх можна окремо, залежно від завдань. При цьому застосовувати його можуть не лише Java-розробники, а й ті, хто працює з Kotlin або Groovy.

До екосистеми Spring входять наступні модулі:
Spring Core, Spring AOP, Spring Web MVC, Spring DAO, Spring ORM, Spring context, Spring Web flow та ініші.

Розглянемо фреймворки, які будуть використовуватися в подальшому в даній роботі.

Spring Core – це базовий модуль Spring, який надає функціональність Inversion of Control (IoC) та Dependency Injection (DI).

Inversion of Control (IoC) - це принцип проектування програмного забезпечення, у якому контроль за створенням та життєвим циклом об'єктів переходить від додатка до контейнера. Це означає, що замість того, щоб явно створювати об'єкти в коді, ми визначаємо конфігурацію об'єктів у контейнері IoC, який потім створює та керує цими об'єктами.

Dependency Injection (DI) - це конкретний механізм реалізації принципу IoC у Spring Framework. Він дозволяє впроваджувати залежності об'єктів до інших об'єктів, не створюючи їх у коді. У Spring DI залежності визначаються в конфігураційних файлах, а Spring контейнер автоматично впроваджує ці залежності в потрібні об'єкти.

Spring Core надає багато можливостей для роботи з IoC та DI. За допомогою Spring Core можливо створювати та керувати об'єктами програми, впроваджувати залежності, вирішувати проблему з надмірною складністю коду, пов'язаною зі створенням та налаштуванням об'єктів. Більш детальний огляд того, як працює DI в Spring буде розглянуто в наступному пункті цього розділу.

Spring Data — модуль, який забезпечує програмним додаткам доступ до даних за допомогою реляційних та нереляційних баз даних, хмарних сервісів та фреймворків map-reduce.

Перш за все, Spring Data включає багато підпроектів, які призначені для певних СУБД, таких як MySQL, Redis, MongoDB та ін. Крім того, є можливість використовувати підмодулі, які розроблені спільнотою Spring для більш спеціалізованих БД типу ArangoDB, Microsoft Azure Cosmos DB, Google Datastore та інших.

Spring Web MVC – це веб-фреймворк за схемою "модель-представлення-контролер". Spring MVC дозволяє створювати спеціальні біни для обробки вхідних HTTP-запитів. Методи контролера відображаються на протокол HTTP.

Spring Security – це фреймворк, що надає механізми побудови систем автентифікації та авторизації, а також інші можливості безпеки для корпоративних додатків, створених за допомогою Spring.

Spring Boot – це фреймворк, який призначений для розробки додатків в екосистемі Spring, він спрощує та прискорює процес розробки, забезпечуючи стандартизований спосіб конфігурації, роботи з залежностями та автоматизованого розгортання.
Функціональні можлиості Spring Boot:

1. Спрощене конфігурування.
   Spring Boot використовує конвенції над конфігураціями, що дозволяє швидко створювати додатки без необхідності великих обсягів XML-конфігурацій. Він автоматично конфігурує багато аспектів додатка на основі класів, анотацій та значень за замовчуванням.

2. Вбудований контейнер сервлетів.
   Spring Boot містить вбудовані контейнери сервлетів (наприклад, Tomcat, Jetty або Undertow), що дозволяє запускати додатки як самостійні JAR-файли без необхідності додаткового конфігурування серверів додатків.

3. Автоматична конфігурація залежностей.
   Spring Boot надає засоби для автоматичного управління залежностями через інструменти збірки проекту, такі як Maven або Gradle. Він дозволяє швидко додавати та оновлювати бібліотеки та фреймворки без необхідності вручного вказування версій.

4. Автоматичне конфігурування Spring.
   Spring Boot автоматично сканує ваші класи та конфігурує Spring контекст за допомогою анотацій, таких як @Component, @Service, @Controller та інші. Це спрощує роботу з інверсією управління та внедренням залежностей.

5. Управління властивостями.
   Spring Boot надає можливість керування властивостями додатка через файл `application.properties` або `application.yml`. Це дозволяє налаштовувати поведінку додатка без перекомпіляції коду.

6. Активація профілів.
   Spring Boot підтримує профілі, які дозволяють налаштовувати додаток для різних середовищ (наприклад, розробка, тестування, продукція). Це дозволяє використовувати різні конфігурації для різних умов.

7. Підтримка тестування.
   Spring Boot надає засоби для тестування додатків, включаючи модульне тестування, інтеграційне тестування та тестування кінцевих точок REST. Він інтегрується з популярними фреймворками для тестування, такими як JUnit та Mockito.

Spring Boot дозволяє розробникам швидко створювати робочі та ефективні додатки на Java, забезпечуючи гнучкість, простоту використання та різноманітність функціональних можливостей.

Також варто зазначити, що окрім фреймворків із екосистеми Spring, під час розробки будуть використані також інші фреймворки, такі як фреймворк Hibernate, який входить до базового пакету spring starter data jpa.

Hibernate Framework – це фреймворк для мови Java, призначений для роботи з базами даних. Він реалізує об'єктно-реляційну модель - технологію, яка "з'єднує" програмні сутності та відповідні записи в базі.

Об'єктно-реляційна модель або ORM дозволяє створити програмну «віртуальну» базу даних з об'єктів. Об'єкти описуються мовами програмування із застосуванням принципів ОВП. Java Hibernate – популярне втілення цієї моделі.

Hibernate побудований на специфікації JPA 2.1 - наборі правил, що визначає взаємодію програмних об'єктів із записами в базах даних. JPA пояснює, як керувати збереженням даних з коду Java в базу.

Застосування цих фреймворків на практиці детально описано у третьому розділі даної роботи.

1.2 Принцип роботи Dependency Injection в Spring.

Як вже зазначалося, впровадження залежності (DI – Dependency injection) – це процес надання програмному компоненту зовнішньої залежності. У Spring Framework DI є одним із ключових принципів, що допомагає вирішувати проблеми, пов'язані з управлінням залежностями та спрощує структуру коду. Цей механізм впроваджує принцип інверсії контролю (Ioc). У Spring відбувається інверсія керування, коли контейнер управляє життєвим циклом та залежностями об'єктів. Замість того, щоб програміст створював об'єкти напряму, контейнер Spring створює їх та встановлює всі необхідні залежності.

Контейнер IoC Container в Spring є контейнером, що керує об'єктами. Під час запуску додатку, контейнер створює об'єкти, ініціалізує їх та встановлює всі залежності. Об'єкти, що управляються контейнером IoC, називаються бінами (або компонентами) . Це можуть бути сервіси, DAO-об'єкти, контролери тощо.
Компоненти JavaBean (їх ще називають Plain Old Java Object (POJO) - старий добрий об'єкт Java) надають стандартний механізм, що дозволяє створювати Java-ресурси, що конфігуруються безліччю способів. За допомогою інтерфейсів можливо отримати найбільшу віддачу від DI, адже біни здатні використовувати для задоволення їх залежності будь-яку реалізацію інтерфейсу.
Для конфігурації компонентів в Spring існують три різних методи:
- Зовнішньо в XML-файлах: Створюється окремий xml файл (Рис. 1.1) конфігурації, в якову описуються біни та залежності між ними (Рис. 1.2);
- За допомогою конфігураційних Java-класів Spring;
- За допомогою Java-анотацій в коді: в цьому методі ін'єкція залежностей відбувається через конструктор(Constructor Injection), сетер (Setter Injection) або поле (Field Injection) **:**

- При Constructor Injection залежні об'єкти передаються через конструктор класу.
- При Setter Injection залежності встановлюються за допомогою сетерів.
- При Field Injection залежні об'єкти ін'єктуються безпосередньо в поля класу.

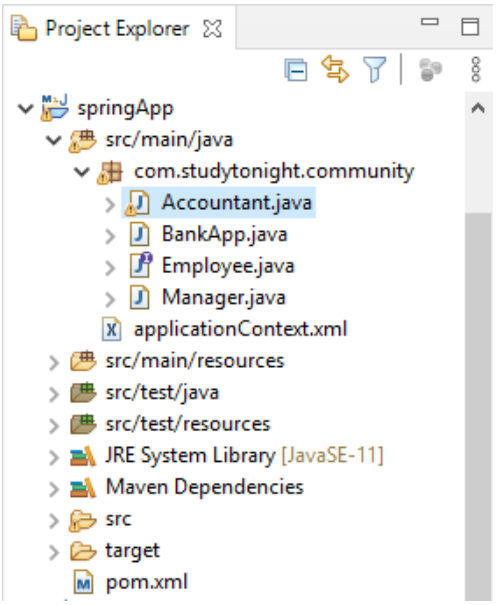

Рис. 1.1

Рис. 1.2

Розглянемо конфігурацію за допомогою конфігураційного класу на прикладі. Припустимо, у нас є клас UserService, який потребує залежності UserRepository. Ми будемо використовувати клас конфігурації для передачі UserRepository в UserService.

UserRepository – це інтерфейс, який має імплементацію у вигляді класу UserRepositoryImpl. Це виглядає наступним чином:

```
public interface UserRepository {
    void save(User user);
    User findById(Long id);
}
public class UserRepositoryImpl implements UserRepository {
    @Override
    public void save(User user) {
        // Логіка для збереження користувача в базі даних
    }
    @Override
    public User findById(Long id) {
        // Логіка для отримання користувача з бази даних за ідентифікатором
    }
}
```

В той же час, UserService має наступний код:

```
public class UserService {
   private final UserRepository userRepository;

   public UserService(UserRepository userRepository) {
      this.userRepository = userRepository;
   }
   public void saveUser(User user) {
      userRepository.save(user);
   }
   public User findUserById(Long id) {
      return userRepository.findById(id);
   }
 }
```

Щоб не встановлювати залежність кожен раз як знадобиться об’єкт класу UserService, Spring дає нам можливість зробити це через файл конфігурації, який позначається спеціальною анотацією @Configuration. Для цього прикладу цей клас буде виглядати наступним чином:

```
@Configuration
public class AppConfig {

   @Bean
   public UserRepository userRepository() {
      return new UserRepositoryImpl();
   }

   @Bean
   public UserService userService(UserRepository userRepository) {
      return new UserService(userRepository());
   }
}
```

Було використано анотацію @Bean, щоб позначити, що методи userRepository() та userService() повертають біни, які будуть доступні в контексті додатку Spring.

В цьому прикладі видно, що залежність між двома бінами встановлюється через констуктор (new UserService(userRepository())). Також, як вже було зазначено, залежність можна встановлювати також через поле або через сеттер.

Встановлення залежності через сеттер:

```
  private UserRepository userRepository;

   public void setUserRepository(UserRepository userRepository) {
      this.userRepository = userRepository;
   }
```

Встановлення залежності через поле:

```
  @Autowired
  private UserRepository userRepository;
```

При встановлені залежності через поле використовується анотація @Autowired, яка показує Spring контейнеру, що тут потрібно встановити залежність.

Окрім встановлення залежностей через клас кофігурації, в Spring є можливість це зробити за допомогою анотацій в самих класах. Для цього над класом ставиться анотація @Component або @Service. . Для прикладу описаного вище, це виглядало б наступним чином:

```
@Component
public class UserService {
   private final UserRepository userRepository;

   public UserService(UserRepository userRepository) {
      this.userRepository = userRepository;
   }

…
```

```
}
```

Також цю анотацію потрібно поставити над класом User RepositoryImpl.

```
@Component
public class UserRepositoryImpl implements UserRepository {
```

Тепер можна обійтися без класу конфігурації, бо Spring бачить створені біни за допомогою анотацій @Component і сам встановлює потрібні залежності.

Отже, після визначення, що таке Dependency Injection та демонстрації, як це працює на практиці, можна виділити основні переваги такого підходу:

1. Спрощення коду: DI дозволяє позбутися відповідальності за створення та управління залежностями, що робить код більш чистим та зрозумілим.

2. Легка заміна компонентів: Через внедрення залежностей, можна легко замінити одну реалізацію іншою без змін в інших частинах коду.

3. Тестованість: DI дозволяє легко створювати тестові об'єкти та замінювати реальні залежності мок-об'єктами для тестування.

Також, можна зазначити певний недолік Dependency Injection в Spring, а точніше помилку, яку часто допускають програмісти початківці. Мова йде про циклічність залежностей: якщо не обережно конфігурувати залежності, це може призвести до циклічних залежностей, що ускладнює розробку та підтримку коду.

Отже, використання Dependency Injection у Spring вважається дуже корисним та ефективним підходом, який допомагає зробити ваш код більш модульним, легким у розумінні та тестуванні.

1.3 Колекція інструментів Spring Cloud

Spring Cloud – це колекція інструментів, що поєднує розробки з відкритим вихідним кодом багатьох компаній, таких як VMware, HashiCorp та Netflix. Spring Cloud спрощує інсталяцію та налаштування проектів та надає реалізацію шаблонів, що особливо часто зустрічаються в додатках на основі Spring. Завдяки цій колекції розробники можуть більше зосередитися на розробці свого коду, залишивши осторонь деталі налаштування всієї інфраструктури, яка використовується для створення давання та розгортання мікросервісів.

Особливості Spring Cloud, які зазначають розробки цієї колекції інструментів:

- Розподілена конфігурація
- Реєстрація та відкриття послуги
- Маршрутизація
- Виклики між службами
- Балансування навантаження
- Автоматичні вимикачі
- Розподілений обмін повідомленнями
- Короткотермінові мікросервіси (завдання)

Spring Cloud надає рішення для таких ключових задач, як керування конфігураціями, реєстрація служб, балансування навантаження, обробка відмов, моніторинг та інші. Розглянемо детальніше основні компоненти Spring Cloud:

1) **Spring Cloud Config** надає можливість централізованого керування конфігураціями додатків. Конфігураційні файли можуть бути збережені у Git, SVN або будь-якому іншому репозиторії. Додатки автоматично завантажують конфігурації з цього репозиторію під час запуску.
2) **Spring Cloud Service Discovery.**
   Програма на основі мікросервісів зазвичай працює у віртуалізованих або контейнерних середовищах. Кількість екземплярів служби та її розташування динамічно змінюються. Розробникам часто необхідно знати, де знаходяться ці екземпляри та їхні імена, щоб дозволити

запитам надходити до потрібного мікросервісу. Саме тут і потрібне виявлення служб. Цей механізм допомагає нам знати, де знаходиться кожен екземпляр. Service Discovery працює як реєстр, де знаходяться адреси всіх примірників. Отже, якщо клієнт хоче зробити запит до служби, він повинен використовувати механізм виявлення служби.

У Spring Cloud Service Discovery можна за допомогою декількох служб, а саме: Consult, Zookeeper, Eureka. Найбільш широко використовуваною є Eureka, тому в цій роботі буде розглянуто саме її.

3) **Spring Cloud Sleuth** надає можливість створення унікальних ідентифікаторів слідування для кожного запиту, що дозволяє відстежувати його маршрут через різні мікросервіси. Він інтегрується з різними інструментами моніторингу, такими як Zipkin або Prometheus, для забезпечення обширного моніторингу додатків.
4) **Spring Cloud Gateway** дозволяє створювати мікросервісні шлюзи, які надають централізований доступ до мікросервісів. Вони можуть виконувати функції маршрутизації, фільтрації, обробки відмов та інших.
5) **Spring Cloud LoadBalancer** спрощує інтеграцію з агентами виявлення служб, такими як Eureka, але також забезпечує балансування навантаження за клієнта. Це дозволяє клієнту продовжувати звертатися до служби, навіть якщо агент виявлення тимчасово недоступний.

Spring Cloud дозволяє розробникам побудувати розподілені системи, які є надійними, масштабованими та легко керованими. Він надає готові рішення для багатьох основних проблем, з якими стикаються розробники при роботі з мікросервісною архітектурою, та дозволяє зосередитися на розробці функціональності додатків.

Висновки за главою 1

Розглянуті в роботі теоретичні аспекти роботи зі стеком фреймворків Spring дозволяють зробити кілька важливих висновків щодо можливостей та переваг цього інструментарію.

Spring Framework є надзвичайно потужним і гнучким інструментом, який забезпечує розробників широким набором модулів для вирішення різноманітних завдань. Основні модулі, такі як Spring Core, Spring AOP, Spring Data, Spring MVC та Spring Security, охоплюють всі аспекти розробки сучасних додатків. Spring Core, зокрема, реалізує концепцію Inversion of Control (IoC).

Dependency Injection (DI) є ключовою концепцією Spring Framework, яка забезпечує ефективне управління залежностями між компонентами додатку. Використання DI дозволяє знизити зв'язаність компонентів, підвищити гнучкість та полегшити управління конфігурацією додатків.

Spring Cloud розширює можливості Spring для роботи з розподіленими системами та мікросервісною архітектурою. Він забезпечує всі необхідні інструменти для створення надійних, масштабованих та гнучких мікросервісних архітектур, дозволяючи легко інтегрувати сервіси, управляти їхньою конфігурацією та забезпечувати високу продуктивність. Spring Cloud Netflix, Spring Cloud Gateway та Spring Cloud Security надають можливості для виявлення сервісів, клієнтського балансування навантаження та реалізації безпеки в розподілених системах.

Отже, використання стеку фреймворків Spring для розробки мікросервісних додатків дозволяє створювати надійні та масштабовані програмні системи. Отримані знання створюють міцну базу для подальшої практичної реалізації мікросервісних додатків з використанням стеку Spring.

Розділ 2.
Теоретичні аспекти роботи з мікросервісною архітектурою.

2.1 Задачі та принципи мікросервісної архітектури.

Мікросервіси - це архітектурний підхід до розробки додатків, який передбачає розбиття додатка на маленькі та незалежні компоненти, кожен із яких виконує певну функцію. Ці компоненти можуть взаємодіяти один з одним через API, що дозволяє розробляти програми більш гнучко та масштабовано.

Для того щоб краще зрозуміти, навіщо потрібна мікросервісна архітектура, варто спочатку розглянути дві інші архітектури побудови програмного забезпечення, а саме: n-рівнева та монолітна архітектури.

N-рівнева архітектура.

Одним із найпоширеніших типів архітектури для розробки корпоративного програмного забезпечення є багаторівнева або n-рівнева архітектура.

Програми, які розроблені в цій архітектурі поділяються на кілька рівнів, кожен з яких має свої обов'язки та функції, такими як інтерфейс користувача, служби, дані, тестування і т. д. . Наприклад, під час створення застосунку окремий проєкт або рішення створюється для користувацького інтерфейсу. Інший проєкт створюється для служб, потім ще один - для сервісів, ще один - для шару даних і так далі. У підсумку кілька проєктів об'єднуються для створення цілого застосунку. У великих корпоративних системах n-рівневі застосунки мають низку переваг, зокрема:

- N-рівневі програми дають змогу чітко розділити завдання. Такі елементи, як користувацький інтерфейс, дані та бізнес-логіка, можуть оброблятися окремо;
- команди розробників можуть працювати над різними компонентами незалежно одна від одної;
- архітектура підприємства добре зрозуміла, що дає змогу відносно легко знайти кваліфікованих розробників для багаторівневих проєктів.

Однак у n-рівневих додатків є й недоліки:

- Після внесення змін до коду необхідно зупиняти і перезапускати всю програму.
- Рефакторинг великих багаторівневих додатків після розгортання може виявитися складним завданням.

N-рівнева архітектура має деяку схожість з мікросервісною, а конкретно тим, що розподіляє елементи одного програмного застосунку на окремі додатки, які взаємодіють між собою. Проте такі окремі програми є залежними від “нижчого рівня” додатка, наприклад front-end від back-end.

Монолітна архітектура.

Багато веб-додатків невеликого та середнього розміру створюються з використанням монолітної архітектури. Монолітне додаток доставляється як єдиний, що розгортається програмний артефакт. Усі його компоненти – користувальницький інтерфейс, бізнес-логіка та логіка доступу до бази даних – об'єднані в єдиний додаток і розгортаються на одному сервері.

Звичайно, програма може бути розгорнута як єдине ціле, але часто над одним таким додатком працює кілька груп розробників. Кожна група відповідає за свою частину програми, яка зазвичай орієнтована на конкретних клієнтів. Наприклад, уявимо такий сценарій: є внутрішній додаток для управління взаємовідносинами з клієнтами, який передбачає координацію дій декількох груп розробки користувальницького інтерфейсу, логіки управління клієнтами, сховища даних, логіки управління фінансами та, можливо, багатьох інших.

Прихильники мікросервісної архітектури часто відгукуються про монолітні додатки з негативним відтінком, але іноді монолітна архітектура є правильним вибором. Моноліти простіше створювати і розгортати, ніж додатки розроблені на мікросервісній архітектурі. Якщо сценарій виковання чітко визначено і навряд чи зміниться в майбутньому, то часто рішення почати з моноліту може виявитися не таким поганим. Однак зі збільшенням розмірів та складності додатка управляти монолітом стає все важче. Будь-яка зміна в монолітному додатку може мати каскадний ефект, впливаючи на інші частини програми, що може суттєво ускладнити інтеграцію та розгортання у промисловій системі. Отже, мікросервісна архітектура, пропонує велику гнучкість та зручність супроводу.

Також, варто зазначити, що перехід на мікросервісну архітектуру, зубомовлений, зокрмема, і факторами конкуренції на ринку програмного забезпечення, а саме:

1. Зросла складність програмних продуктів. Клієнти очікують, що всі підрозділи організації знатимуть, хто вони є. Але «ізольовані» програми, що взаємодіють з однією базою даних і не що інтегруються з іншими додатками, більше не являються нормою. Сучасні додатки повинні взаємодіяти з безліччю служб та баз даних, що знаходяться не тільки всередині компанії, але і в інших компаніях.

2. Клієнти програмних продуктів хочуть швидше отримувати оновлення. Клієнти більше не хочуть чекати виходу наступної щорічної версії програмного пакета. Вони очікують, що функції у програмному продукті будуть розділені та нові версії будуть випускатися швидко, протягом кількох тижнів (чи навіть днів).

3. Значно виросла кількість клієнтів та їх запитів. Клієнти очікують, що їхні програми завжди будуть доступні. Глобальні програми надзвичайно ускладнюють прогнозування кількості транзакцій, з якими зможе впоратися додаток, і коли ця кількість буде досягнуто. Збої або проблеми в одній частині додатка не повинні призводити до припинення роботи всього додатка.

Тепер, коли були обговорені причини переходу на мікросервісну архітектуру, можна визначити основні принципи мікросервісної архітектури:

- логіка програми розбита на дрібні компоненти з чітко узгодженими межами відповідальності;
- кожен компонент відповідає за вузьке коло завдань та розгортається незалежно від інших;
- один мікросервіс відповідає за одну частину предметної області;
- для обміну даними між собою мікросервіси використовують полегшені протоколи, такі як HTTP та JSON (JavaScript Object Notation - форма запису об'єктів JavaScript);
- програми на основі мікросервісів завжди обмінюються даними з використанням технологічно нейтрального формату (найчастіше використовується JSON), тому технічна реалізація служби не має значення. Це означає, що додаток, що складається з мікросервісів, може бути написаний кількома мовами та з використанням кількох технологій;

Отже, головними задачами мікросервісної архітектури є:

- розробка легко маштабованого програмного продукту
- можливість більш швидкого впровадження оновлення, для якого не потрібно повністю перезапускати застосунок.
- розробка системи окремих додатків, яка взаємодіє між собою та має спільну базу даних.

2.2 Недоліки мікросервісів.

У попередньому пункті цього розділу вже зазначалося чому мікросервіси являють собою потужний архітектурний шаблон. Проте ще нічого не було сказано про те, коли не слід використовувати мікросервіси для створення додатків. Давайте розглянемо основні перешкоди:

1. Складність розподілених систем.

Мікросервіси за своєю природою є невеликими і розмежованими службами, тому вони тягнуть за собою на додачу ще ті складності, які відсутні в монолітних додатках. Мікросервісні архітектури вимагають високого ступеня технічської зрілості. Не слід розглядати можливість використання мікросервісів, якщо є бракує коштів на засоби автоматизації експлуатації та супроводу (моніторинг, масштабування і т. д.), яка необхідна для надійної роботи розподіленої програми.

2. Зростання віртуальних серверів або контейнерів без належного регулювання.

   Одна з найпоширеніших моделей розгортання мікросервісів – один екземпляр мікросервісу в одному контейнері. Великий додаток на основі мікросервісів може виявитися розгорнутим на великій кількості (від 50 до 100) серверів або контейнерів (зазвичай віртуальних), які необхідно створювати та підтримувати. Навіть за невисокої вартості запуску цих служб у хмарі, складність управління та моніторингу всього комплексу мікросервісів може бути дуже значною.

3. Тип програми.

   Мікросервіси орієнтовані на багаторазове використання і надзвичайно корисні для створення великих додатків, которі повинні бути дуже стійкими та масштабованими. Це одна із причин високої популярності мікросервісів. Якщо розробляються невеликі програми для окремих підрозділів організації або з невеликою базою користувачів, то складність, супутня побудові розподіленої моделі на основі мікросервісів, може призвести до невиправданого збільшення витрат.

4. Транзакції та узгодженість даних.

   Починаючи розглядати можливість впровадження мікросервісів,

обов'язково варто дослідити шаблони використання даних вашими службами та користувачами. Мікросервіс обгортає невелику кількість таблиць і добре працює як механізм виконання простих завдань, таких як створення і додавання даних і виконання нескладних запитів до сховища даних. Якщо ж програма повинна виконувати складні перетворення або обробляти дані з декількох джерел, то розподілена природа мікросервісів ускладнить цю роботу. Мікросервіси неминуче братимуть на себе занадто багато відповідальності можуть стати вразливими для проблем з продуктивністю.

2.3 Програмний засіб контейнеризації застосунків Docker.

Контейнери — це спосіб стандартизації розгортки програми та відокремлення її від загальної інфраструктури. Екземпляр програми запускається в ізольованому середовищі, що не впливає на основну операційну систему.

Це необхідно розробникам, оскільки їм не потрібно замислюватися, в якому оточенні працюватиме їхня програма, а також, чи будуть там необхідні налаштування та залежності. Вони мають змогу створити додаток, упаковати усі залежності та налаштування у певний єдиний образ. Згодом цей образ можна запускати на інших системах, не турбуючись, що програма не запуститься.

Контейнери є важливим інструментом для мікросервісної архітектури.

Це дозволяє розробляти нову функціональність швидше, адже у випадку з монолітною архітектурою зміна якоїсь частини може торкнутися всієї іншої системи.

Docker – це платформа для розробки, доставки та запуску контейнерних програм. Docker дозволяє створювати контейнери, автоматизувати їх запуск та розгортання, керує життєвим циклом. Він дозволяє запускати безліч контейнерів на одній машині.

Популярність Docker продовжує зростати, тому що його підтримує велика спільнота. Платформа здобула широке застосування на хвилі популярності DevOps, швидких конвеєрів доставки та автоматизації.

Віртуалізація в Docker реалізується лише на рівні ОС. Віртуальне середовище запускається прямо з ядра основної операційної системи та використовує її ресурси.

У поставку Docker входять такі компоненти:

- Docker host це операційна система, на яку встановлюють Docker і на якій він працює.
- Docker daemon – служба, яка управляє Docker-об'єктами: мережами, сховищами, образами та контейнерами.

- Docker client – консольний клієнт, за допомогою якого користувачі взаємодіють із Docker daemon і відправляють йому команди, створюють контейнери та керують ними.
- Docker image – це незмінний образ, з якого розгортається контейнер.
- Docker container — розгорнутий та запущений додаток.
- Docker Registry – репозиторій, у якому зберігаються образи.
- Dockerfile – файл-інструкція для збирання образу.
- Docker Compose – інструмент для керування кількома контейнерами. Він дозволяє створювати контейнери та задавати їх конфігурацію.
- Docker Desktop – GUI-клієнт, який розповсюджується по GPL. Безкоштовна версія працює на Windows, MacOS.

Docker спочатку створювався під Linux. Тому на Windows і MacOS запускають віртуальну машину з Linux, а поверх неї - Docker. У macOS використовують VirtualBox, а у Windows - Hyper-V.
З появою WSL та WLS2 з'явилася можливість використовувати Docker Desktop на Windows без необхідності встановлення окремої віртуальної машини. Також з приходом WSL2 було виправлено багато багів у роботі з Docker та його мережами, та продуктивністю ядра та самих Docker контейнерів.

Розробники програмного забезпечення для того щоб встановити бібліотеку мають ознайомитися з порядком її інсталяції на сайті розробника. Після цього іде процес налаштування бібліотеки та її запуску. Під час переходу на іншу бібліотеку, розробники мають видалити, як саму бібліотеку так і залежності пов'язані з нею.

Docker надає інший спосіб. Бібліотеки, фреймворки та постачальники баз даних практично щодня публікують своє програмне забезпечення у вигляді

образів Docker на Docker Hub. Образи завантажуються, розгортаються, управляються і виконуються через Docker, а потім зупиняються або видаляються, не залишаючи жодних слідів в операційній системі.

Завдяки єдиному інтерфейсу управління не потрібно окремих команд. Усе, що потрібно – це вивчити команди Docker: як завантажити / зробити образ, запустити контейнер, зупинити і видалити його. За допомогою Docker можливо запускати будь-яку кількість однакових баз даних у рамках однієї операційної системи. Завдяки ізоляції, якщо щось піде не так, помилка не вплине на операційну систему і нічого не зламає.

Висновки до глави 2

Розділ "Теоретичні аспекти роботи з мікросервісною архітектурою" надав огляд ключових аспектів, які стосуються впровадження мікросервісної архітектури. Мікросервісний підхід забезпечує значні переваги в гнучкості, масштабованості та швидкості розробки додатків у порівнянні з монолітною та n-рівневою архітектурами. Однак, цей підхід також має свої недоліки, пов'язані зі складністю управління розподіленими системами та забезпеченням їхньої узгодженості і стабільності.

Використання контейнеризації за допомогою Docker значно спрощує процес розгортання та управління мікросервісами, забезпечуючи

ізольоване середовище для їх виконання і стабільність у різних середовищах. Docker дозволяє легко переносити контейнери між різними середовищами, спрощує управління залежностями і конфігураціями.

Розділ 3.
Реалізація мікросервісного застосунку на базі фреймворків Spring.

3.1 Загальний огляд застосунку

В рамках даної роботи був розроблений проєкт, який надає повноцінний функціонал для реалізації планувальника завдань. Цей застосунок має назву Scheduler API.

Scheduler API - це проєкт, що надає RESTful API для керування та організації різноманітних завдань, подій та календарних планів. Він

створений з метою полегшення планування робочих процесів, особистих подій, нагадувань та спільної організації часу для користувачів.

Головною метою проекту Scheduler API є надання зручного та ефективного інструменту для планування та керування розкладом. Він допомагає користувачам організовувати свої розклади, події та завдання, надаючи можливість доступу до них через програмний інтерфейс.

Функціональні можливості Scheduler API:

1. Створення та управління календарями.

   Користувачі можуть створювати різні календарі для різних цілей, таких як робочі завдання, особисті події, нагадування тощо.

2. Додавання та редагування подій.

   Інтерфейс API дозволяє додавати нові події до календаря, редагувати їх та встановлювати час та інші параметри.

3. Керування завданнями.

   Користувачі можуть створювати та видаляти завдання. Також є можливість додавання до свого календаря щотижневих завдань, які будуть повторюватися в один і той же час кожен тиждень до певної дати.

4. Спільна робота над календарями.

   Проект надає можливість ділитися календарями та подіями з іншими користувачами, що дозволяє спільно організовувати робочі процеси та події.

5. Автентифікація та авторизація.

Scheduler API забезпечує доступ до календарів та подій, використовуючи механізми аутентифікації та авторизації.

6. Гнучкість та розширюваність.

   Проект розроблений з використанням сучасних технологій, що дозволяє легко розширювати його функціональні можливості та адаптувати під конкретні потреби користувачів.

Отже, Scheduler API – це інструмент для організації робочих процесів та особистих подій. Він дозволяє користувачам ефективно керувати своїм часом, створюючи розклади та завдання за допомогою зручного та легкого використання API. Цей проект допомагає покращити продуктивність та організованість користувачів у їх повсякденних справах.

Застосунок Scheduler API складається із чотирьох основних мікросервісів:

- schedule_service – відповідає за доступ користувача до його розкладу
- user_login_service – відповідає за автентифікацію користувача
- user_register_service – відповідає за реєстрацію користувача
- admin_panel – надає інформацію адміністратору застосунку про його користувачів і створені ними каталоги та групи.

-schedule_service відповідає за надання інформації про розклад користувача за його ідентифікаційним номером. Цей сервіс має наступні endpoints:

- PUT /schedule/get_catalogs/catalog/updateName: Оновлює назву каталогу.
- PUT /schedule/get_catalogs/catalog/remove_admin/{userId}: Видаляє адміністратора з каталогу.
- PUT /schedule/get_catalogs/catalog/groups/enroll_group/{userId}/: Записує групу в каталог.
- PUT /schedule/get_catalogs/catalog/group: Оновлює групу в каталозі.

- PUT /schedule/get_catalogs/catalog/group/edit_event: Редагує подію у групі.
- PUT /schedule/get_catalogs/catalog/add_admin/{userId}: Додає адміністратора до каталогу.
- PUT /schedule/enroll_catalog/{userId}: Записує користувача в каталог.
- POST /schedule/get_catalogs/create/{userId}: Створює новий каталог.
- POST /user/get_catalogs/catalog/group/add_event: Додає подію до групи у каталозі.
- POST /schedule/get_catalogs/catalog/create_group/{userId}: Створює нову групу в каталозі.
- POST /schedule/get_catalogs/catalog/create/{userId}: Створює новий каталог.
- GET /schedule/user_schedule/{userId}: Отримує розклад для користувача в заданому проміжку днів.
- GET /schedule/get_catalogs/{userId}: Отримує каталоги, пов'язані з користувачем.
- GET /schedule/get_catalogs/catalog: Отримує деталі конкретного каталогу.
- GET /schedule/get_catalogs/catalog/groups: Отримує групи в межах каталогу.
- GET /schedule/get_catalogs/catalog/get_admins: Отримує адміністраторів каталогу.
- DELETE /schedule/get_catalogs/catalog/remove/{userId}: Видаляє каталог.
- DELETE /schedule/get_catalogs/catalog/group/remove_event: Видаляє подію з групи в межах каталогу.
- DELETE /schedule/get_catalogs/catalog/group/remove: Видаляє групу з каталогу.

-login-service – призначений для отримання ідентифікаційним номеру користувача за його логіном та паролем. Цей сервіс має один ендпоінт:

- GET /api/login: в якості параметрів запиту передаються логін та пароль. Відповіддю запиту, при коректному логіні та паролі, буде ідентифікаційний номер користувача.

-register-service – призначений для реєстрації нового користувача “Scheduler” застосунку. Цей сервіс має лише один ендпоінт:

- POST /api/register: в якості параметрів запиту передаються логін, пароль та пошта користувача. Метод повертає ідентифікаційний номер користувача, якщо такого логіну та пошти немає в базі даних.

-admin-panel – призначений для отримання адміністратором застосунку інформацію про користувачів та їхні групи і каталоги. Має наступні ендпоінти:

GET /admin/getAllUsersInformation: призначений для отримання інформації про користувачів застосунку “Scheduler”.

Реалізація взаємодії всіх мікросервісів буде показана в подальших пунктах розділу.

3.2 Структура бази даних проекту

Проект Scheduler API використовує базу даних з наступними таблицями: "user_entity", "user_catalog", "catalog", "user_group", "group", та "event". Таблиці "user" та "catalog" забезпечують зберігання інформації про користувачів та їх каталоги, відповідно. Таблиці "user_catalog", "user_group" встановлюють зв'язки між користувачами, каталогами та групами.

Таблиця "catalog” має відношення сама до себе, що надає можливість додавання каталогів всередині батьківського каталогу. Це реалізовано за допомогою поля parent_catalog_id.

Таблиця "group" визначає групи в межах каталогів, а "event" - події, пов'язані з певними групами. Ці таблиці взаємодіють між собою через зовнішні ключі.

Для взаємодії додатку та бази даних було використано фреймворк Spring Data Jpa. Як вже описувалося в розділі 1, Spring Data JPA дозволяє розробникам ефективно працювати з даними, використовуючи об'єктно-реляційне відображення та репозиторії, що дозволяє уникнути прямої роботи з SQL-запитами.

Спочатку, при запуску додатку, Spring Boot автоматично ініціалізує з'єднання з базою даних відповідно до налаштувань, вказаних у файлі конфігурації. Використовуючи інструкції, надані в файлі “application.properties”, Spring Data JPA встановлює з'єднання з базою даних і автоматично створює всі необхідні таблиці на основі оголошених сутностей.

На рис. 3.1 зображено налаштування, які вказані в файлі “application.properties”.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/schedulerDB
spring.datasource.username=root
spring.datasource.password=1111

spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.database=postgresql
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect

spring.jpa.hibernate.ddl-auto=create-drop
```

Рис. 3.1

Налаштуванню “spring.jpa.hibernate.ddl-auto” було встановлено значення “create-drop”, оскільки так було простіше працювати над розробкою

проєкту. Таке рішення зменшувало кількість помилок при взаємодії з БД, а також спрощувало тестування. При закінченні роботи над проектом значення “spring.jpa.hibernate.ddl-auto” було змінено на update.

Коли додаток запущено і готово до взаємодії з базою даних, він може використовувати репозиторії Spring Data JPA для здійснення різноманітних операцій з даними. Репозиторії представляють собою інтерфейси, які розширюють `JpaRepository` або інші інтерфейси Spring Data, і автоматично надають методи для роботи з базою даних.

Наприклад, якщо ми маємо репозиторій `UserRepository` для сутності `User`, ми можемо викликати методи, такі як `save()`, `findById()`, `findAll()`, `delete()`, тощо, для створення, зчитування, оновлення та видалення даних. Spring Data JPA автоматично генерує SQL-запити на основі назв методів у репозиторії та правильно обробляє результати операцій.

У проекті використовується PostgreSQL як система керування базами даних для зберігання інформації про користувачів, каталоги, групи та події.

На рис. 3.2 показано як виглядає структура БД в програмі PgAdmin.

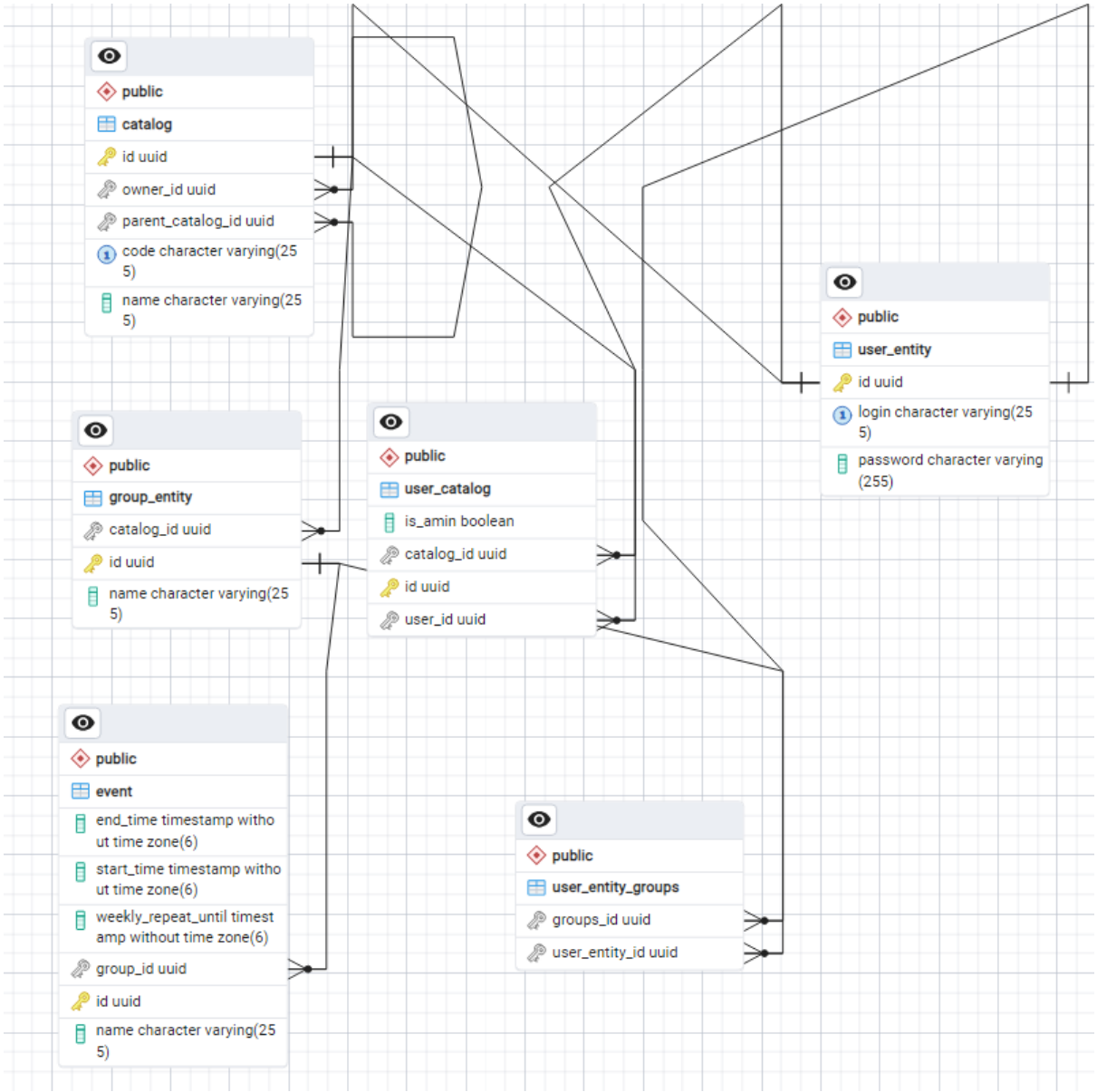

Рис. 3.2

3.3 Порівняльна характеристика RestTemplate та WebClient, як інструментів взаємодії мікросервісів

В сучасних розподілених системах, побудованих на мікросервісній архітектурі, взаємодія між різними компонентами є критично важливою.

RestTemplate та WebClient – це два основних інструменти в екосистемі Spring для забезпечення такої взаємодії.

RestTemplate був стандартним інструментом для взаємодії з REST-сервісами в Spring Framework до версії 5.0. Він надає простий та зручний інтерфейс для виконання HTTP-запитів та обробки їх відповідей.

Основні можливості RestTemplate:

- RestTemplate підтримує виконання HTTP-запитів різних методів, таких як GET, POST, PUT, DELETE тощо.
- Він автоматично обробляє відповіді сервера та надає зручний спосіб розпарсити їх у відповідні об'єкти Java.
- RestTemplate може бути налаштований з різними власностями, такими як таймаути, пул з'єднань, перехоплення запитів тощо.

Приклади використання RestTemplate можна побачити на рис. 3.3 та 3.4.

```
String resourceUrl = "http://localhost:8081/api/login";
ResponseEntity<String> response = restTemplate.getForEntity(
        url: resourceUrl + "?login="+login+"&password="+password,
        String.class
);
```

Рис 3.3

```
String resourceUrl = "http://localhost:8081/api/register";
ResponseEntity<String> response = restTemplate.postForEntity(
        url: resourceUrl+ "?login="+login+"&password="+password,
        request: null,
        String.class);
```

Рис. 3.4

На рис. 3.3 показано взаємодію з мікросервісом входу користувачів в систему. Метод getForEntity робить GET запит за вказаною url адресою та повертає ResponseEntity в тілі якого знаходиться id користувача.
На рис. 3.4 показано id користувача після реєстрації його в системі. Метод postForEntity робить вже POST запит та повертає ResponseEntity в тілі якого знаходиться id користувача.

WebClient був розроблений з виходом Spring 5.0 на заміну RestTemplate та надає більш сучасний та функціональний підхід до взаємодії з мережею. Він побудований на основі реактивного програмування та використовує Project Reactor.

Основні можливості WebClient:

- WebClient надає можливість виконувати HTTP-запити асинхронно, що дозволяє забезпечити ефективне використання ресурсів та високу продуктивність.
- Він інтегрується з реактивними бібліотеками Project Reactor та використовує реактивні типи, такі як Mono та Flux, для обробки відповідей та створення стримів даних.
- WebClient дозволяє легко налаштовувати різні аспекти взаємодії, такі як таймаути, заголовки, обробники помилок тощо.

Отже, якщо зробити реєстацію користувача з обовзяковим підтвердженням електронної пошти (тобто, користувач не буде зареєстрований доки не підтвердить email), то потрібно реалізувати асинхронну взаємодію між мікросервісами використовуючи WebClient.

Простий приклад використання WebClient наведено на рис 3.5.

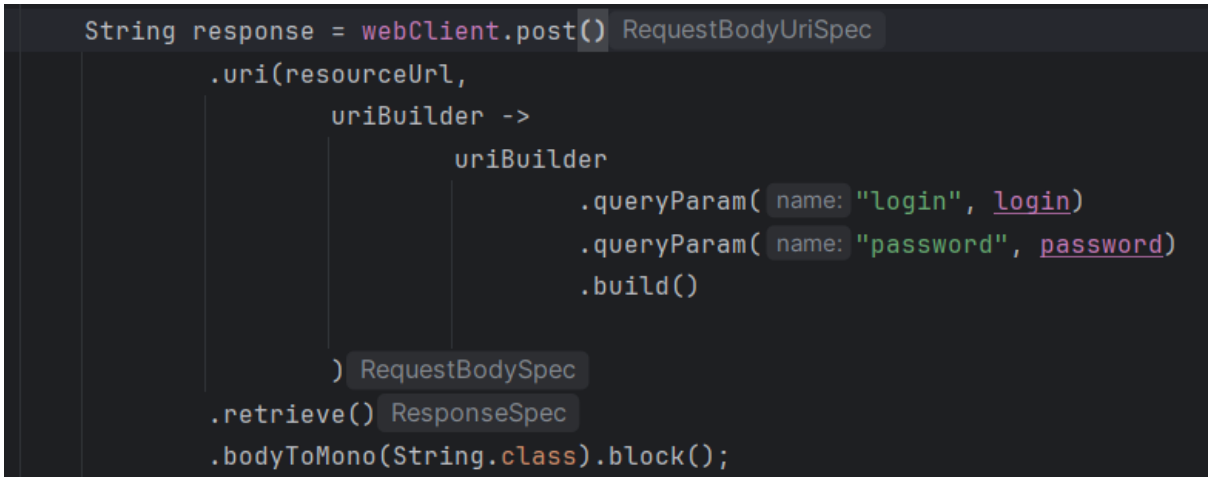

Рис. 3.5

На рис. 3.5 показано виконання того ж запиту того ж, що й на рис. 3.4, проте вже з використанням WebClient.

Підводячи підсумки, варто зазначити, що обидва цих інструменти відкривають можливості для ефективної взаємодії між мікросервісами в розподіленій системі. Проте використання WebClient є більш рекомендованим, оскільки він пропонує асинхронну взаємодію, яка необхідна в багатьох ситуаціях. WebClient також надає більш гнучкий та легко налаштовуваний підхід до взаємодії, дозволяючи зручно налаштовувати таймаути, обробники помилок та інші параметри.

3.4 Налаштування виявлення служб (Service Discovery) використовуючи Netflix Eureka.

Як вже було зазначено в першому розділі, для належної взаємодії мікросервісів необхідно мати ефективний механізм виявлення та реєстрації

служб. Netflix Eureka – це інструмент, який дозволяє це зробити швидко та надійно.

Початковим кроком є додавання залежності “spring-cloud-dependencies” до тегу <dependencyManagement> у файлі pom.xml батьківського проєкту, як показано на рис. 3.6. Це необхідно для того щоб всі мікросервіси мали також цю залежність.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-dependencies</artifactId>
    <version>2023.0.1</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

Рис. 3.6

Наступним кроком було створення нового Spring Boot застосунку під назвою “eureka_server” та налаштування його як сервера реєстрації та виявлення служб. Для цього було додано залежність “spring-cloud-starter-netflix-eureka-server” у файлі pom.xml нового модуля.

Також, для того щоб сервер працював коректно було додано анотацію @EnableEurekaServer до класу запуску застосунку, як це показано на рис. 3.7.

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServerApplication {
    public static void main(String[] args) { SpringApplication.run(DiscoveryServerApplication.class, args);
}
```

Рис. 3.7

Важливо не забути додати основні налаштування в файл “application.properties” або “application.yml” , так як показано на рис. 3.8

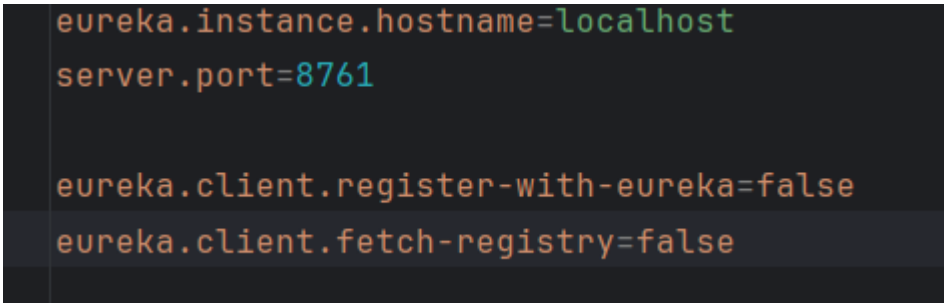

Рис. 3.8

Налаштування “eureka.client.register-with-eureka=false” потрібне для того, щоб Eureka сервер не вважав себе самого за клієнтський застосунок. Воно контролює, чи реєструється цей клієнт сам і, отже, стає видимим. Само по собі це не означає, що цей клієнт збирається отримувати інформацію про кінцеві точки інших служб і, отже, матиме змогу підключитися до них.

Налаштування “eureka.client.register-with-eureka=false” вказує на те, що тепер клієнтський застосунок не зможе підключитися до серверів Eureka, щоб завантажити інформацію про кінцеві точки інших служб. Цю дію можна зробити без реєстрації.

“server.port=” вказує на порт на якому буде знаходитися сервер. “eureka.instance,hostname=” вказує ім’я хоста сервера.

Після налаштування серверу реєстрації та виявлення служб. Потрібно зробити налаштування клієнтських мікросервісів, які будуть взаємодіяти з сервером. Тому до всіх мікросервісів було додано залежності, які показані на рис 3.9.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-netflix-eureka-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-client</artifactId>
</dependency>
```

Рис. 3.9

Ці залежності вказують Eureka серверу, що це клієнтський застосунок.

Після впровадження всіх необхідних налаштувань, можемо запустити всі міросервіси та перейти на порт серверу Eureka, який в даному випадку є 8761. На рис. 3.10 показано як це виглядає.

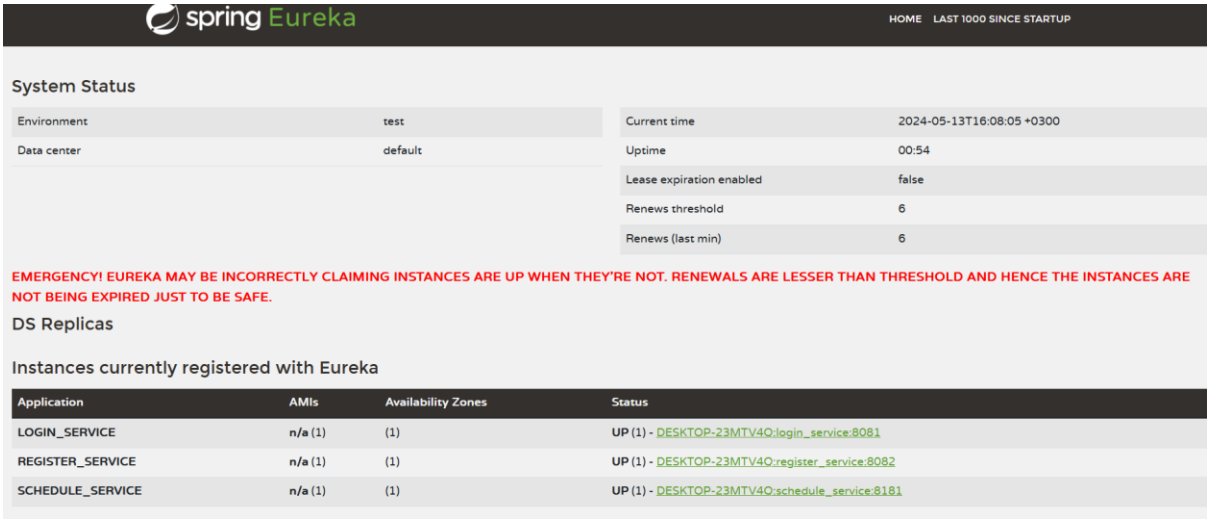

Рис. 3.10

На рис. 3.10 можемо бачити інформацію про налаштування серверу, а також які служби до нього підєднанні. В даному випадку до серверу Eureka під'єднані три мікросервіси, а саме: LOGIN_SERVICE, REGISTER_SERVICE, SCHEDULE_SERVICE. Як ми можемо бачити, кожен сервіс має лише один екземпляр на сервері. Якщо припустити таку ситуацію, що сервіс надання розкладу користувачам (SCHEDULE_SERVICE) перегружений і потрібно його робити

масштабування, тобто запускати ще один SCHEDULE_SERVICE. Змоделюємо ситуацію за якої створюється ще одни сервіс з портом 8182. Це можна зробити за допомогою інструментів Intellij Idea, як показано на рис. 3.11.

Рис. 3.11

В “program arguments” було змінено порт з 8181 на 8182.

Тепер, при запуску ще й цього додаткового екземпляру сервісу, отримуємо, що SCHEDULE_SERVICE стало два, що можемо бачити на рис. 3.12.

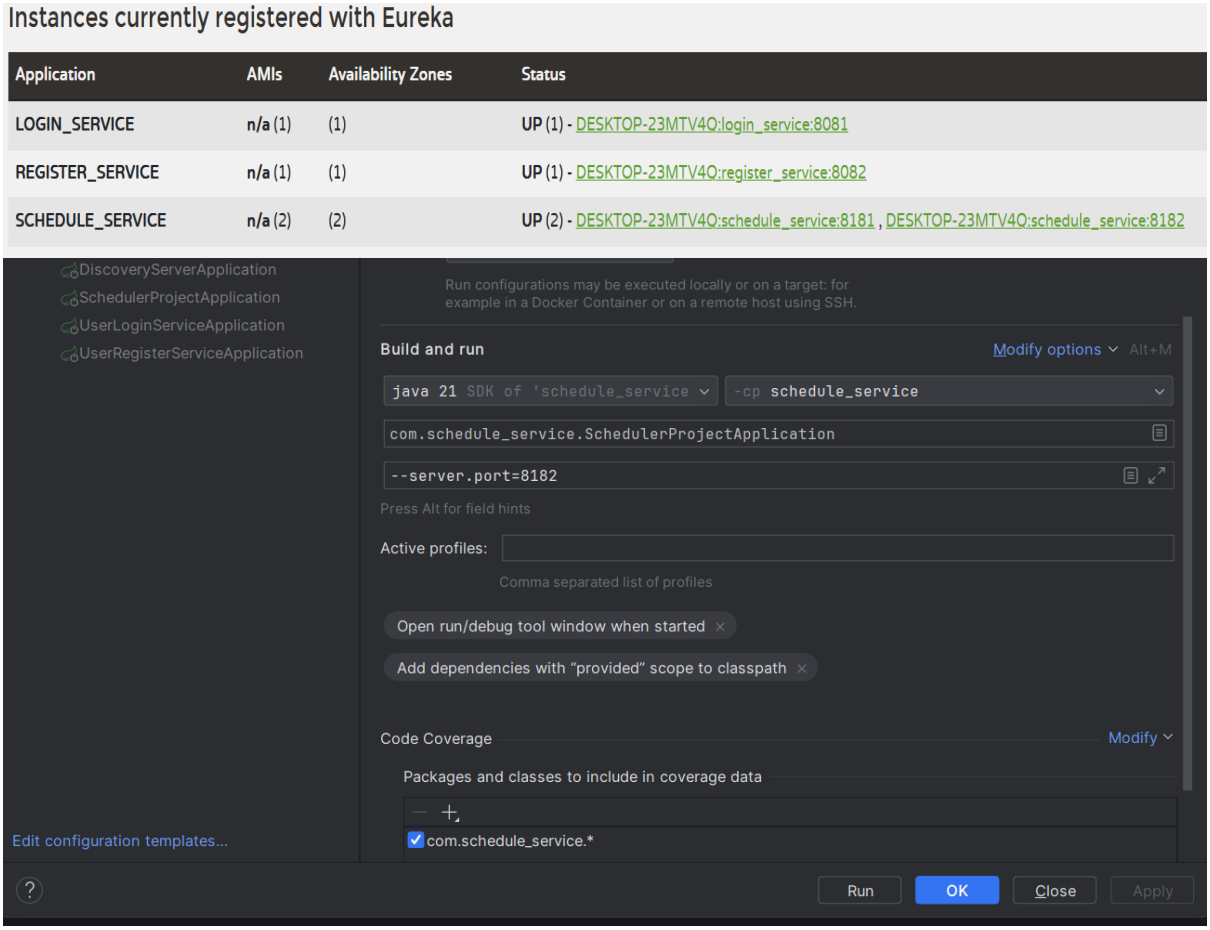

Рис. 3.12

Також можливо змінити налаштування таким чином щоб кожен екземпляр програми запускався з рандомним портом. Для цього потрібно змінити значення “server.port” з конкретного порта на “0”, тоді, при запуску сервісу, Spring сам присвоїть йому рандомний не зайнятий порт.

Тепер, після налаштування Eureka серверу, є можливість не вказувати точний порт при відправці запиту на сервіс. Вказуючи замість імені хоста та порта лише назва сервісу. Приклад можете бачити на рис 3.13.

```
String resourceUrl = "http://REGISTERSERVICE/api/register";
```

Рис. 3.13

Щоб Spring знав якому конкерному сервісу з двох однакових запущених адресувати запит, потрібно додати анотацію @LoadBalanced над конфігурацією біна WebClient. Цей метод вказаний в офіційній документації до версії spring-cloud-dependencies 2022.0.3. Проте з новою залежністю 2023.0.1, цей спосіб не спрацював. Видало помилку, що такого імені хоста не існує, хоча Eureka server зареэстрував цей хост.

Було знайдено вихід із ситуації шляхом визначення наступного біна:

```
ReactorLoadBalancerExchangeFilterFunction lbFunction
```

Додавання його в WebClient.Builder метод під назвою filter() вирішило проблему. Тепер функція отримання відповіді від мікросервісу реєстації виглядає як показано на рис. 3.14.

Рис. 3.14

Отже, було реалізовано технологію Service Discovery за допомогою Netflix Eureka. Тепер є можливим масштабування мікросервісів шляхом створення додаткових екземплярів сервісів без вказування потрібного порта. Також, було розглянуто балансування навантаження та реалізовано його за допомогою ReactorLoadBalancerExchangeFilterFunction. Тепер при існуванні двох та більше екземплярів сервісу, Spring буде авоматично балансувати навантаження між ними.

3.5 Spring Cloud Gateway

Spring Cloud Gateway – це інструмент для спрямовування трафіку в мікросервісній архітектурі. Він виступає як API шлюз, надаючи можливість маршрутизації запитів, обробки навантаження, виконання перехоплення запитів та інші корисні функції. Нижче будуть розглянуті кроки для практичної реалізації Spring Cloud Gateway.

По перше, необхідно створити новий Spring Boot проєкт. Було створено проєкт під назвою api-getaway. До цього проєкту були додані наступні залежності: spring-cloud-starter-gateway, spring-cloud-netflix-eureka-client,

spring-cloud-starter-loadbalancer, spring-cloud-commons, spring-boot-starter-security, spring-security-oauth2-resource-server, spring-boot-starter-oauth2-client. Всі додані залежності можна бачити на рис. 3.15.

Рис. 3.15

На цьому етапі бібліотеки пов'язані з безпекою застосунку, а саме spring-boot-starter-security, spring-security-oauth2-resource-server та spring-boot-starter-oauth2-client, не є необхідними. Проте вони будуть потрібними при

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-netflix-eureka-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-commons</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-resource-server</artifactId>
</dependency>
```

впровадженні аутентифікації користувача, яка буде розглянута в подальшому. Другим кроком, при впровадженні Spring Cloud Gateway, є

додавання необхідних налаштувань в applications.properties або application.yml (в даному випадку був використаний файл з розширенням .yml). Приклад таких налаштувань можна побачити на рис. 3.16.

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: login-service
          uri: lb://register-service/
          predicates:
            - Path=/api/register
        - id: login-service
          uri: lb://login-service/
          predicates:
            - Path=/api/login
        - id: schedule-service
          uri: lb://schedule-service/
          predicates:
            - Path=/schedule/**
      discovery:
        locator:
          enabled: true
          lower-case-service-id: true
    loadbalancer:
      configurations: health-check
  application:
    name: gateway
```

Рис. 3.16

Розглянемо окремо кожне налаштування:

- spring.cloud.getaway.routes – це масив шляхів за якими відбувається маршрутизація, де “path” позначає шлях який вводить користувач API для необхідного запиту, а “uri” позначає на який сервіс буде йти цей

запит. Для прикладу візьмемо елемент масиву з рис. 3.16, який має назву(id) “login-service”. Також, встановимо порт застосунку, нехай він буде 8090. Отже, після запуску програми “api-getaway” ми можемо зробити вхід через мікросервіс “login-service” не викликаючи його напряму. До прикладу, зробимо вхід для користувача “admin” з паролем “1111”. Для цього в браузері необхідно буде ввести наступний адрес: http://localhost:8090/api/login?login=admin&password=1111.

- spring.cloud.getaway.discovery.locator.enabled: true – це налаштування дозволяє використовувати кінцеві точки Eureka серверу, такі як “lb://“login-service”, “lb://“register-service” та інші.

- spring.cloud.getaway.discovery.locator.lower-case-service-id: true – дозволяє використовувати імена сервісів, які підключені до Eureka серверу, в нижньому регістрі. Якщо значення цього налаштування було б false, то необхідно було б писати всі імена екземплярів сервісів у верхньому регістрі, до прикладу: “LOGIN-SERVICE”.

- spring.cloud.loadbalancer.configurations: health-check – це налаштування встановлює перевірку стану балансувальника навантаження. Тобто вона слідкує за тим щоб loadbalancer постійно був доступний.

- spring.application.name: – встановлює назву застосунку.

Також не варто забувати про налаштування ApiGetaway як клієнта Eureka серверу. Налаштування, які були використані в Scheduler API, показані на рис. 3.17.

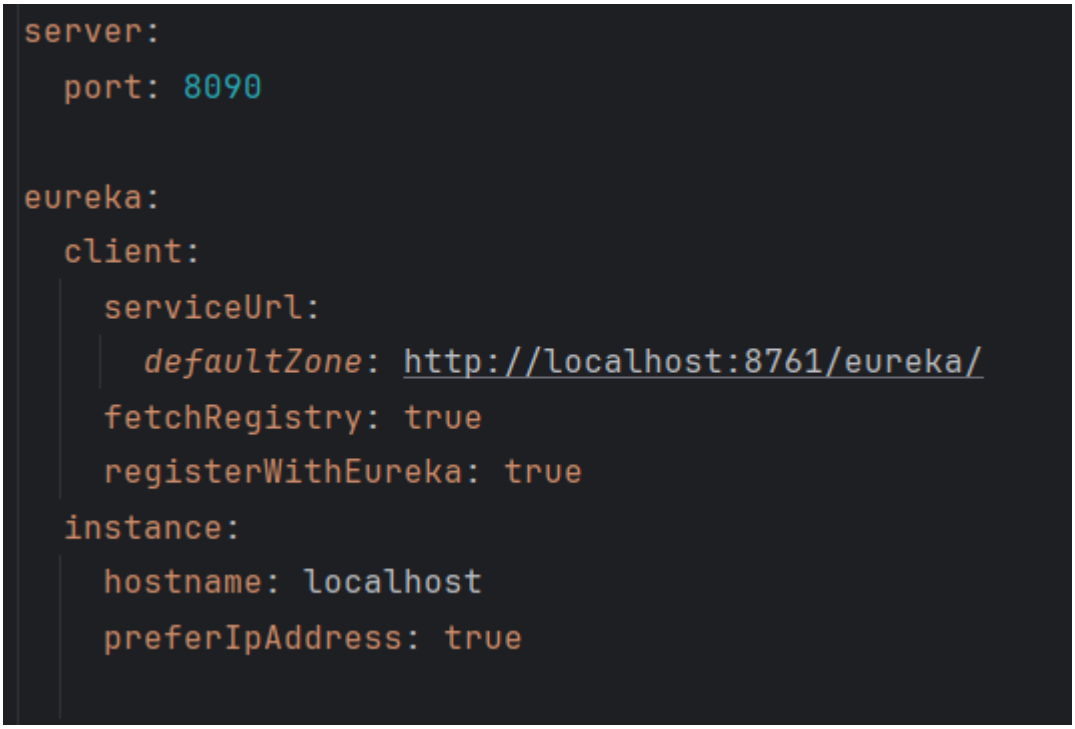

Рис. 3.17

Детальніше описувалися ці налаштування в пункті 3.4, де йшла мова про налагодження Eureka серверу та його клієнтів.

Налаштування маршрутизації можна реалізувати за допомогою визначення біна RouteLocator. На рис. 3.18 показано створення цього біна в класі конфігурації та налаштування маршрутизації.

```java
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
            .route( id: "login", r -> r.path("/api/login")
                    .uri("lb://login-service"))

            .route( id: "register", r -> r.path("/api/register")
                    .uri("lb://register-service"))
            .route( id: "schedule-service", r -> r.path("/schedule/**")
                    .uri("lb://schedule-service"))
            .route( id: "admin-panel", r -> r.path("/admin/**")
                    .uri("lb://admin-panel"))
            .build();
}
```

Рис. 3.18

Після проведення всіх необхідних налаштувань роботи застосунку, подивимося на приклади його роботи. Як вже зазначалося, тепер для доступу до мікросервісів не потрібно знати порт кожного з них, достатньо лише знати його для api getaway. В даному випадку, у застосунку Scheduler Api було використано порт 8090.

Отже, тепер, щоб отримати userID за логіном та паролем, потрібно перейти за адресою http://localhost:8090/api/login?login=admin&password=1111 (як вже зазначалося у прикладі вижче). Відповіддю на це буде UUID цього користувача, як показано на рис. 3.19. На цьому рисунку показано результат відповіді з програми Postman.

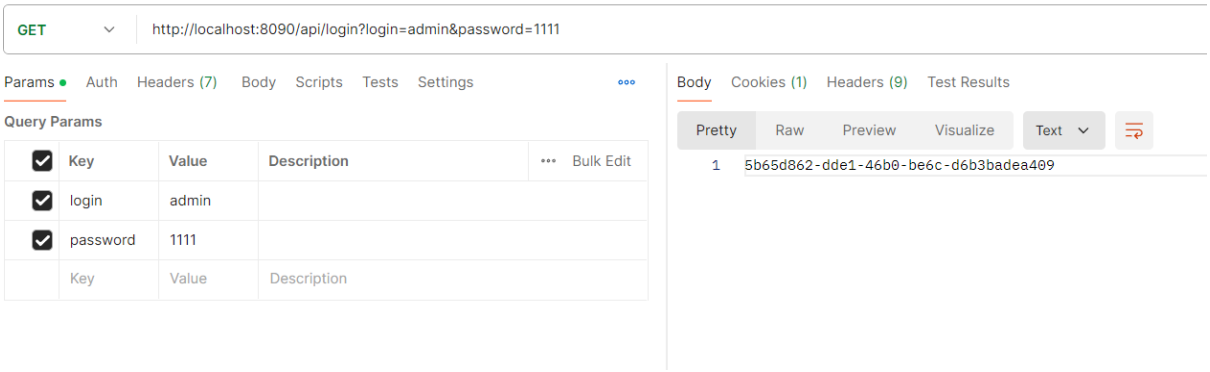

Рис. 3.19

На рис. 3.17 помітно, що використовується порт 8090, який зазначений для Getaway застосунку. З цього можемо зробити висновки, що програма працює коректно.

3.6 Забезпечення безпеки мікросервісів використовуючи Spring Security та Keycloak

Забезпечення безпеки мікросервісів є однією з ключових вимог при розробці розподілених систем. У цьому пункті буде розглянуто застосування фреймворку Spring Security та його інтеграцію з Keycloak для забезпечення безпеки мікросервісів.

Spring Security — це фреймворк, основною задачею якого є забезпечення як автентифікації, так і авторизації програм Java. Як і в усіх проектах Spring, справжня сила Spring Security полягає в тому, наскільки легко його можна розширити, щоб відповідати користувацьким вимогам.

Keycloak – це відкритий сервер управління доступом, який підтримує сучасні стандарти безпеки, такі як OAuth 2.0, OpenID Connect та SAML 2.0. Він надає можливість централізовано керувати аутентифікацією та авторизацією користувачів, що значно спрощує налаштування безпеки у розподілених системах.

Розглянемо покрокове налаштування Keycloak на прикладі його застосування в застосунку Scheduler API.

Крок 1. Встановлення Keycloak:

Оскільки передбачається подальша контейнеризація та розгортання мікросервісного застосунку за допомогою допомогою Docker контейнерів, тому вирішено було застосовувати Keycloak зразу за допомогою контейнеру з його образом.

Було написано Docker Compose файл з розширенням yml у якому створювався контейнер mysql бази даних для зберігання даних з Keycloak. Також створюється контейнер з образом “quay.io/keycloak/keycloak:24.0.1” та встановлюється зв’язок між контейнерами за допомогою “depends_on”. На рис. 3.20 показано вміст файлу “docker-compose.yml” на даний. Також на ньому можна помітити як прописаний меппінг портів. Підключення до Keycloak відбувається за портом 8383. Файлу “docker-compose.yml” був розміщений в корінь проєкту “api-getaway”.

Для завантаження образів та створення контейнерів потрібно запустити термінал в папці в якій розміщений “docker-compose” файл та ввести команду: docker compose up -d. “-d” означає, що контейнери будуть запущені в режимі “detach”, тобто їхні логи не будуть показуватися в терміналі та можна буде далі користуватися цим терміналом. Також необхідною умовою запуску команди є працюючий Docker Engine, в операційній системі Windows 11 це можна зробити запустивши програму Docker Desktop.

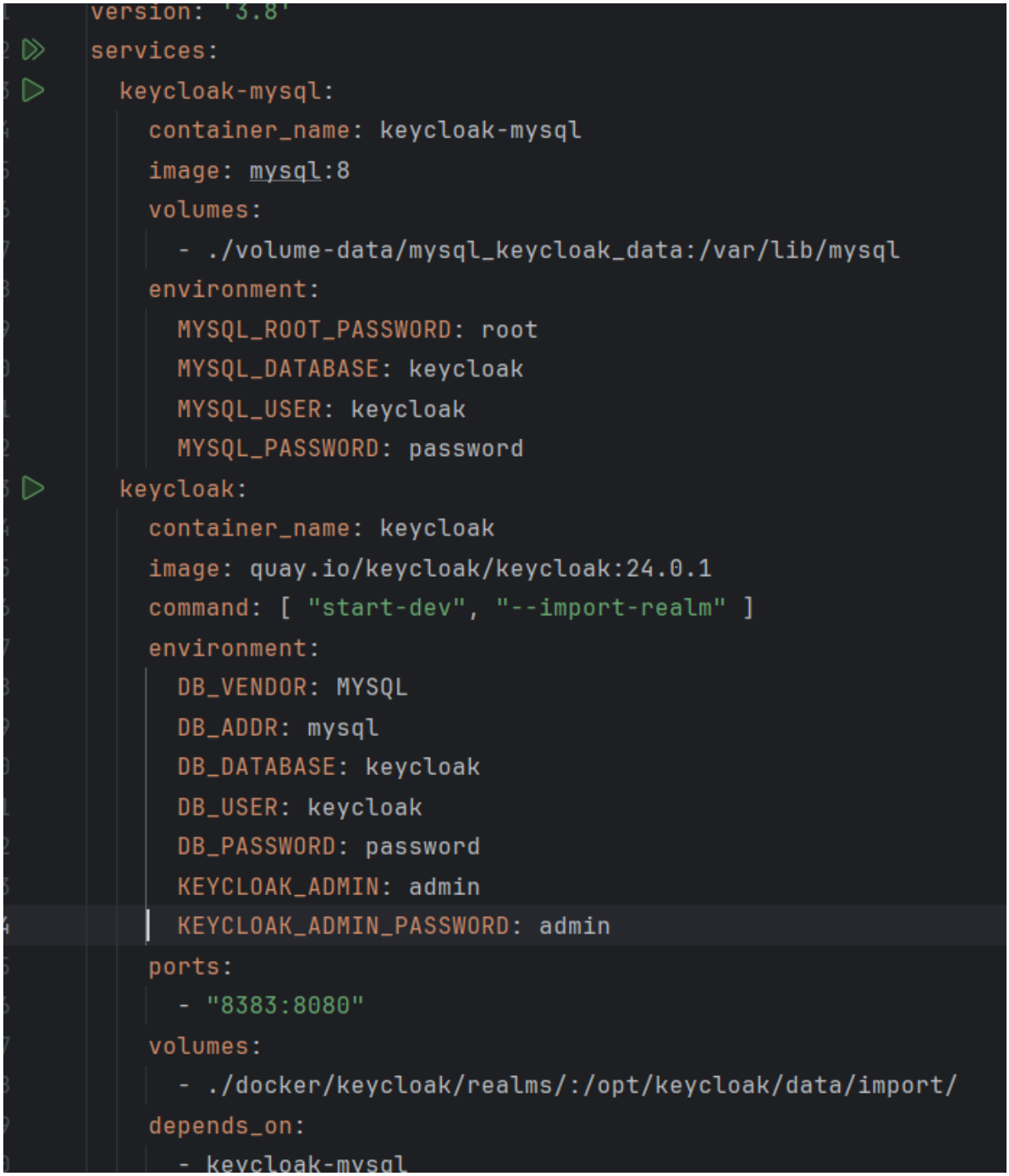
version: '3.8'
services:
keycloak-mysql:
container_name: keycloak-mysql
image: mysql:8
volumes:
- ./volume-data/mysql_keycloak_data:/var/lib/mysql
environment:
MYSQL_ROOT_PASSWORD: root
MYSQL_DATABASE: keycloak
MYSQL_USER: keycloak
MYSQL_PASSWORD: password
keycloak:
container_name: keycloak
image: quay.io/keycloak/keycloak:24.0.1
command: ["start-dev", "--import-realm"]
environment:
DB_VENDOR: MYSQL
DB_ADDR: mysql
DB_DATABASE: keycloak
DB_USER: keycloak
DB_PASSWORD: password
KEYCLOAK_ADMIN: admin
KEYCLOAK_ADMIN_PASSWORD: admin
ports:
- "8383:8080"
volumes:
- ./docker/keycloak/realms/:/opt/keycloak/data/import/
depends_on:
- keycloak-mysql

Рис. 3.20

Крок 2. Налаштування Keycloak.

Налаштування проводилося в користувацькому інтерфейсі сервісу Keycloak. Спочатку було створено новий “realm” (з англ. сфера або галузь) під назвою “scheduler-microservices”. Потім до нього було додано користувача з ім’ям “admin1”. Конфігурація можливостей цього користувача показана на рис. 3.21. Також на цьому рис. видно як виглядає користувацький інтерфейс Keycloak. До речі, вхід до налаштувань Keycloak здійснюється за допомогою параметрів “KEYCLOAK_ADMIN” та “KEYCLOAK_ADMIN_PASSWORD” (рис. 3.18), які були вказані в docker-compose файлі.

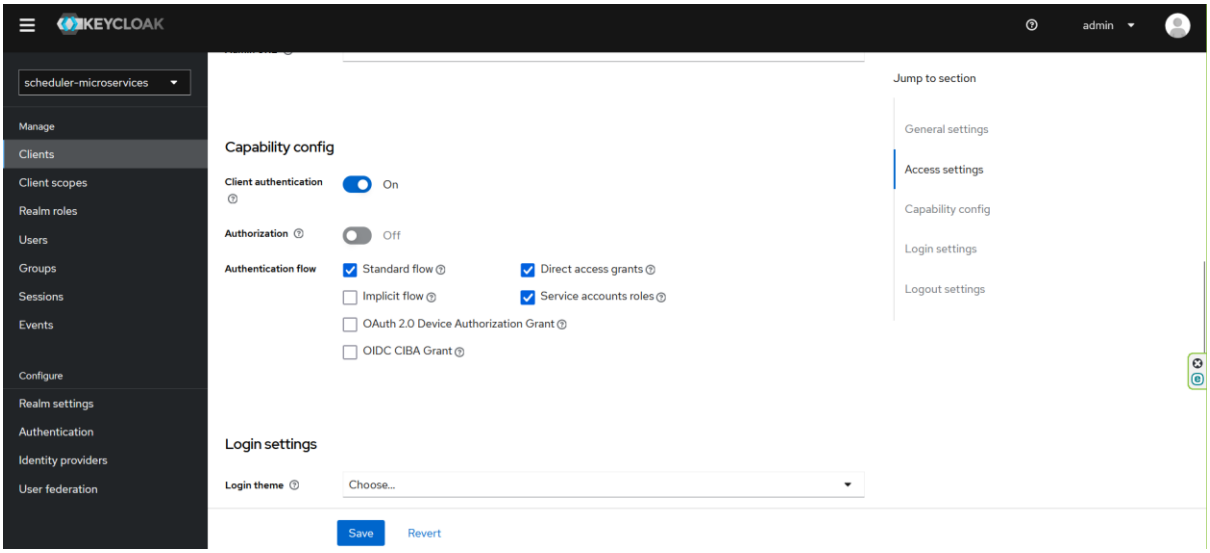

Рис. 3.21

Крок 3. Налаштування роботи Spring Security в проєкті “api-getaway”.

Взаємодія застосунку “api-getaway” та Keycloak буде відбуватися за допомогою“oauth2 resourceserver” та “oauth2 client”. Налаштування цих

двох бібліотек продемонстровано на рис. 3.22. Налаштування записуються в файл “application.yml”.

```yaml
security:
  oauth2:
    resourceserver:
      jwt:
        jwk-set-uri: http://localhost:8383/realms/scheduler-microservices/protocol/openid-connect/certs
    client:
      provider:
        keycloak:
          issuer-uri: http://localhost:8383/realms/scheduler-microservices
      registration:
        spring-with-test-scope:
          provider: keycloak
          client-id: admin1-id
          client-secret: dxNey1Zrw0FgPHDyptIeglKVYu58Jd1V
          authorization-grant-type: authorization_code
          scope: openid
```

Рис. 3.22

Крок 4. Налаштування доступів в SecurityWebFilterChain.

Було використано SecurityWebFilterChain замість SecurityFilterChain, оскільки використовувався реактивний стек Spring, в тому числі фреймворк “spring-cloud-starter-gateway”. Саме налаштування доступів було реалізовано доволі просто. Всі шляхи, до яких має доступ лише у адміна, тобто “/admin” та “/admin/**”, були позначені як ті до яких потрібна аутентифікація адміністратора, яка проходить через Keycloak. Це було зроблено в класі конфігурації “SecurityConfig”. На рис. 3.23 показано код класу SecurityConfig.

Також в налаштуваннях безпеки застосунку вказано, що аутентифікація буде реалізована за допомогою протоколу авторизації “OAuth2”.

Анотація “@EnableWebFluxSecurity” вказує Spring, що потрібно виключити можливість налаштування параметрів безпеки в цьому класі конфігурації.

```java
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http.authorizeExchange(auth -> auth
                        .pathMatchers("/admin", "/admin/**").authenticated()
                        .anyExchange().permitAll())
                .oauth2Login(withDefaults())
                .oauth2ResourceServer((oauth2) -> oauth2.jwt(withDefaults()));
        http.csrf(ServerHttpSecurity.CsrfSpec::disable);
        return http.build();
    }

}
```

Рис. 3.23

Налаштування фреймворку Spring Security та його інтеграцію з Keycloak було успішно реалізовано. Тепер можна перейти до тестування безпеки застосунку.

Як вже зазначалося, обов'язкове проходження аутентифікації потрібне для шляхів "/admin" та "/admin/**. Тому, якщо намагатися отримати доступ до /admin/getAllUsersInformation за шляхом: http://localhost:8090/admin/getAllUsersInformation, то отримаємо код відповіді 400, як показано на рис. 3.24.

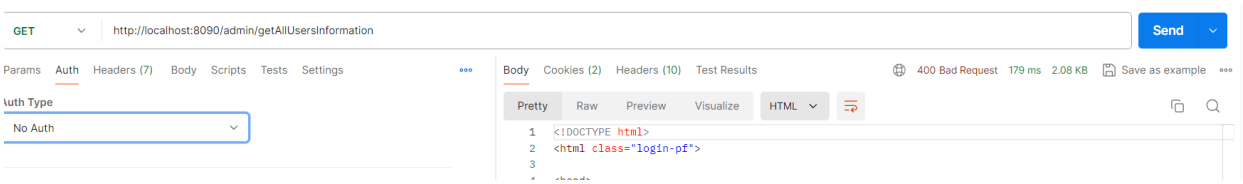

Рис. 3.24

Якщо змінити параметр “Auth type” в програмі Postman з “No auth” на “OAuth 2.0”, але вказати неправильний токен, то код відповіді буде вже 401 (не авторизований).

Отже, для доступу адміністратора застосунку до сервісу, спочатку необхідно отримати jwt токен від Keycloak. Для цього необхідно, як показано на рис. 3.25, надати Client ID, Client Secret та Access Token URL. Access Token URL можна отримати за адресою “http://HOSTNAME:PORT/realms/REALM_NAME/.well-known/openid-configuration” (у випадку роботи з Scheduler Api: http://localhost:8383/realms/scheduler-microservices/.well-known/openid-configuration).

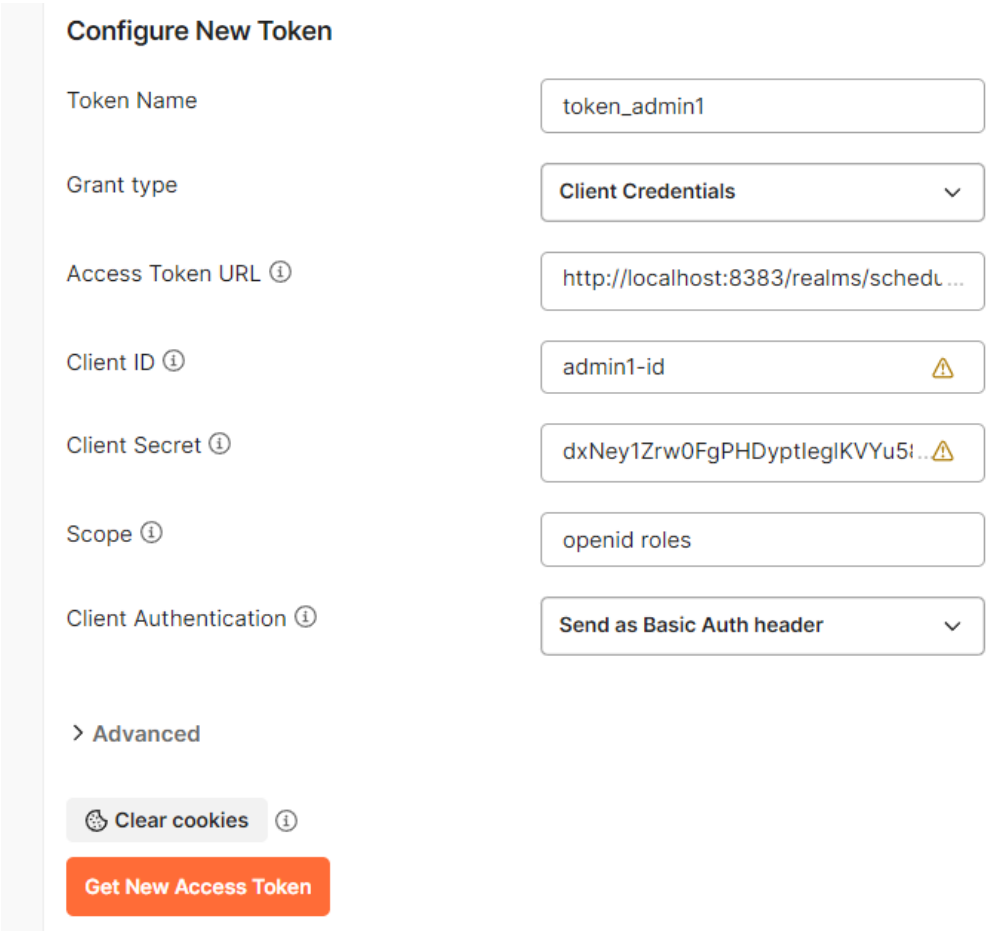

Рис. 3.25

Після успішної генерації токену можна його використовувати певний час для отримання необхідної інформації з серверу.

На рис. 3.26 показано відповідь з сервісу панелі адміністратора на запит після успішної авторизації за через OAuth 2.0.

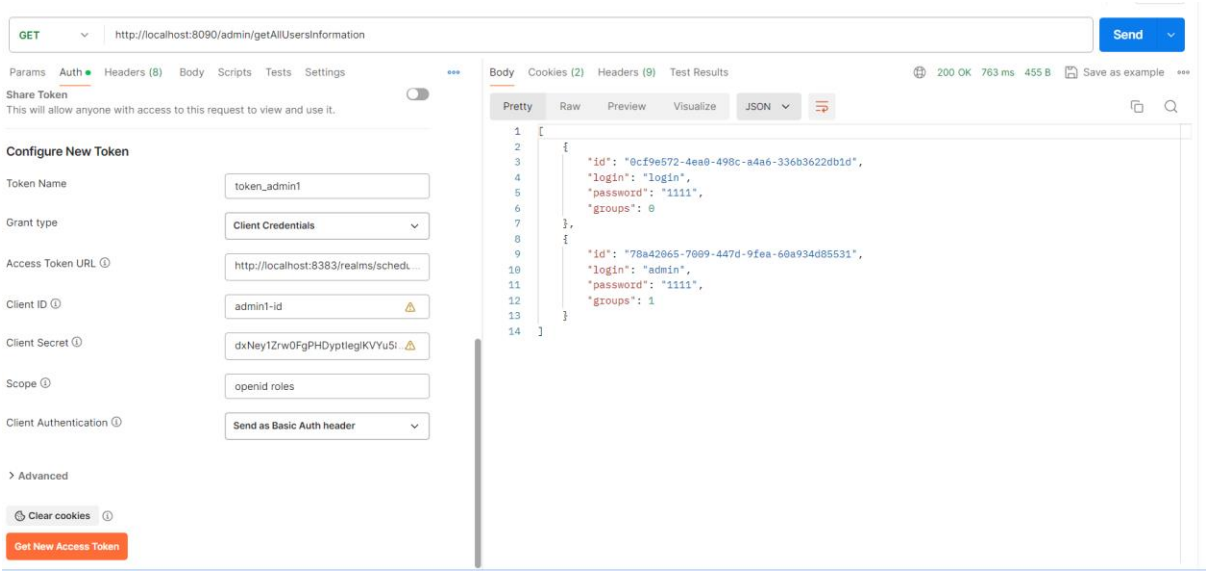

Рис. 3.26

Підсумовуючи все вищесказане, вийшло реалізовувати механізм обмеження доступу до сервісу панелі керування адміністратора. Це вдалося зробити використавши та коректно налаштувавши Spring Security та зробити її інтеграцію з програмою Keycloak.

Висновки до глави 3

В цьому розділі було детально розглянуто практичні аспекти розробки мікросервісного застосунку Scheduler API, використовуючи сучасні інструменти та технології Spring. Практична реалізація Scheduler API на базі фреймворків Spring демонструє ефективність та переваги мікросервісної архітектури.

Порівняння RestTemplate та WebClient показало, що WebClient є більш сучасним і функціональним інструментом для взаємодії між мікросервісами. WebClient підтримує асинхронні операції, що дозволяє значно підвищити продуктивність і масштабованість додатків, порівняно з синхронним підходом, який використовує RestTemplate. Використання WebClient є більш ефективним у контексті мікросервісної архітектури, де асинхронна взаємодія між сервісами є ключовою для досягнення високої продуктивності.

Налаштування виявлення служб за допомогою Netflix Eureka забезпечило динамічне виявлення та реєстрацію мікросервісів, що значно спростило управління розподіленими системами. Використання Eureka дозволило автоматично виявляти нові сервіси, балансувати навантаження та забезпечувати стійкість системи до збоїв, що підвищило надійність та ефективність роботи Scheduler API.

Використання Spring Cloud Gateway надало потужні можливості для маршрутизації та управління трафіком. Використання цього інструменту дозволило забезпечити централізоване управління доступом до

мікросервісів, включаючи балансування навантаження, маршрутизацію запитів та забезпечення безпеки.

Забезпечення безпеки застосунку було реалізовано за допомогою таких інструментів як: Spring Security та Keycloak. Їхнє використання дозволяє ефективно реалізовувати механізми аутентифікації та авторизації у мікросервісній архітектурі. Використання цих інструментів забезпечило надійний захист даних і доступу до сервісів, що підвищує загальний рівень безпеки системи.

Розділ 4.
Контейнеризація та моніторинг роботи мікросервісного застосунку.

4.1. Контейнеризація застосунку

Контейнеризація застосунку за допомогою Docker є ефективним способом ізоляції додатків і їх середовищ виконання, що забезпечує можливість переносу додатків на іншу платформу із збереженням коректної їх роботи. Docker Compose, як інструмент для роботи з Docker, надає можливість описати багатоконтейнерні додатки в одному файлі і легко управляти ними. Зараз буде розглянуто процес контейнеризації мікросервісного додатоку Scheduler API з використанням Docker Compose.

Зазвичай для використання Docker Compose потрібно для кожного мікросервісу прописати Dockerfile та створити за них образ цього застосунку. Приклад Dockerfile для Java додатку показаний на рис. 4.1.

```
FROM openjdk:17

COPY target/*.jar app.jar

ENTRYPOINT ["java","-jar","/app.jar"]
```

Рис. 4.1

Проте у цій роботі був використаний інший підхід. Було використано інструмент “jib”. Jib — це Java-інструмент із відкритим вихідним кодом, підтримуваний Google для створення образів Docker програм Java. Це

спрощує контейнеризацію, оскільки з ним нам не потрібно писати докер-файл. Також, Jib є у вигляді плагіну Maven.

До батьківського pom файлу застосунку Scheduler API було додано плагін Jib, як показано на рис. 4.2.

```
<plugin>
  <groupId>com.google.cloud.tools</groupId>
  <artifactId>jib-maven-plugin</artifactId>
  <version>3.4.1</version>
  <configuration>
    <from>
      <image>eclipse-temurin:21</image>
    </from>
    <to>
      <image>registry.hub.docker.com/viktorvavd/${project.artifactId}</image>
      <auth>
        <username>viktorvavd</username>
        <password>[redacted]/password>
      </auth>
    </to>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>dockerBuild</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Рис. 4.2

Додавши такий плагін з такими конфігураціями, отримуємо, що створюються образи кожного мікросервісу та додаються на docker репозиторій під назвою “viktorvavd”. При створені кожного окремого образу буде братися ім’я цього мікросервісу, для цього було вказано “${project.artifactId}”.

Запустити створення образів можна за допомогою наступної команди в кореневому каталозі проекту:

mvn compile jib:build

Після створення образів можна переходити до написання Docker Compose файлу. Потрібно створити файл під назвою “docker-compose.yml” у кореневій папці проекту. У цьому файлі вказуються всі необхідні сервіси, та вказується образи, які потрібні для створення контейнерів цих сервісів.

На рис. 4.3 показано, як у Docker Compose файлі задається створення контейнеру сервісу “schedule-service” з образу, який був взятий з репозиторія “Docker Hub”.

```
schedule-service:
  container_name: schedule-service
  image: viktorvavd/schedule-service:latest
  environment:
    - SPRING_PROFILES_ACTIVE=docker
    - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres-db:5432/schedulerDB
  depends_on:
    - postgres-db
    - discovery-server
    - api-gateway
```

Рис. 4.3

Задання параметрів створення контейнеру бази даних Postgres показано на рис. 4.4. На рис. 4.3 видно, що встановлюється залежність контейнеру schedule-service від postgres-db контейнера. Також при підключені до бази даних вказується порт 5432, той же порт, що і при створені контейнера бази даних.

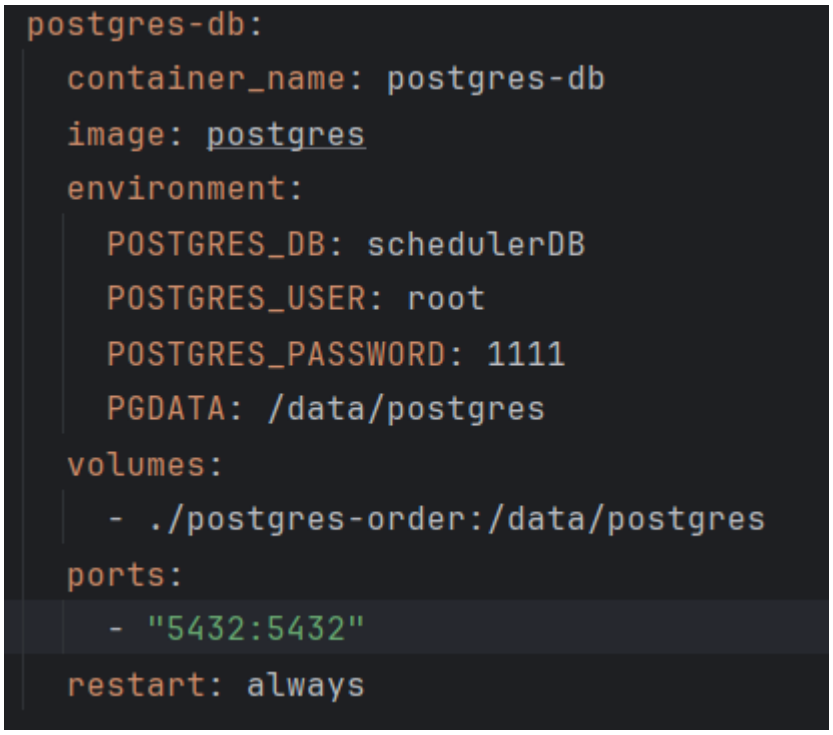

Рис. 4.4

Зміст docker-compose.yml файлу є у додатку А.

Для створення та запуску всіх контейнерів потрібно в терміналі ввести команду:

docker compose up

Після запуску всіх контейнерів у програмі Docker Desktop можна бачити їх роботу, як показано на рис. 4.5. Також натиснувши на кожен з них можна бачити логи цього мікросервісу.

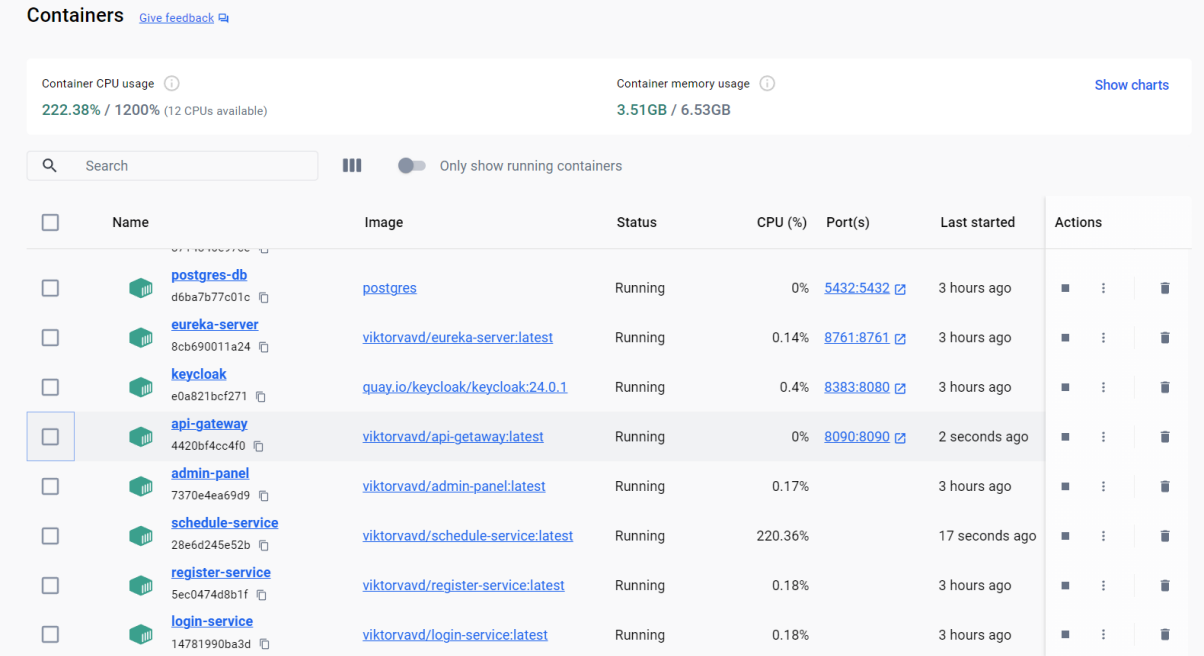

Рис. 4.5

4.2 Моніторинг мікросервісів використотвуючи Prometheus та Grafana

Як вже зазначалося, можна дивитися логи мікросервісів в контейнерах використовуючи програму Docker Desktop. Проте використовувати її не завжди комфортно.

Для моніторингу роботи мікросервісів всередині контейнерів часто використовують інструменти Prometheus та Grafana.

Prometheus — це набір інструментів для моніторингу та оповіщення систем із відкритим вихідним кодом, створений у SoundCloud.

Grafana Cloud — це високодоступна, швидка, повністю керована платформа OpenSaaS для журналювання, метрики, трасування та профілювання, яка також забезпечує керування інцидентами та службу моніторингу додатків.

Для їх підключення потрібно зробити наступні кроки:

Крок 1. Додавання залежностей до мікросервісів.

Потрібно додати дві нових залежності, а саме: “micrometer-registry-prometheus” та “spring-boot-starter-actuator” (Рис. 4.6).

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId
    <scope>runtime</scope>
</dependency>
```

Рис. 4.6

Крок 2. Додавання необхідних налаштувань.

До налаштувань у файлі “application.properties” до кожного модуля необхідно додати:

management.endpoints.web.exposure.include= prometheus

Крок 3. Додавання опису контейнерів Prometheus та Grafana до Docker Compose файлу.

Отже, після додавання всіх необхідних налаштувань до мікросервісних застосунків, потрібно додати опис ще контейнерів Prometheus та Grafana до файлу “docker-compose.yml” (рис. 4.7, 4.8).

```yaml
# Prometheus
prometheus:
  image: prom/prometheus:v2.37.1
  container_name: prometheus
  restart: always
  ports:
    - "9090:9090"
  volumes:
    - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
  depends_on:
    - admin-panel
    - login-service
    - register-service
    - schedule-service
```

Рис. 4.7

```yaml
grafana:
  image: grafana/grafana-oss:8.5.2
  container_name: grafana
  restart: always
  ports:
    - "3000:3000"
  links:
    - prometheus:prometheus
  volumes:
    - ./grafana:/var/lib/grafana
  environment:
    - GF_SECURITY_ADMIN_USER=admin
    - GF_SECURITY_ADMIN_PASSWORD=password
```

Рис. 4.8

Крок 4. Додавання конфігурації Prometheus.

Створюється файл “prometheus.yml”. У цей файл додаються всі необхідні конфігурації мікросервісів.

Зміст файлу “prometheus.yml” показано на рис. 4.9.

```yaml
scrape_configs:
  - job_name: 'schedule-service'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['schedule-service:8080']
        labels:
          application: 'Schedule Service Application'
  - job_name: 'login-service'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['login-service:8855']
        labels:
          application: 'Login Service Application'
  - job_name: 'register-service'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['register-service:8888']
        labels:
          application: 'Register Service Application'
  - job_name: 'admin-panel'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['admin-panel:9900']
        labels:
          application: 'Admin Panel Application'
```

Рис. 4.9

У цьому прикладі показано налаштування Prometheus для збору метрик з кількох мікросервісів: Schedule Service, Login Service, Register Service та Admin Panel. Конфігурація включає визначення job'ів, які описують, звідки збирати метрики, яким шляхом, та які мітки додавати до метрик.

Файл конфігурації prometheus.yml складається з кількох job'ів, кожен з яких відповідає за збір метрик з окремого мікросервісу. Кожен job визначає наступні параметри:

- job_name: Унікальне ім'я job'у для ідентифікації джерела метрик.
- metrics_path: Шлях, за яким Prometheus збиратиме метрики.
- static_configs: Статичні конфігурації, що включають цільові адреси (targets) та мітки (labels).

Крок 5. Запускаємо всі контейнери.

Для початку оновлюємо всі контейнери в сховищі “docker hub”. Для цього активуємо побудову образів та відсилання їх до репозиторію за допомогою плагіну Jib. Для цього виконуємо команду: ***mvn compile jib:build.***

Після цього оновлюємо локальні образи з репозиторію. Це можна зробити через термінал або в програмі Docker Desktop.

За допомогою команди “***docker compose up***” запускаємо всі контейнери.

Всі запущені контейнери застосунку Scheduler API можна бачити на рис. 4.10.

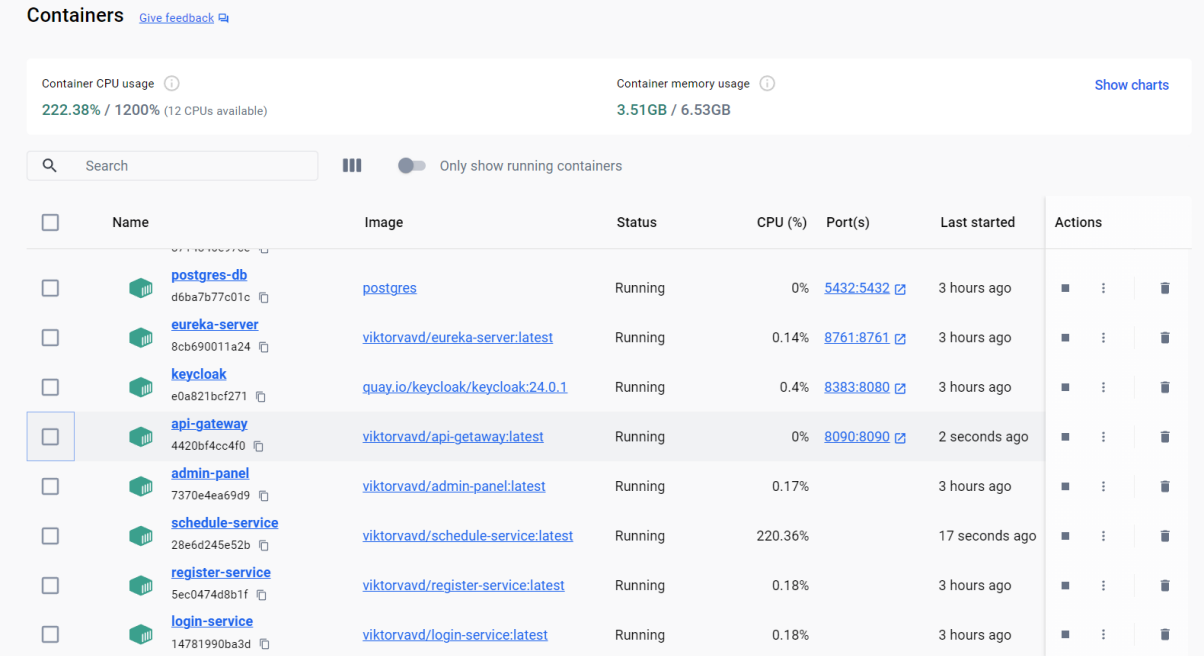

Рис. 4.10

Після запуску всіх необхідних для роботи програми контейнерів, можна переходити до моніторингу застосунку.

Спочатку варто переконатися, що всі мікросервіси підключені до Prometheus коректно. Для цього потрібно перейти за адресою http://localhost:9090 та зайти у вкладку Status --> Service Discovery. Після цього пересвідчитися, що всі сервіси мають статус “UP”.

На рис. 4.11 показано як виглядає вкладка Service Discovery в Prometheus для застосунку Scheduler API. Можна бачити, що всі мікросервіси працюють коректно та жодних помилок не було знайдено. Це означає, що була вказана правильна конфігурація у файлі “prometheus.yml”.

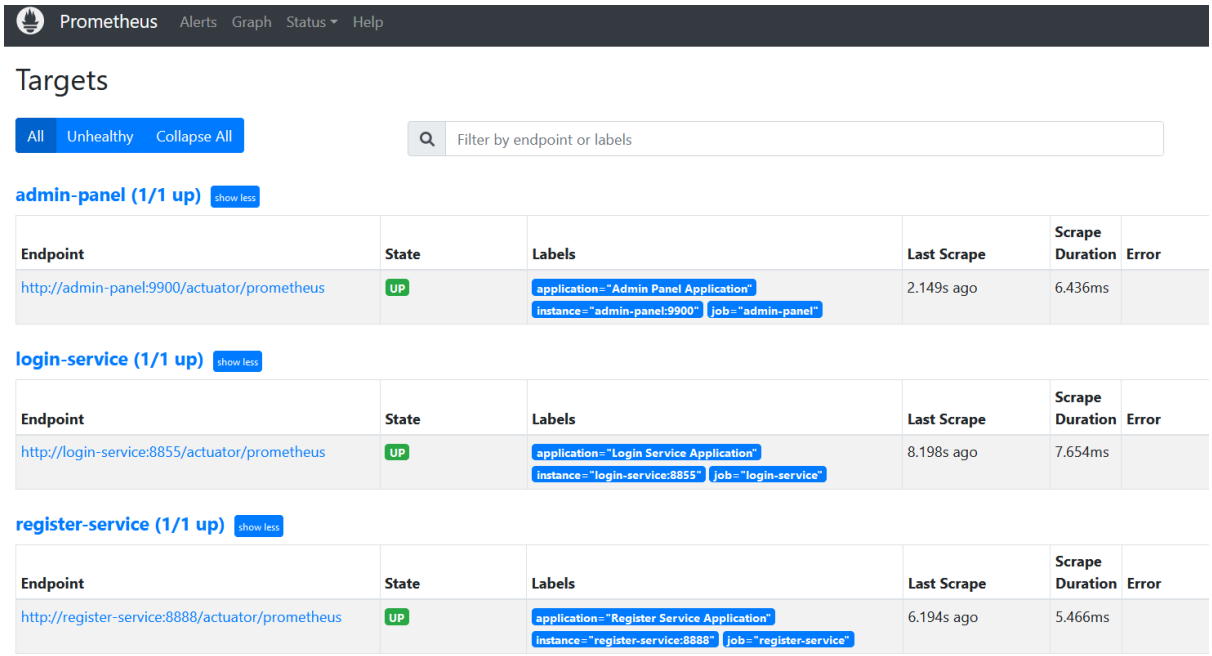

Рис. 4.11

Після перевірки коректності конфігурації сервісів в Prometheus варто перейти до налаштувань Grafana.

Джерело даних для відображення в Grafana може бути взяте з різних ресурсів, таких як: Graphite, OpenTSDB, InfluxDB, Google Cloud Monitoring, Postgres, MySQL та багато інших, в тому числі і Prometheus.

Імпортувавши дані з Prometheus, для цього необхідно, всього лиш, вказати посилання на нього (в даному випадку: http://localhost:9090), можна приступати до створення нової “dashboard”. Для цього потрібно перейти на відповідну вкладку та вказати потрібні джерела даних.

У випадку з Scheduler API маємо, у якості джерела даних, інструмент для моніторингу роботи міросервісів Prometheus.

На рис. 4.12 показано, як виглядає dashboard для запиту “logback_events_total”, який показує логування в мікросервісах.

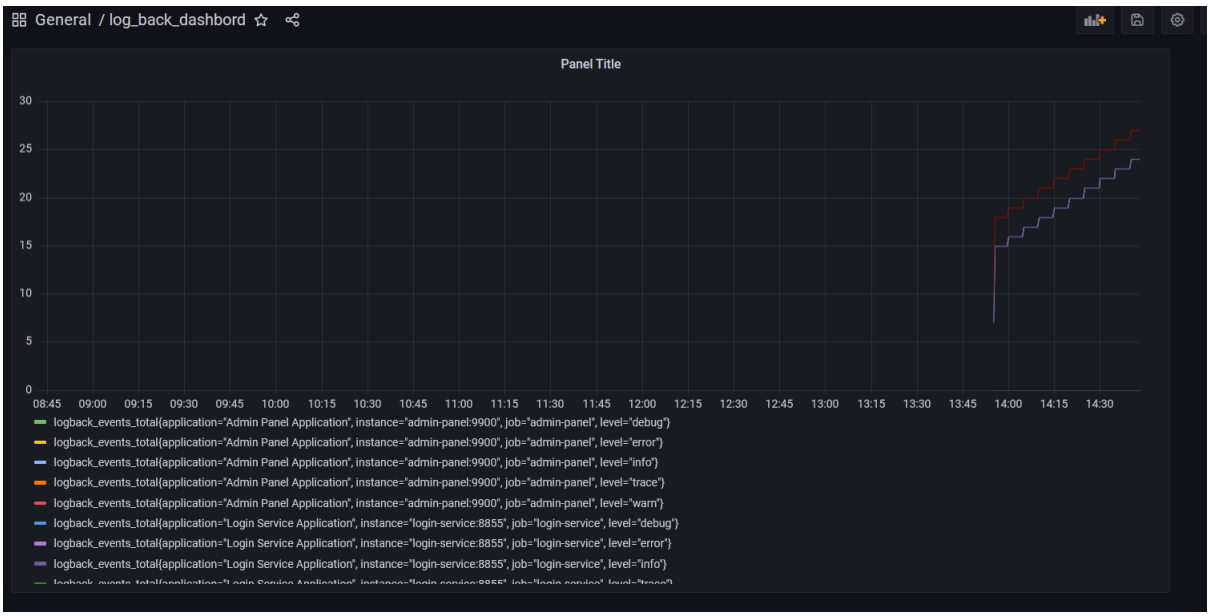

Рис. 4.12

Висновки до глави 4

У цьому розділі було проаналізовано та реалізовано ключові аспекти розгортання та моніторингу мікросервісного застосунку Scheduler API.

Як вже зазначалося, контейнеризація є важливим кроком у розгортанні мікросервісного застосунку, оскільки вона дозволяє ізолювати кожен мікросервіс у власному контейнері, забезпечуючи відокремлене середовище виконання. Використання Jib для контейнеризації забезпечило декілька переваг в порівнянні зі створенням Dockerfile для кожного окремого сервісу, основна з яких це автоматизоване створення образів та завантаження їх на Docker Hub.

За допомогою Docker Compose, у свою чергу, було створено зручний спосіб керування багатоконтейнерним застосунком Scheduler API. Використовуючи один конфігураційний файл, було описано всі необхідні сервіси, які необхідні для роботи додатку, а також легко їх запускати, зупиняти та масштабувати.

Було використано такі інструменти, як Prometheus та Grafana, для моніторингу мікросервісів Scheduler API.

Використання Prometheus надало можливість ефективно збирати, зберігати та обробляти метрики з різних мікросервісів. В свою чергу, використання Grafana надало інструменти для візуалізації даних, що збираються Prometheus. Це дозволило створювати інформативні дашборди, які наочно відображають стан і продуктивність мікросервісів.

Висновки по роботі

У кваліфікаційній роботі "Технології реалізації застосунку з використанням стеку фреймворків Spring на базі мікросервісної архітектури" розглянуто різноманітні аспекти розробки, контейнеризації та моніторингу мікросервісного додатку з використанням сучасних технологій та інструментів.

У першому розділі роботи було розглянуто основні модулі Spring, що включають Spring Core, Spring MVC, Spring Data та Spring Boot. Детально розібрано принцип роботи Dependency Injection (DI) у Spring. Крім того, досліджено можливості Spring Cloud для побудови розподілених систем, включаючи балансування навантаження та виявлення сервісів.

У другому розділі було розглянуто опис задач та принципів мікросервісної архітектури, зокрема модульність, незалежність розгортання та масштабування. Проаналізовано недоліки мікросервісного підходу, такі як складність управління та забезпечення безпеки. Детально розглянуто інструмент контейнеризації застосунків Docker.

Практична частина роботи демонструє створення мікросервісного додатку Scheduler API. Проведено загальний огляд застосунку та описано структуру бази даних проекту. Виконано порівняльну характеристику інструментів взаємодії мікросервісів, таких як RestTemplate та WebClient, та обрано найбільш підходящий інструмент для цього застосунку. Налаштовано виявлення служб з використанням Netflix Eureka, що забезпечує динамічне виявлення та реєстрацію мікросервісів. Впроваджено Spring Cloud Gateway для управління маршрутизацією запитів. Також було впроваджено забезпечення безпеки мікросервісів за допомогою Spring Security та Keycloak.

В останньому розділі роботи описано процес контейнеризації застосунку за допомогою Jib та Docker Compose, що дозволяє автоматизувати створення та розгортання контейнерів. Для забезпечення моніторингу роботи контейнерів використано Prometheus та Grafana. Prometheus забезпечує збір та зберігання метрик, а Grafana - їх візуалізацію, що дозволяє оперативно реагувати на можливі проблеми та аномалії в роботі мікросервісів.

В цілому, виконана робота підтверджує, що використання стеку фреймворків Spring у поєднанні з мікросервісною архітектурою є ефективним підходом для створення масштабованих, модульних та надійних додатків.

Література та список джерел:

- Spring in Action, 6th edition. Craig Walls
- Spring Microservices in Action. Second Edition . John Carnell
- Using Docker: Developing and Deploying Software with Containers 1st Edition. Adrian Mouat
- Docker in Action 2nd ED. Jeff Nickoloff

- https://docs.spring.io/spring-boot/docs/current/reference/html/
- https://www.baeldung.com/jib-dockerizing
- https://docs.spring.io/spring-cloud-gateway/reference/index.html
- https://docs.spring.io/spring-cloud-config/reference/
- https://docs.spring.io/spring-cloud-netflix/docs/current/reference/html/#netflix-eureka-client-starter
- https://docs.spring.io/spring-cloud-security/docs/current/reference/html/#_oauth2_protected_resource
- https://cloud.google.com/java/getting-started/jib
- https://www.keycloak.org/getting-started/getting-started-docker
- https://docs.docker.com/compose/gettingstarted/

Додаток А. Зміст файлу «docker-compose.yml» .

```yaml
version: '3.8'
services:
  postgres-db:
    container_name: postgres-db
    image: postgres
    environment:
      POSTGRES_DB: schedulerDB
      POSTGRES_USER: root
      POSTGRES_PASSWORD: 1111
      PGDATA: /data/postgres
    volumes:
      - ./postgres-order:/data/postgres
    ports:
      - "5432:5432"
    restart: always

  ## Keycloak Config with Mysql database
  keycloak-mysql:
    container_name: keycloak-mysql
    image: mysql:8
    volumes:
      - ./volume-data-keycloak/mysql_keycloak_data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: keycloak
      MYSQL_USER: keycloak
      MYSQL_PASSWORD: password
  keycloak:
    container_name: keycloak
    image: quay.io/keycloak/keycloak:24.0.1
    command: [ "start-dev", "--import-realm" ]
    environment:
      DB_VENDOR: MYSQL
      DB_ADDR: mysql
      DB_DATABASE: keycloak
      DB_USER: keycloak
      DB_PASSWORD: password
      KEYCLOAK_ADMIN: admin
      KEYCLOAK_ADMIN_PASSWORD: admin
    ports:
      - "8383:8080"
    volumes:
      - ./docker/keycloak/realms/:/opt/keycloak/data/import/
    depends_on:
      - keycloak-mysql


  ## Eureka Server
  discovery-server:
    image: viktorvavd/eureka-server:latest
    container_name: eureka-server
    ports:
      - "8761:8761"
    environment:
      - SPRING_PROFILES_ACTIVE=docker
```

```
  api-gateway:
    image: viktorvavd/api-getaway:latest
    container_name: api-gateway
    ports:
      - "8090:8090"
    expose:
      - "8090"
    environment:
      - SPRING_PROFILES_ACTIVE=docker
      - LOGGING_LEVEL_ORG_SPRINGFRAMEWORK_SECURITY= TRACE
    depends_on:
      - discovery-server
      - keycloak

  ## Login-Service Docker Compose Config
  login-service:
    container_name: login-service
    image: viktorvavd/login-service:latest
    environment:
      - SPRING_PROFILES_ACTIVE=docker
      - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres-
db:5432/schedulerDB
    depends_on:
      - discovery-server
      - api-gateway
      - postgres-db


  register-service:
    container_name: register-service
    image: viktorvavd/register-service:latest
    environment:
      - SPRING_PROFILES_ACTIVE=docker
      - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres-
db:5432/schedulerDB
    depends_on:
      - postgres-db
      - discovery-server
      - api-gateway


  schedule-service:
    container_name: schedule-service
    image: viktorvavd/schedule-service:latest
    environment:
      - SPRING_PROFILES_ACTIVE=docker
      - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres-
db:5432/schedulerDB
    depends_on:
      - postgres-db
      - discovery-server
      - api-gateway


  admin-panel:
    container_name: admin-panel
    image: viktorvavd/admin-panel:latest
    environment:
      - SPRING_PROFILES_ACTIVE=docker
      - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres-
db:5432/schedulerDB
```

```
    depends_on:
      - discovery-server
      - api-gateway
      - postgres-db


  prometheus:
    image: prom/prometheus:v2.37.1
    container_name: prometheus
    restart: always
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
    depends_on:
      - admin-panel
      - login-service
      - register-service
      - schedule-service


  grafana:
    image: grafana/grafana-oss:8.5.2
    container_name: grafana
    restart: always
    ports:
      - "3000:3000"
    links:
      - prometheus:prometheus
    volumes:
      - ./grafana:/var/lib/grafana
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=password
```