КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Кваліфікаційна робота

на здобуття освітнього рівня магістра
за спеціальністю 121 Інженерія програмного забезпечення
на тему:

РОЗРОБКА РЕАКТИВНИХ СЕРВЕРНИХ ЗАСТОСУНКІВ З ВИКОРИСТАННЯМ ФРЕЙМВОРКУ VERT.X ТА МОВИ ПРОГРАМУВАННЯ KOTLIN

Виконав студент 2-го курсу магістратури
Андрій БЛАГИЙ

(підпис)

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Оксана ШКІЛЬНЯК

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем
«10» травня 2023 р.,
протокол № 9
Завідувач кафедри
Олександр ПРОВОТАР _____
(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи 66 сторінок, 22 ілюстрації, 15 джерел посилань.

АСИНХРОННІСТЬ, БАЗА ДАНИХ, БЕКЕНД, ЗАСТОСУНОК, KOTLIN, НАСТІЛЬНА ГРА, РЕАКТИВНІСТЬ, СЕРВЕР, JVM, VERT.X.

Об’єктом роботи є дослідження та розробка реактивного та асинхронного програмного забезпечення для серверів. Предметом роботи є розробка реактивного серверного застосунку-соцмережі для гравців в настільні ігри.

Метою роботи є дослідження основних аспектів розробки асинхронного та реактивного програмного забезпечення для серверів та створення реактивного серверного застосунку.

Методи та інструменти роботи: мова програмування Kotlin, фреймворки Vert.x та Vert.x-web, співпрограми (coroutine), система автоматичної збірки Gradle, комерційне інтегроване середовище розробки Intellij IDEA Ultimate Edition, база даних PostgreSQL.

Результати роботи: проаналізовано доцільність розробки асинхронних реактивних застосунків, створено інфраструктурний код для зручності розробки з використанням фреймворку Vert.x, розроблено асинхронний реактивний серверний застосунок-соціальну мережу для гравців у настільні ігри.

Розроблений інфраструктурний код для створення серверного застосунку може використовуватись як додаткова бібліотека для комерційних серверних застосунків, а також як основа для виконання лабораторних робіт на курсах «Об'єктноорієнтоване програмування» та «Інформаційні системи». Сама розроблена система може використовуватись в реальному житті для гравців в настільні ігри.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

API (Application Programming Interface) – прикладний програмний інтерфейс;

JVM (Java Virtual Machine) – віртуальна машина Java;

I/O (Input/Output) – ввід/вивід інформації;

HTTP (Hypertext Transfer Protocol) – протокол передачі гіпертекстових документів;

SMTP (Simple Mail Transfer Protocol) – комунікаційний протокол для пересилання електронної пошти;

JSON (JavaScript Object Notation) – текстовий формат обміну даними між комп'ютерами;

XML (Extensible Markup Language) – розширювана мова розмітки;

JDBC (Java Data Base Connection) – з'єднання з базами даних на Java;

JWT (JSON Web Token) – стандарт токена доступу на основі JSON;

URI (Uniform Resource Identifier) – уніфікований ідентифікатор ресурсів;

DI (Dependency Injection) – механізм впровадження залежностей.

ВСТУП

Оцінка сучасного стану об’єкта розробки. Розробка серверних застосунків стала надзвичайно актуальною в останні роки внаслідок швидкого розвитку хмарних технологій, та масштабування застосунків в реальному часі. Проте все частіше класичні серверні застосунки витісняються асинхронними та реактивними. У зв’язку з цим, фреймворк Vert.x та мова програмування Kotlin знайшли широке застосування в розробці високопродуктивних та масштабованих серверних застосунків на платформі JVM.

Актуальність роботи та підстави для її виконання. В наш час, коли відбувається стрімкий розвиток хмарних технологій та IoT, розробка високопродуктивних та масштабованих застосунків є надзвичайно важливою.

Однак, розробка серверних застосунків зустрічає деякі труднощі. Серверні застосунки часто потребують масштабування та підтримки багатьох одночасних підключень, що може призводити до затримок та втрати продуктивності. Також, традиційні серверні архітектури не завжди можуть бути ефективними при розробці застосунків з високими вимогами до продуктивності та масштабованості.

У зв’язку з цим, фреймворк Vert.x та мова програмування Kotlin стали досить популярними в розробці серверних застосунків. Vert.x - це фреймворк, який дозволяє розробляти високопродуктивні та масштабовані серверні застосунки з використанням асинхронного та неблокуючого програмування. Kotlin - мова програмування, що має простий та зрозумілий синтаксис, що дозволяє писати чистіший та структурованіший код. Разом, Vert.x та Kotlin дозволяють ефективно розробляти реактивні

серверні застосунки з високими вимогами до продуктивності та масштабованості.

Мета й завдання роботи. Метою кваліфікаційної роботи є дослідження основних аспектів розробки асинхронного та реактивного програмного забезпечення для серверів та створення реактивного серверного застосунку з використанням фреймворку Vert.x та мови програмування Kotlin. Основним завданням є створення ефективного та масштабованого рішення, що забезпечує високу продуктивність та мінімальний час відгуку.

Для досягнення мети було поставлено такі наступні завдання:

- дослідити стандартні технології для реалізації серверних застосунків;
- порівняти класичний та реактивний підходи для розробки серверних застосунків;
- дослідити технічні особливості фреймворків Vert.x, Vert.x-web та мови програмування Kotlin;
- розробити структуру застосунку та вибрати оптимальні компоненти для його реалізації;
- імплементувати інфраструктурний код для полегшення впровадження фреймворку vert.x
- реалізувати механізми, що забезпечують взаємодію між компонентами застосунку та їх синхронізацію;
- імплементувати реактивний серверний застосунок;

В результаті виконання цих завдань очікується отримати готовий застосунок, який буде містити ефективну та оптимізовану реалізацію різноманітних функцій та можливостей для подальшої розробки його функціоналу.

Об'єкт, методи й засоби розроблення. Об’єктом роботи є дослідження та розробка реактивного та асинхронного програмного забезпечення для серверів.

Розробці даного серверного застосунку передував аналіз різних підходів до розробки серверних застосунків та порівняння цих підходів. Після цього був здійснений аналіз можливостей вибраного фреймворку, та супутніх бібліотек. Також, було впроваджено додатковий інфраструктурний код, який полегшував розробку даного серверного застосунку.

Інструментом розробки було обрано мову програмування Kotlin, фреймворки Vert.x та Vert.x-web, співпрограми (coroutine), систему автоматичної збірки Gradle, комерційне інтегроване середовище розробки Intellij IDEA Ultimate Edition (безкоштовно поширюване для студентів), базу даних PostgreSQL.

Можливі сфери застосування. Розроблений інфраструктурний код для створення серверного застосунку може використовуватись як основа для комерційних серверних застосунків, а також як основа для виконання лабораторних робіт на курсах «Об'єктноорієнтоване програмування» та «Інформаційні системи». Сама розроблена система може використовуватись в реальному житті для гравців в настільні ігри.

Взаємозв’язок з іншими роботами. За методами розробки та інструментальними засобами робота є продовженням тематики яка розглядалась в навчальних дисциплінах «Об'єктноорієнтоване програмування» та «Інформаційні системи».

РОЗДІЛ 1. ОГЛЯД СУЧАСНОГО СТАНУ В РОЗРОБЦІ СЕРВЕРНИХ ЗАСТОСУНКІВ

1.1 Серверні застосунки та особливості їх розробки

Серверні (Back-End) застосунки – це програмні компоненти, які надають можливість взаємодії з іншими програмними компонентами за допомогою мережі Інтернет. Вони дозволяють виконувати різноманітні дії, такі як відправлення запитів, отримання даних та інші. Вони можуть бути використані для створення різноманітних систем, таких як системи електронної комерції, банківські системи, системи управління контентом та інші [1].

Серверні застосунки мають декілька ключових принципів, які необхідно враховувати під час їх розробки. Першим принципом є безпека. Вебсервіси повинні бути захищені від несанкціонованого доступу та зламів. Другим принципом є інтероперабельність. Вебсервіси повинні бути здатні працювати з різними системами та послугами, які використовують різні технології. Третім принципом є масштабованість. Вебсервіси повинні бути здатні працювати з великою кількістю користувачів та обробляти великий потік даних. Четвертим принципом є надійність. Вебсервіси повинні бути стійкими до збоїв та забезпечувати неперервну роботу.

Для розробки вебзастосунків використовують різні відкриті стандарти та протоколи для інтеграції з різними програмами [2]:

SOAP (Simple Object Access Protocol) – це протокол обміну даними на основі XML, який використовується для обміну повідомленнями між різними системами. SOAP дозволяє використовувати різні мережеві

протоколи, такі як HTTP, SMTP та інші. SOAP має вбудовані механізми для обробки помилок та підтвердження доставляння повідомлень.

REST (Representational State Transfer) – це архітектурний стиль для створення вебсервісів, який використовується для передачі даних між різними системами. REST використовує HTTP протокол для передачі даних та має стандартні методи HTTP, такі як GET, POST, PUT та DELETE. REST повністю використовує можливості мережі Інтернет та дозволяє використовувати різні формати даних, такі як JSON та XML.

GraphQL – це мова запитів та середовище виконання запитів, яке дозволяє взаємодіяти зі схемами даних. GraphQL дозволяє клієнтам здійснювати точні запити до сервера та отримувати тільки необхідні дані. GraphQL дозволяє зменшити кількість запитів, які необхідно здійснювати, та зменшити розмір переданих даних.

Один з найважливіших аспектів розробки вебсервісів – це безпека. Найпоширеніші загрози для вебсервісів - це атаки на кінцеві точки API та атаки на доступ до бази даних. Для захисту вебсервісів використовуються різні методи та технології, такі як автентифікація та авторизація, шифрування даних та використання HTTPS (HTTP Secure).

Автентифікація – це процес перевірки, що користувач є тим, за кого себе видає. Авторизація – це процес визначення, чи має користувач право доступу до певного ресурсу. Шифрування даних - це процес перетворення даних у такий формат, який не може бути зрозумілий без ключа для розшифрування. HTTPS – це захищений протокол передачі даних, який шифрує всі дані, що передаються між клієнтом та сервером.

1.2 Стандартні технології для розробки серверних застосунків

Згідно з результатами опитування з сайту dou.ua (рисунок 1) найпопулярнішими мовами програмування для розробки серверних застосунків є Java, C#, PHP, Python, Typescript. [3].

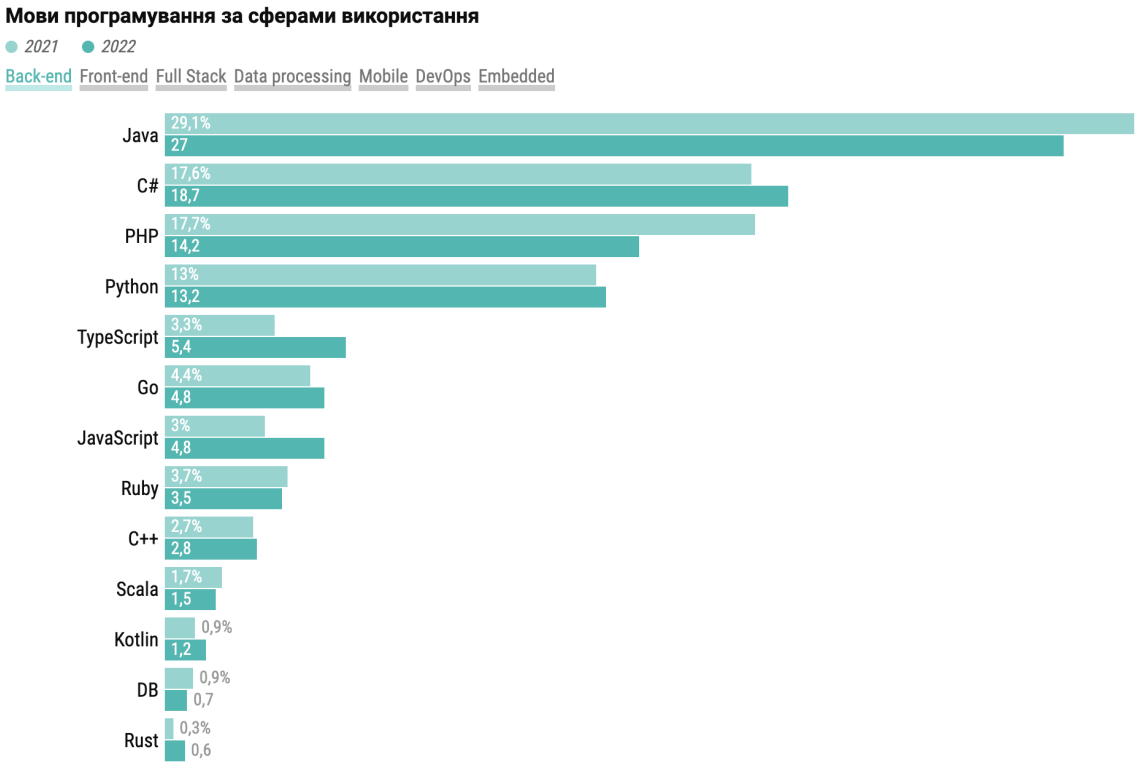

Рисунок 1 – Розподіл мов програмування за сферами використання

Розглянемо їх трішки детальніше [4]:

– Java. Одна з найпопулярніших мов програмування для розробки серверних застосунків. Вона має багато фреймворків, таких як Spring та JSF, які допомагають швидко розробляти та випускати вебзастосунки;
– C#. Ця мова програмування використовується в Microsoft .NET Framework для розробки серверних застосунків. Засоби, такі як

ASP.NET та ASP.NET Core, дозволяють розробляти потужні вебсервіси на платформі .NET;

- PHP. Мова програмування, яка спеціалізується на веброзробці, зокрема розробці вебсервісів. PHP має багато фреймворків, таких як Laravel та Symfony;
- Python. Ця мова програмування дуже популярна для роботи з даними та розробки нейронних мереж, штучного інтелекту, чат-ботів, а також вебзастосунків. Серед найпопулярніших фреймворків для вебсервісів є Django та Flask;
- TypeScript. Мова програмування, яка є розширенням JavaScript, з сильнішою типізацією. Вона зарекомендувала себе в розробці серверних застосунків, особливо з використанням фреймворків, таких як Node.js.

Оскільки повноцінна розробка без використання фреймворків майже неможлива, розглянемо їх також детальніше:

- Spring – це один з найпопулярніших фреймворків для розробки серверних застосунків на мові Java. Він має широкі можливості для розробки, такі як вбудована підтримка Object Relationship Mapping технологій, безпеки, тестування та інших. Spring також пропонує багато різних модулів, таких як Spring MVC (Model View Controller), Spring Security, Spring Boot, Spring Cloud і т.д. Вони роблять розробку застосунків простішою та швидшою;
- ASP.NET – це фреймворк для розробки серверних застосунків на мові C#. Він розроблений компанією Microsoft та є частиною .NET Framework. ASP.NET має великий набір функцій, що дозволяє розробляти різноманітні застосунки – від вебсайтів до вебслужб та вебзастосунків. Фреймворк має

вбудовану підтримку безпеки, обробки помилок та інші корисні можливості;

- Laravel – це фреймворк для розробки серверних застосунків на мові PHP. Він використовує шаблон проєктування MVC (Model-View-Controller) та має вбудовану підтримку маршрутизації, шаблонів та інші корисні функції. Laravel є одним з найпопулярніших фреймворків для розробки вебзастосунків на мові PHP;
- Django – це фреймворк для розробки вебзастосунків на мові Python. Він використовує шаблон проєктування MVC та має вбудовану адміністративну панель, яка дозволяє легко управляти базою даних і моделями застосунку. Django також забезпечує підтримку автоматичної генерації адміністративного інтерфейсу, що дозволяє розробникам швидко створювати застосунки з використанням готових компонентів;
- Node.js – це фреймворк для розробки серверних застосунків на мові JavaScript, який підтримує реактивний підхід. Node.js має неймовірну швидкість та масштабованість, що робить його ідеальним для розробки високонавантажених застосунків в реальному часі. Node.js також має велику кількість модулів та пакетів, доступних через менеджер пакетів npm, що дозволяє розробникам ефективно використовувати готові рішення для своїх проєктів.

Як можна помітити, із вибраних популярних фреймворків лише Node.js підтримує реактивний підхід. Розглянемо, що таке реактивний підхід, і навіщо він потрібен.

1.3 Порівняння класичного та реактивного підходу до розробки серверних застосунків

Розробка серверних застосунків – це процес створення програмного забезпечення, яке виконується на сервері та забезпечує обмін даними між клієнтами. Сьогодні на ринку програмного забезпечення є два основних підходи до розробки серверних застосунків: класичний та реактивний [5].

Класичний підхід, якого дотримуються більшість фреймворків, до розробки серверних застосунків передбачає використання синхронної моделі програмування з використанням багатопоточності (multithreading). Модель наявності одного потоку на з’єднання є зручною, оскільки розробники можуть покластися на код традиційного імперативного стилю програмування. Внаслідок цього можна уникнути більшості стандартних помилок пов'язаних з багатопотоковим доступом до пам'яті.

Проте проблема виникає, коли зростає навантаження на сервіс. Для цього явища є навіть окрема назва – C10k. Це умовна назва проблеми, коли підключено одночасно десять тисяч користувачів. Деякі з цих потоків буде заблоковано, оскільки вони очікують завершення операцій вводу-виводу, деякі будуть готові до обробки результатів введення-виведення, а деякі виконуватимуть інтенсивні задачі центрального процесора.

Сучасні процесори мають дуже хороші планувальники, але не можна очікувати, що вони справлятимуться з п'ятдесятьма тисячами потоків так само легко, як з п'ятьма тисячами. Крім того, потоки недешеві: створення потоку займає декілька мілісекунд, а новий потік споживає близько 1 МБ пам'яті.

Реактивний підхід до розробки серверних застосунків передбачає використання асинхронної моделі програмування. У цьому підході запити обробляються незалежно один від одного, що дозволяє серверу

обслуговувати багато клієнтів одночасно та зменшує час очікування клієнтів на відповідь. Для реалізації реактивного підходу використовуються спеціальні фреймворки та бібліотеки, які дозволяють програмістам розробляти асинхронні застосунки.

Наведемо основні принципи реактивної системи [6]:

- відгукливість (responsive): реактивна система повинна забезпечувати швидкий і постійний час реакції, а отже, постійну якість обслуговування;
- відмовостійкість (resilience): реактивна система повинна залишатися доступною та функціональною у випадку випадкових збоїв через реплікацію та ізоляцію;
- еластичність (elastic): така система повинна бути здатною автоматично збільшувати потужність за непередбачуваних робочих навантажень завдяки економічно ефективній масштабованості;
- керованість повідомленнями (message-driven): система повинна спиратися на асинхронну передачу повідомлень між компонентами системи, що дозволяє забезпечувати гнучкість та розділення обов'язків.

Однією з ключових переваг реактивного підходу є можливість обробки великої кількості запитів одночасно, що робить його більш масштабованим та ефективним в порівнянні з класичним підходом. Крім того, реактивний підхід дозволяє програмістам легко обробляти асинхронні події та реалізовувати взаємодію з іншими мікросервісами.

Однак, розробка реактивних застосунків також має свої недоліки. Один з найбільш важливих недоліків – це складність розробки та налагодження реактивних систем. Реактивні системи потребують від

програмістів великого рівня володіння асинхронним програмуванням та знання реактивних патернів.

1.4 Мова програмування Kotlin

Kotlin – це мова програмування з відкритим вихідним кодом, яка була створена компанією Jetbrains у 2011 році із метою розширення можливостей Java та забезпечення ефективнішої розробки на платформі Java [7]. Kotlin підтримує функціональне програмування, безпечну типізацію, інтероперабельність з Java та багато інших функцій, які допомагають зробити код ефективнішим та зручнішим для розробників.

Значною перевагою Kotlin є співпрограми (coroutine, корутини). Співпрограма – це механізм, що дозволяє виконувати частини коду в нескінченному циклі, перериваючи їх відповідно до певних умов та зберігаючи їх стан між викликами. Використовуючи співпрограми можна покращити ефективність та простоту програми.

Ще однією з основних функцій Kotlin є підтримка функціонального програмування. Функціональне програмування – це підхід до програмування, в якому програми складаються з функцій, які можуть бути передані як аргументи та повернені як результати інших функцій.

У Kotlin функції є об'єктами першого класу, що дозволяє передавати їх як параметри, повертати їх як результат та зберігати в змінних. Kotlin також підтримує лямбда-вирази, які дозволяють створювати функції без визначення окремих методів.

Наприклад, розглянемо функцію, яка виводить список чисел, які більші за певне значення (рисунок 2). У цьому прикладі filter – це функція вищого порядку, яка приймає лямбда-вираз { it > 5 } як параметр. Цей

лямбда-вираз визначає, що елементи списку, які більші за 5, повинні бути включені до відфільтрованого списку.

```
fun main() {
    val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    val filteredNumbers = numbers.filter { it > 5 }
    println(filteredNumbers)
}
```

Рисунок 2 – Приклад використання лямбда функцій

Іншою важливою перевагою Kotlin є безпечна типізація. Це означає, що Kotlin намагається забезпечити безпеку виконання програми на етапі компіляції, запобігаючи помилкам виконання на етапі виконання.

У Java, наприклад, є можливість виникнення помилок класу NullPointerException через те, що змінна, на яку вказує посилання, не містить жодного об'єкта. Kotlin спрощує цю проблему, використовуючи оператор ?, що дозволяє забезпечити безпеку виконання навіть якщо значення змінної може бути null (рисунок 3).

```
fun main() {
    val name: String? = null
    println(name?.length)
}
```

Рисунок 3 – Приклад використання безпечних викликів

У цьому прикладі, якщо змінна name містить значення null, виклик методу length буде проігнорований, а програма продовжить свою роботу

без помилки. Якщо ж name містить значення не-null, то буде повернуто значення довжини рядка.

Kotlin підтримує інтероперабельність з Java, що дозволяє використовувати код, написаний на Java, в Kotlin-проєктах та навпаки. Це означає, що Kotlin може бути використаний як альтернатива Java для розробки на платформі JVM.

Наприклад, використовуючи Kotlin, можна легко інтегрувати Java-бібліотеки у свій проєкт (рисунок 4). У цьому прикладі ArrayList – це клас з бібліотеки Java, який легко інтегрується у Kotlin-проєкт.

```
import java.util.*

fun main() {
    val list = ArrayList<String>()
    list.add("Hello")
    list.add("Kotlin")
    println(list)
}
```

Рисунок 4 – Приклад інтероперабельності з Java

Kotlin має вбудовану систему null safety, що дозволяє зменшити кількість помилок, пов'язаних з використанням значень null. Null safety дозволяє встановлювати, чи можуть значення змінних бути null, та при компіляції перевіряє, чи вони коректно використовуються в програмі. Якщо виконання програми може призвести до помилки, Kotlin повідомить про це ще на етапі компіляції, що дозволяє забезпечити безпечніше виконання програми на етапі виконання (рисунок 5).

```
fun main() {
    val name: String? = null
    println(name.length)
}
```

Рисунок 5 – Приклад null safety

У цьому прикладі Kotlin попереджає про помилку на етапі компіляції, оскільки name може мати значення null, але виклик методу length потребує не-null значення. І результатом компіляції цього коду буде наступне повідомлення про помилку: Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?

РОЗДІЛ 2. ОГЛЯД ФРЕЙМВОРКУ VERT.X ТА СУПУТНІХ БІБЛІОТЕК

Vert.x – це фреймворк для розробки асинхронних та реактивних застосунків на Java та інших мовах програмування [8]. Він надає розробникам потужні інструменти для створення ефективних застосунків, що можуть обробляти багато запитів одночасно. Згідно зі статистикою ресурсу TechEmpower, Vert.x посідає провідні місця по швидкодії серед фреймворків написаних мовою програмування Java [9].

Фреймворк забезпечує високу продуктивність, масштабованість та надійність завдяки використанню асинхронного програмування та неблокуючих операцій введення/виведення (I/O). Для забезпечення асинхронної та масштабованої обробки запитів Vert.x використовує модель акторів.

В основі фреймворку Vert.x лежить бібліотека vertx-core. Вона забезпечує такі функції, як:

- написання TCP (Transmission Control Protocol) клієнтів і серверів;
- написання HTTP-клієнтів і серверів, включаючи підтримку WebSockets;
- шина подій (The Event bus);
- спільні дані – локальні карти (local maps) та кластерні розподілені карти (clustered distributed maps);
- періодичні (periodic) та відстрочені (delayed) дії;
- розгортання (deploying) та скасування вершин (Verticles);
- сокети дейтаграм;
- DNS (Domain Name System) клієнт;

- доступ до файлової системи;
- висока доступність;
- кластеризація.

Ядро Vert.x має обмежений функціонал і не містить рішень для доступу до баз даних, авторизації або вебфункціоналу високого рівня. Для цих завдань необхідно використовувати додаткові бібліотеки. Воно невелике та легке. Можна просто використовувати ті частини, які потрібно. Тобто в проєкт будуть вбудовані лише ті речі, які руками додали, на противагу Sping Boot, який із коробки підтягує велику кількість залежностей.

Для використання vertx-core потрібно спершу підключити залежність. У gralde це відбувається додаванням наступного рядка в файл build.gradle:

implementation('io.vertx:vertx-core:4.4.1')

2.1 Створення ядра Vert.x

У екосистемі Vert.x головний об'єктом є Vertx [10]. Це центр керування, в якому можна зробити майже все, наприклад, отримати посилання на шину подій, створення вебклієнтів та вебсерверів, налаштування таймерів тощо. Створюється екземпляр Vertx наступним чином: Vertx.vertx().

Для більшості програм буде достатнього одного екземпляру. Проте інколи бувають випадки, коли потрібна ізоляція між різними групами, і тоді потрібно створювати декілька екземплярів. Проте це радше виняток з правила.

Під час створення екземпляра можна налаштувати велику кількість параметрів, таких як кластеризація, висока доступність, розмір пулу та інші параметри.

Також одним із принципів у Vert.x є використання Fluent API. Це означає, що декілька викликів методів можна об'єднати в один ланцюг. Це загальноприйнятий шаблон у Vert.x.

Vert.x здебільшого керується подіями. Це означає, що коли у Vert.x відбуваються події, які нас цікавлять, Vert.x викликатиме методи, надсилаючи відповідні події. Деякі приклади подій:

- спрацював таймер;
- дані надійшли на сокет;
- дані були зчитані з диска;
- відбувся виняток (exception);
- HTTP сервер отримав запит.

Для обробки різних подій у Vertx потрібно мати відповідні обробники. Наприклад, щоб отримувати подію таймера кожну секунду треба викликати метод setPeriodic з передачею часу очікування та лямбда-виразу обробника. Або для обробки HTTP-запитів необхідно викликати метод requestHandler з передачею лямбда-виразу, який приймає об'єкт типу request. Ці методи потрібно викликати від екземпляра класу Vertx. Коли відбудеться одна з цих подій, Vertx викличе відповідний обробник.

Це призводить до дуже важливої концепції у фреймворку Vert.x – більшість методів не блокують потік (крім деяких запитів до файлової системи). Якщо обробка методу займає деякий час, то метод приймає обробник, який викличеться, коли метод закінчить обробку та буде готовий результат. Оскільки Vert.x не блокує потоки, це дозволяє використовувати

Vert.x для обробки великої кількості паралельних процесів, використовуючи не велику кількість фізичних потоків.

У звичайних фреймворках, таких як Spring блокування потоку може відбуватись у наступних ситуаціях:

- читання даних із сокета;
- запис даних на диск;
- надсилання повідомлення адресату та очікування відповіді;
- виконання запиту до бази даних;
- та багато інших операцій.

У всіх вищезгаданих випадках, коли потік очікує на результат, він не може виконувати жодних інших завдань і стає непродуктивним. Це означає, що для сервісу, який потребує багато паралельних процесів за допомогою блокуючих API, знадобиться велика кількість потоків, щоб програма продовжувала працювати. Крім того, потоки потребують значної кількості пам'яті (наприклад, для зберігання свого стека) і викликають накладні витрати при перемиканні контексту. Для рівнів паралельності, необхідних у багатьох сучасних програмах, підхід блокування просто не масштабується.

Vertx викликає обробники в спеціальному потоці під назвою «цикл подій» (event loop). Оскільки ці обробники не блокуються, то цей потік може обробити значно більше запитів за один проміжок часу, ніж стандартний підхід із виділенням потоку під кожен запит. Це називається шаблоном реактора (reactor pattern). До прикладу, в Node.js він реалізований за замовчуванням.

У стандартній реалізації реактора є єдиний потік циклу подій, який працює в циклі, доставляючи події обробникам, коли ці події надходять. Проблема з одним потоком полягає в тому, що він може працювати лише на одному ядрі в момент часу, тому, якщо потрібно масштабувати

однопотокову програму реактора (наприклад, програму Node.js) на багатоядерному сервері, потрібно запустити та керувати багатьма процесами.

Vert.x працює по-іншому. Замість одного циклу подій кожен екземпляр Vert.x підтримує кілька циклів подій. За замовчуванням він вибирає число на основі кількості доступних ядер на машині, але це можна змінити в налаштуваннях. Це означає, що один процес Vert.x може масштабуватися на сервері, на відміну від Node.js. І цей шаблон називається Multi-Reactor Pattern.

Проте, попри те, що екземпляр Vertx підтримує кілька циклів подій, будь-який окремий обробник ніколи не виконуватиметься одночасно в різних циклах подій. Тож, якщо заблокувати цикл подій в обробнику, то цей цикл подій не зможе робити нічого іншого, поки він заблокований. Якщо заблокувати всі цикли подій, то програма повністю зависне. Серед блокувань потоку, можна виділити наступні приклади:

- Thread.sleep();
- очікування на mutex, або monitor (в контексті Java мови - syncronized секція);
- виконання «тяжких» запитів до бази даних;
- виконання довготривалих операцій.

Значним проміжком часу називається тривалість більша, ніж очікувана тривалість виконання запиту. Тобто, якщо ми хочемо, щоб наш сервер обробляв тисячу запитів за секунду, то значним проміжком часу буде одна мілісекунда. Тобто, якщо тривалість блокування потоку більша ніж одна мілісекунда, то слід використовувати додаткові інструменти, такі як Future results.

Якщо програма не реагує, це може бути ознакою того, що десь блокується цикл подій. Щоб допомогти діагностувати такі проблеми, Vert.x

автоматично реєструватиме попередження, якщо виявить, що цикл подій не завершив виконання протягом деякого часу.

Thread vertx-eventloop-thread-3 has been blocked for 10183 ms

Vert.x також забезпечить трасування стека, щоб точно визначити, де відбувається блокування.

Більшість бібліотек в екосистемі JVM мають синхронні API і багато методів, ймовірно, блокуватимуть потік. Наприклад, JDBC — він за своєю суттю синхронний і Vert.x не може зробити щось, щоб він став асинхронним. Тому потрібно реалізувати спосіб безпечного використання «традиційних» API блокування в програмі Vert.x.

Це робиться шляхом виклику методу executeBlocking, передаючи обробник, який виконує код що блокує, та обробник результатів, який викликається асинхронно після виконання блокувального коду.

2.2 Майбутні результати (Future results)

У попередньому підрозділі було сказано, що для виконання довготривалих операцій слід використовувати futures. Будь-який асинхронний метод повертає екземпляр Future, в незалежності від того, виклик завершиться успіхом чи невдачею.

Прямого доступу до результатів об'єкта futures не існує. Замість цього, слід встановити обробник, який буде викликаний, коли future завершиться і результат стане доступним.

Крім того, необхідно дбати про порядок виконання декількох futures. Хоча методи onComplete, onFailure і onSuccess дозволяють встановити обробник на результат future, вони не гарантують, що виконання відбудеться в тому ж порядку, що й в коді. В таких випадках краще

використовувати метод andThen, який дозволяє виконати наступний future в певному порядку після завершення попереднього.

За допомогою Vert.x можна досягти координації кількох futures. Він підтримує одночасну композицію (паралельне виконання кількох асинхронних операцій) і послідовну композицію (ланцюгові асинхронні операції).

CompositeFuture.all приймає список futures аргументів і повертає Future, який є успішним, коли всі передані futures є успішними, і невдалим, коли принаймні один із futures не виконується. У разі успіху метод гарантує результати в тому самому порядку, що вказаний у виклику CompositeFuture.all.

CompositeFuture.any приймає список futures аргументів і повертає Future, який є успішним, коли хоча б один із futures буде успішним, і невдалим, коли всі futures не є успішними.

CompositeFuture.join чекає поки всі передані futures не будуть завершені з успіхом або невдачею. Метод приймає список futures аргументів і повертає Future, який є успішним, коли всі futures успішні, і невдалим, коли всі futures завершені, і принаймні один із них невдалий. Відмінність від методу all у тому, що у випадку невдачі на одному з futures цей метод чекає на завершення всіх futures, а all впаде відразу.

2.3 Вершини (Verticles)

Vert.x надає просту та масштабовану модель розгортання та паралельної роботи, яка схожа на акторську модель. Проте, ця модель не є обов'язковою для використання в програмах Vert.x, і можна створювати програми, які не використовують цю модель. Щоб використовувати цю модель, потрібно писати код як набір вершин (Verticle).

Verticles – це фрагменти коду, які розгортаються та запускаються в Vert.x. За замовчуванням, екземпляр Vert.x підтримує N потоків циклу подій (де N – кількість ядер пристрою, помножена на два). Verticles можуть бути написані будь-якою мовою програмування, яку підтримує Vert.x. В одній програмі можуть міститися Verticles, написані різними мовами.

Застосунок, як правило, складається з багатьох екземплярів Verticle, що працюють в одному екземплярі Vert.x одночасно. Різні екземпляри Verticle можуть спілкуватись один з одним, надсилаючи повідомлення по шині подій (event bus).

Реалізовані вершини повинні імплементувати Verticle інтерфейс. Вони можуть імплементувати це безпосередньо, але зазвичай простіше розширити абстрактний клас AbstractVerticle, перевизначивши методи start та stop.

Коли Vert.x розгортає вершину, він викликає метод start, і коли метод завершить виконання, вершина вважається запущеною. Також можна замінити метод зупинки. Його буде викликано Vert.x, коли вершина згортається (undeployment), і коли він завершить роботу, вершина вважається зупиненою.

Існує два типи вершин: стандартні (Standard) та робочі (Worker).

Стандартні вершини – це найпоширеніші та найкорисніші вершини, вони завжди виконуються за допомогою потоку циклу подій. Під час створення вершини, їй призначається потік циклу подій, і метод запуску викликається з цим циклом подій. Vert.x гарантує, що при виклику будь-яких методів, що включають обробник основного API з циклу подій, ці обробники будуть виконуватися в тому ж самому циклі подій. Це означає, що весь код в екземплярі verticle завжди буде виконуватись у тому самому циклі подій (за умови, що не були створені власні потоки).

Тож можна писати весь код у своїй програмі як однопотоковий, оскільки Vert.x автоматично турбується про потоки та масштабування. Це означає, що більше не потрібно стежити за синхронізацією та уникати нестабільності, а також можна уникнути стану гонитви (race condition) і тупикових ситуацій (deadlocks), які є поширеними у ручній розробці «традиційних» багатопотокових програм.

Робочі вершини запускаються за допомогою потоку з робочого пулу. Екземпляр вершини ніколи не буде виконуватись одночасно більш ніж одним потоком. Ці вершини призначені для виклику блокуючого коду, оскільки вони блокуватимуть потік з робочого пулу, а не з потоку подій. Також можна використовувати стандартні вершини для запуску блокуючого коду, використовуючи метод executeBlocking.

Для того, щоб використовувати вершину як робочу, необхідно викликати метод setWorker в DeploymentOptions. Як вже було сказано, екземпляр робочої вершини ніколи не виконується Vert.x одночасно кількома потоками, проте він може виконуватися різними потоками в різний час.

Для того, щоб запустити вершину, використовується метод deployVerticle, до якого можна передати або саму вершину, або її назву. Якщо вказати назву вершини, Vert.x автоматично знайде відповідну VerticleFactory для створення екземпляра даної вершини. Цей метод також може використовуватися для запуску вершини, написаної іншою мовою програмування, яку підтримує Vert.x.

При пошуку VerticleFactory використовується назва вершини, яка може містити префікс перед двокрапкою або суфікс (розширення файлу). Якщо префікс присутній, то за ним шукається відповідна фабрика, якщо ж префікс відсутній, то Vert.x шукатиме фабрику за суфіксом. Якщо ж назва вершини не містить ні префікса, ні суфікса, то Vert.x вважатиме її повним

ім'ям класу Java та використовуватиме відповідну фабрику для її створення.

Приклади переданих рядків:

- js:foo.js – за префіксом вибереться JavaScript фабрика;
- groovy:com.SomeGroovyCompiledVerticle – за префіксом вибереться Groovy фабрика;
- foo.rb – за суфіксом вибереться Ruby фабрика;
- com.SomeJavaCompiledVerticle – через відсутність префікса та суфікса вибереться Java фабрика.

Розгортання вершини саме по собі є асинхронним і може завершитися через деякий час після повернення виклику розгортання. Щоб отримати сповіщення про успішне завершення розгортання, можна використати метод onComplete, передавши йому обробник завершення розгортання. Якщо розгортання успішне, результатом буде ідентифікатор розгортання, який можна використовувати пізніше для скасування розгортання.

Розгортання можна скасувати за допомогою методу undeploy, передавши в нього ідентифікатор вершини. Скасування розгортання також є асинхронним, тому, щоб отримати сповіщення про завершення скасування розгортання, потрібно також використовувати onComplete метод.

Під час розгортання вершини за допомогою назви можна вказати кількість екземплярів вершини, які потрібно розгорнути. Це корисно для легкого масштабування на кількох ядрах.

Коли Vert.x надає подію обробнику або викликає методи запуску або зупинки вершини, виконання пов’язується з екземпляром класу Context. Зазвичай контекст пов'язаний з контекстом циклу подій та прив'язаний до конкретного потоку циклу подій. Це означає, що виконання в цьому

контексті завжди відбувається в тому ж самому потоці циклу подій. У випадку робочих вершин та запуску вбудованого блокуючого коду, контекст виконання буде пов'язаний з виконанням, яке використовує потік з пулу робочих потоків. Для отримання контексту необхідно викликати метод getOrCreateContext. Якщо з поточним потоком пов’язаний контекст, він повторно використовує об’єкт контексту. Якщо ні, створюється новий екземпляр контексту.

Також можна запустити код асинхронно для певного контексту. Іншими словами, можна передати завдання, яке пізніше буде виконане в цьому самому контексті. Для цього необхідно використати метод runOnContext().

Якщо декілька обробників працюють в одному контексті, вони можуть мати потребу у спільному доступі до даних. Об’єкт контексту надає методи для зберігання та отримання даних, які можуть бути спільно використані в контексті. Наприклад, це дає можливість передавати дані до певної дії, яка буде виконана за допомогою методу runOnContext.

У Vert.x часто виникає необхідність виконати дію з певною затримкою або з певною періодичністю. У стандартних вершинах не можна просто зупинити потік, щоб створити затримку виконання певної дії, оскільки це заблокує потік циклу подій. Замість цього можна використовувати таймери Vert.x, які можуть бути одноразовими або періодичними.

Одноразовий таймер у Vert.x спрацьовує один раз після встановленої затримки, вираженої у мілісекундах. Щоб налаштувати таймер, необхідно викликати метод setTimer з відповідними параметрами – тривалість затримки та обробник, який буде виконуватися при спрацюванні таймера. Унікальний ідентифікатор таймера, який можна використовувати для скасування таймера, буде повернутий самим методом setTimer. Крім того,

унікальний ідентифікатор таймера передається обробнику, що дозволяє йому ідентифікувати таймер, який спрацював.

У Vert.x можна також налаштувати таймер на періодичне спрацьовування, використовуючи метод setPeriodic. Обробник подій буде викликано через задану періодичність після першого спрацювання таймера. Як і у випадку з одноразовим таймером, метод setPeriodic поверне унікальний ідентифікатор таймера, який можна використовувати для скасування таймера. Проте, якщо тривалість виконання обробника більша за періодичність таймера, обробник буде виконуватись безперервно, що може призвести до накопичення подій. Якщо потрібно, щоб наступне спрацювання таймера відбулося через певний час після завершення виконання обробника, краще використовувати метод setTimer в кінці обробника. Варто зазначити, що таймери автоматично скасовуються, якщо вершина, в якій вони були налаштовані, згортається.

2.4 Шина подій (The Event bus)

Шина подій є нервовою системою у Vert.x, що міститься в кожному екземплярі класу Vertx. Вона забезпечує взаємодію різних частин програми між собою незалежно від мови, на якій вони написані, та того, чи перебувають вони в тому ж екземплярі Vert.x чи в іншому.

API шини подій у Vert.x простий та підтримує публікацію/підписку, обмін повідомленнями типу «точка-точка» та запит-відповідь. Щоб використовувати шину подій, необхідно зареєструвати обробники, скасувати реєстрацію та надсилати чи публікувати повідомлення.

Повідомлення надсилаються по шині подій на вказаний адрес і Vert.x не накладає жодних обмежень на формат адреси, тому це може бути будь-який рядок. Однак рекомендується використовувати певну схему,

наприклад, використовувати крапки для розмежування простору імен, наприклад, user.new, user.changeMail, або post.update.

Обробники отримують повідомлення, що приходять до шини подій, підписуючись на певну адресу. На одну адресу можна зареєструвати декілька різних обробників, і разом з тим один обробник може бути зареєстрований за кількома різними адресами.

Шина подій використовує шаблон публікація/підписка для доставки повідомлень до відповідних адресатів. Якщо певне повідомлення має бути доставлено до певної адреси, шина подій доставляє це повідомлення всім обробникам, які підписалися на цю адресу.

Vert.x використовує шаблон точка/точка для надсилання повідомлень за адресою, який дозволяє скерувати повідомлення лише до одного зареєстрованого обробника за цією адресою. Якщо за адресою зареєстровано декілька обробників, то буде використаний нестрогий алгоритм циклічного планування для вибору обробника.

За допомогою обміну повідомленнями «точка-точка» можна вказати обробник відповіді, який є необов’язковим. Якщо одержувач отримав повідомлення та обробив його, то він може відповісти на нього, і це може почати діалог між двома вершинами. Цей шаблон називається шаблоном запит-відповідь.

Vert.x має можливість надсилати будь-який простий тип, String або buffers, як повідомлення з коробки. Однак у Vert.x рекомендується використовувати JSON для надсилання повідомлень, оскільки він дуже легкий для створення, читання та аналізу в будь-якій мові програмування, яку підтримує Vert.x.

Шина подій у Vert.x є дуже гнучкою і дозволяє надсилати довільні об'єкти через неї. Для цього необхідно визначити кодек (Codec) для цих

об'єктів, які потрібно буде надіслати. Кодек дозволяє кодувати та декодувати об'єкти для передачі через шину подій.

2.5 Створення HTTP серверу

Vert.x дозволяє легко створювати неблокуючі HTTP-клієнти та сервери з використанням протоколів HTTP/1.0, HTTP/1.1 і HTTP/2.

Для створення HTTP серверу потрібно викликати метод createHttpServer. При виклику цього методу можна передати параметри, які будуть використовуватись для налаштування сервера, або залишити їх за замовчуванням.

Після налаштування сервера потрібно викликати метод listen і передати йому параметри, що вказують на хост та порт, який має бути прослухуваний.

Для обробки запитів у Vert.x слід використовувати метод requestHandler та передати обробник, який буде викликаний при надходженні запиту. У цьому обробнику передається екземпляр HttpServerRequest, який представляє запит HTTP на стороні сервера. Обробник запиту буде викликаний після того, як будуть прочитані всі заголовки запиту. Тіло запиту (якщо воно є) може надійти на сервер з певною затримкою після виклику обробника запиту, оскільки воно може бути дуже великим і зберігання його одразу може вичерпати доступну пам'ять сервера.

Об'єкт запиту сервера містить корисну інформацію про запит, таку як метод, URI, path, params та headers. Щоб отримати кожен із цих параметрів, потрібно скористатись відповідним методом.

Для того, щоб отримати всі фрагменти тіла запиту одразу, можна використати метод bodyHandler на об'єкті запиту. Цей метод буде

викликаний один раз, коли тіло запиту повністю отримано. Якщо тіло запиту є формою, то можна використати метод formAttributes, щоб отримати всі атрибути цієї форми, включно з завантаженими файлами.

Для того, щоб надати відповідь на запит, можна скористатись об'єктом response, який знаходиться в об'єкті request. Після завершення виконання всіх обробників, ця відповідь буде надіслана клієнту. Відповідь може містити статус-код, заголовки та тіло. Зауважимо, що тіло відповіді також може бути файлом, який буде передано клієнту асинхронно і може зайняти певний час після повернення виклику.

У випадку, якщо під час обробки запиту виникне виняток (exception), який не буде оброблений, Vert.x автоматично перехопить його. Для обробки таких виключень можна налаштувати exceptionHandler, передавши йому обробник.

2.6 Фреймворк Vert.x-web

Vert.x-Web – це інструментарій для створення вебзастосунків з використанням Vert.x. Оскільки ядро Vert.x надає достатньо мало функцій для обробки HTTP запитів, для багатьох програм цього може бути недостатньо. Vert.x-Web розширює ядро Vert.x, надаючи ширший набір функціональних можливостей, які полегшують створення повноцінних вебзастосунків.

Можна скористатись Vert.x-Web для створення різних видів вебзастосунків на стороні сервера, таких як класичні вебзастосунки, RESTful вебзастосунки та вебзастосунки реального часу (real time), а також для розробки будь-яких інших вебзастосунків. Vert.x-Web є хорошим вибором для написання RESTful HTTP мікросервісів.

Деякі з ключових функцій Vert.x-Web включають:

- маршрутизація (на основі методу та шляху запиту);
- зіставлення шаблону регулярного виразу для шляхів;
- вилучення параметрів зі шляхів;
- узгодження типу тіла запиту;
- обробка тіла запиту;
- обмеження розміру тіла;
- багатокомпонентні форми;
- завантаження багатокомпонентних файлів;
- підтримка сесії – як локальної (для закріплених сеансів), так і кластерна (для не закріплених);
- підтримка CORS (Cross Origin Resource Sharing);
- обробник сторінки про помилку;
- базова/дайджест-автентифікація HTTP;
- аутентифікація на основі перенаправлення;
- обробники авторизації;
- авторизація на основі JWT/OAuth2;
- авторизація користувача/ролі/дозволу;
- обробка піктограм;
- обробка часу відповіді;
- обслуговування статичних файлів, включаючи логіку кешування та список каталогів;
- шина подій (event bus).

Більшість функцій у Vert.x-Web реалізовано як обробники, тож завжди можна замінити реалізацію на власну.

Для використання Vert.x-Web потрібно спершу підключити відповідну залежність. У gralde це відбувається добавлянням наступного рядка в файл build.gradle:

```
implementation('io.vertx:vertx-web:4.4.1').
```

Сам Vert.x-Web використовує та надає API з ядра Vert.x, проте значно розширює можливості.

2.7 Використання маршрутизатора (Router)

Маршрутизатор є ключовим елементом Vert.x-Web. Він представляє об'єкт, який містить набір маршрутів (Routes). Коли запит надходить на сервер, маршрутизатор забезпечує збереження тіла запиту та знаходить перший відповідний маршрут для нього. Після знаходження маршруту запит передається до пов'язаного з ним обробника. Обробник може виконати необхідну логіку та повернути відповідь, або передати запит іншому відповідному обробнику.

Маршрутизатор створюється за допомогою конструктора, в який передається екземпляр класу Vertx. Після цього можна створити маршрут за допомогою методу route. При цьому якщо не вказувати додаткові критерії, то він буде відповідати усім запитам, які надходять на сервер. У цей метод також потрібно передати обробник, який, у цьому випадку, буде викликано для всіх запитів, які надходять на сервер.

Об’єкт, який приймає обробник, – це контекст маршрутизації (RoutingContext). Він містить стандартні (тобто з ядра Vert.x) HttpServerRequest, HttpServerResponse, а також інші корисні речі, які спрощують роботу з Vert.x-Web.

Для кожного запиту, який маршрутизується, існує унікальний екземпляр контексту маршрутизації, і той самий екземпляр передається всім обробникам цього запиту (якщо їх більше одного).

Після налаштування всіх обробників потрібно повідомити об'єкту Vertx, що всі запити буде обробляти об'єкт Router. Це робиться за допомогою методу handle, в який передається екземпляр маршрутизатора.

Коли Vert.x-Web маршрутизує запит до відповідного маршруту, викликається обробник маршруту, який отримує екземпляр RoutingContext. Один маршрут може мати кілька обробників, які можуть бути додані за допомогою методу handler. Якщо відповідь не завершена в першому обробнику, потрібно викликати метод next, щоб інші відповідні маршрути могли обробити запит (якщо такі є). Не обов'язково викликати метод next до завершення роботи обробника. Його можна викликати з затримкою в декілька секунд, якщо це необхідно. Щоб завершити обробку запиту, потрібно викликати метод end.

Іноді може знадобитися щось зробити в обробнику, що може заблокувати цикл подій на деякий час, наприклад, викликати блокуючий API або виконати деякі великі обчислення. Оскільки не можна робити це у звичайному обробнику, тому у Vert.x є можливість установлювати блокуючі обробники на маршруті. Блокуючий обробник виглядає так само як звичайний обробник, але його викликає Vert.x за допомогою потоку з робочого пулу без використання циклу подій. Щоб встановити блокуючий обробник на маршруті, потрібно використати метод blockingHandler. Сигнатура методу така ж як методу handle. За замовчуванням будь-які блокуючі обробники, що виконуються в тому самому контексті (наприклад, у тому самому екземплярі вершини), будуть впорядковані. Це означає, що наступний не буде виконано доки не завершиться попередній.

Маршрут можна налаштувати таким чином, щоб він відповідав шляху URI запиту. Це означає, що маршрут буде відповідати на будь-який запит, який має шлях, що збігається з вказаним шляхом. Для цього можна передати шлях в конструктор маршруту або у виклик методу path для відповідного маршруту.

Маршрут можна налаштувати для відповіді на шлях, який починається з певного «підшляху», якщо передати цей підшлях у

конструктор маршруту. Наприклад, якщо передати «/some/path/*», то цей маршрут буде відповідати на шляхи «/some/path/foo» та «/some/path/other/abc». Також можна вказувати шлях за допомогою регулярного виразу (RegExp).

У шляху також можна вказати параметри, які потім будуть доступні в об'єкті pathParam контексту маршруту. Щоб визначити параметр, слід вставити двокрапку в шляху. Наприклад, якщо вказати шлях як «/catalogue/products/:productType/:productID/», то параметрами будуть productType та productID.

Маршрут також можна налаштувати відповідно до методу запитів. За замовчуванням він відповідатиме на всі методи HTTP. Проте, якщо є потреба, щоб маршрут збігався лише з певним методом HTTP, можна передати метод в конструктор маршруту, або ж у виклик методу method для відповідного маршруту.

Також можна вказати, що маршрут буде відповідати відповідним запитам типу MIME, використовуючи метод consumes. У цьому випадку запит міститиме content-type заголовок із зазначенням типу MIME тіла запиту. Це буде зіставлено зі значенням, указаним у consumes. По суті, consumes описує, які типи MIME може використовувати обробник даного маршруту. Зіставлення можна виконати за точним збігом MIME типу або ж за певним патерном, наприклад, лише за типом, або лише за підтипом MIME типу.

Аналогічно можна вказати, що маршрут буде відповідати типам, які приймає клієнт, використовуючи метод produces. У цьому випадку запит міститиме accept заголовок для визначення того, які типи MIME відповіді прийнятні для клієнта. Заголовок accept може містити декілька MIME типів, розділених символом «,». Також MIME типи можуть мати q-значення, додане до них, яке означає вагу, яку слід застосувати, якщо

доступний більше ніж один тип MIME відповіді, що відповідає заголовку accept. Значення q – це число від 0 до 1,0. Якщо відсутнє, тоді за замовчуванням 1,0. Якщо сервер може дати більш ніж один із прийнятних типів відповіді, він повинен віддати той, в якого значення q більше. Зіставлення можна виконати за точним збігом MIME типу, або ж за певним патерном, наприклад лише по типу, або лише по підтипу MIME типу.

Всі ці критерії можна використовувати, як по одному, так і комбінувати різними способами. Якщо жоден маршрут не відповідає жодному конкретному запиту, Vert.x-Web сповістить про помилку залежно від невідповідності:

- 404 – якщо жоден маршрут не відповідає шляху;
- 405 – якщо маршрут відповідає шляху, але не відповідає методу HTTP;
- 406 – якщо маршрут відповідає шляху та методу, але він не може надати відповідь із відповідним Accept заголовком типу вмісту;
- 415 – якщо маршрут відповідає шляху та методу, але він не може прийняти Content-type;
- 400 – якщо маршрут відповідає шляху та методу, але він не може приймати порожнє тіло.

Також можна вручну керувати цими помилками за допомогою errorHandler

За замовчуванням маршрути зіставляються в порядку їх додавання до маршрутизатора. Коли надходить запит, маршрутизатор проходить по кожному маршруту і перевіряє, чи він збігається. Якщо збігається, тоді буде викликано обробник для цього маршруту. Якщо обробник у процесі обробки викликає метод next для контексту, тоді запуститься обробник для наступного відповідного маршруту (якщо такий є). І так далі. За бажанням

можна змінити типовий порядок для маршрутів. Це робиться за допомогою методу order, передавши туди цілочисельне значення.

Маршрутам призначається порядок під час створення, який відповідає порядку, в якому вони були додані до маршрутизатора, з номером першого маршруту 0, другого маршруту з номером 1, і так далі. Вказавши порядок для маршруту, можна змінити порядок за замовчуванням. Порядок маршруту також може бути негативним. Це корисно, щоб бути впевненим, що маршрут виконається перед маршрутом з номером 0. Якщо ж два маршрути будуть мати однаковий порядковий номер, то першим виконається той, який був доданий раніше. Також за бажанням можна вказати, що маршрут повинен оброблятись останнім. Це відбувається за допомогою методу last.

Також однією із корисних можливостей в маршрутизації запитів є використання підмаршрутизаторів. Для цього треба підключити маршрутизатор до іншого маршрутизатора. Встановлений маршрутизатор називається підмаршрутизатором. Підмаршрутизатори можуть використовувати інші підмаршрутизатори, тобто за бажання можна мати кілька рівнів підмаршрутизаторів.

Для того, щоб підключити маршрутизатор до іншого маршрутизатора, його потрібно спершу створити. Створюється він так само як звичайний маршрутизатор. Після цього, в «батьківському» маршрутизаторі створюється маршрут за певним шляхом (наприклад, «/productsAPI/*»), і для цього маршруту викликається метод subRouter, в який передається підмаршрутизатор. Тоді всі шляхи з підмаршрутизатора будуть доступні із префіксом productsAPI.

2.8 Обробка тіла запиту з використанням BodyHandler

BodyHandler – це обробник, який дозволяє отримати тіло запиту, обмежити його розмір та обробити завантажені файли. Щоб мати можливість отримати тіло запиту в зручному форматі, обробник тіла має бути доданий до відповідного маршруту для будь-яких запитів, які потребують тіла запиту.

Після того, як тіло запиту буде оброблено, ми зможемо отримати його в зручному форматі. Якщо ми знаємо, що це JSON, то можна використовувати метод getBodyAsJson для його отримання у вигляді JSON. Якщо ми впевнені, що це рядок, то можна використовувати метод getBodyAsString. А якщо ми хочемо отримати тіло запиту як буфер, то слід використовувати метод getBody.

Для захисту від DDoS-атак можна обмежити розмір тіла запиту. Для цього в BodyHandler можна використовувати метод setBodyLimit, щоб встановити максимальний допустимий розмір тіла запиту в байтах. Це допоможе уникнути перевищення ліміту пам'яті при обробці дуже великих тіл запиту. Якщо хтось надішле тіло, розмір якого перевищує ліміт, то сервер поверне статус код 413 «Request Entity Too Large».

Обробник тіла також може використовуватись для обробки завантажень файлів, які складаються з кількох частин. Якщо обробник тіла знаходиться на відповідному маршруті, то будь-які завантажені файли автоматично зберігаються в каталозі для завантажень (за замовчуванням – file-uploads). Кожен файл отримує унікальне ім'я (UUID, Universally Unique Identifier), інформація про завантажені файли доступна через контекст маршрутизації за допомогою fileUploads. Кожне завантаження файлу описується об'єктом FileUpload, який містить інформацію про ім'я та розмір завантаженого файлу.

2.9 Автентифікація / авторизація у Vert.x

Vert.x постачається з деякими готовими обробниками для обробки як автентифікації, так і авторизації [11]. У екосистемі Vert.x значення цих 2 слів такі:

- автентифікація – повідомляє, хто є користувачем;
- авторизація – повідомляє, що користувачеві дозволено робити.

Сам Vert.x підтримує наступні види автентифікації:

- базова автентифікація HTTP;
- дайджест-автентифікація HTTP;
- автентифікація OAuth2;
- тощо.

Якщо є потреба, щоб усі запити до шляхів (крім тих, по яких користувач може автентифікуватись) підлягали автентифікації, тоді потрібно, щоб обробник автентифікації передував всім обробникам на цих шляхах. Якщо обробник автентифікації успішно автентифікував користувача, він впровадить об’єкт User у RoutingContext, щоб він був доступний в обробниках за допомогою методу user. Отримавши об’єкт користувача, також можна програмно використовувати його методи для авторизації користувача.

Щоб створити обробник авторизації, потрібно екземпляр AuthenticationProvider. Провайдер автентифікації використовується для автентифікації користувачів. Vert.x надає декілька екземплярів постачальника автентифікації, наприклад:

- Oauth2 Auth;
- JWT Auth;
- OTP (One Time Password) Auth;
- Webauthn Auth;

– тощо.

Розглянемо на прикладі JWT Auth [12]. Він дозволяє створювати та перевіряти JWT-токени. Цей модуль підтримує різні алгоритми підпису, такі як HS256 (з використанням HMAC-SHA256), RS256 (з використанням RSA-SHA256) та інші.

Для генерації JWT-токена в Vert.x можна використовувати класи JWTAuth та JWTOptions. JWTAuth – це клас, який дозволяє створювати та перевіряти JWT-токен. JWTOptions – це клас, який дозволяє налаштовувати параметри JWT-токена, такі як час життя токена, алгоритм підпису та інші.

Для перевірки токена JWT в Vert.x можна використовувати метод authenticate класу JWTAuth. Таким чином, Vert.x дозволяє легко реалізувати різні механізми автентифікації та авторизації, використовуючи різні інструменти, такі як бази даних, LDAP (Lightweight Directory Access Protocol), OAuth2 та інші. Для цього Vert.x має різноманітні модулі, які дозволяють використовувати ці інструменти.

Наприклад, для автентифікації з використанням бази даних можна використовувати модуль «vertx-auth-jdbc». Цей модуль дозволяє підключитись до бази даних та здійснювати автентифікацію користувачів за допомогою SQL-запитів.

Для автентифікації з використанням LDAP можна використовувати модуль «vertx-auth-ldap». Цей модуль дозволяє підключитись до LDAP-сервера та здійснювати автентифікацію користувачів з використанням протоколу LDAP.

Для автентифікації з використанням OAuth2 можна використовувати модуль «vertx-auth-oauth2». Цей модуль дозволяє використовувати різні провайдери OAuth2, такі як Google, Facebook, Twitter та інші.

2.10 Доступ до бази даних у Vert.x

Оскільки нам не підходить звичайний JDBC через те, що він блокує потік, у Vert.x використовуються власні клієнти для бази даних. Оскільки PostgreSQL є одним з найпопулярніших, то розглянемо доступ до бази даних на прикладі vertx-pg-client [13].

Reactive PostgreSQL Client є клієнтом для PostgreSQL з простим API, спрямованим на масштабованість та ефективність в роботі. Цей клієнт є реактивним та неблокуючим, що дозволяє обробляти багато підключень до бази даних за допомогою одного потоку. Для підключення vertx-pg-client до застосунку Vert.x потрібно додати залежність до проєкту та сконфігурувати підключення до бази даних.

Клас PgPool є основним елементом бібліотеки vertx-pg-client і надає методи для взаємодії з базою даних. Для створення екземпляра класу PgPool необхідно використовувати конфігураційний об'єкт PgConnectOptions, який містить важливу інформацію про параметри підключення до бази даних, такі як URL бази даних, ім'я користувача та пароль.

За допомогою PgPool можна виконати запит до бази даних, виконавши метод query, передавши в нього SQL запит. Коли запит завершиться, Vert.x викличе функцію зворотного виклику з результатами виконання запиту. Якщо запит успішний, можна повернути результат виконання, інакше кинути помилку.

Крім методу query(), PgClient також надає методи для виконання інших запитів, таких як insert(), update() та delete(). Кожен з цих методів повертає об'єкт Future, який можна використовувати для обробки результатів запиту асинхронно.

Окрім бази даних PostgreSQL, Vertx-pg-client також підтримує інші бази даних, такі як MySQL, Oracle та Microsoft SQL Server, за допомогою відповідних драйверів.

2.11 Огляд бібліотеки Kodein-di для впровадження залежностей

У сучасному програмуванні впровадження залежностей (DI) є важливою складовою розробки програмного забезпечення. Воно допомагає розробникам зменшувати залежності між класами, що полегшує тестування та підтримку коду.

Kodein-DI – це легка і проста у використанні бібліотека для управління залежностями в програмах на Kotlin [14]. Вона пропонує зручний і читабельний синтаксис для впровадження залежностей та має мінімальні вимоги до конфігурації. Завдяки цьому, Kodein-DI є популярним інструментом для розробки на Kotlin.

Код, написаний з використанням Kodein-DI, має декларативний стиль. Розробники не повинні вказувати, як саме і коли створювати об'єкти, оскільки Kodein-DI забезпечує автоматичну ін'єкцію залежностей. Це спрощує розробку та робить код більш зрозумілим та простим для підтримки.

Одним з ключових елементів Kodein-DI є контейнер залежностей. Контейнер є відповідальним за зберігання об'єктів та їхніх залежностей, а також за забезпечення їхнього створення та ін'єкції у відповідні частини програми. Контейнер створюється за допомогою функції Kodein{}, де можна визначити всі залежності, які потрібні програмі.

Для того, щоб зареєструвати залежності у контейнері, розробники можуть скористатися функцією bind(). Ця функція дозволяє вказати тип залежності, яку потрібно зареєструвати, та її фабрику. Фабрика відповідає

за створення об'єкта та його налаштування перед інжектуванням, а реєстр зберігає всі необхідні залежності. Крім того, Kodein-DI має можливість встановлення правил залежностей на рівні модулів. Це дозволяє створювати модулі, які містять в собі залежності та їх правила, і легко включати їх до головної конфігурації. Таким чином, Kodein-DI забезпечує гнучкість та розширюваність конфігурації залежностей.

Однією з переваг Kodein-DI є те, що він підтримує використання інтерфейсів для реєстрації залежностей. Це дозволяє використовувати більш абстрактні типи даних та спрощує зміну реалізації залежності без необхідності змінювати код клієнта.

Однак Kodein-DI має деякі недоліки. Наприклад, він не надає можливість автоматичного виявлення залежностей та їх встановлення, як це робить Spring Framework, тобто, коли додаємо новий клас в застосунок, потрібно додавати його в конфігурацію. Крім того, Kodein-DI не надає вбудованої підтримки для використання анотацій для встановлення залежностей, що може затримати розробку проєкту.

РОЗДІЛ 3. ВИКОРИСТАННЯ ФРЕЙМВОРКУ VERT.X ДЛЯ РОЗРОБКИ ВЕБСЕРВІСУ

Зазвичай вебсервіси окрім серверної частини потребують клієнтської частини. Оскільки розробка клієнтської частини не є завданням даної роботи, було прийняте рішення використати уже розроблену клієнтську частину на попередніх курсах навчання, виправивши незначні помилки. Розроблений сервіс є соціальною мережею для людей, які люблять грати в настільні ігри. За допомогою нього користувач може переглянути всю інформацію про певну настільну гру, запланувати гру з друзями, поспілкуватись з ними в чаті та багато іншого.

3.1 Ініціалізація проєкту

Проєкт буде використовувати gradle для збірки. Оскільки наш застосунок буде містити клієнтську і серверну частини, було використано модульну структуру, в якій один модуль – це вже розроблена клієнтська частина, а інший модуль – це серверна частина, яку ми будемо розробляти в контексті цієї роботи.

Щоб не створювати проєкт вручну, можна використати генератор проєктів на сайті start.vertx.io (рисунок 6). Він створить проєкт з вказаним GroupId та ArtifactId і додасть вказані залежності вибраної версії Vert.x.

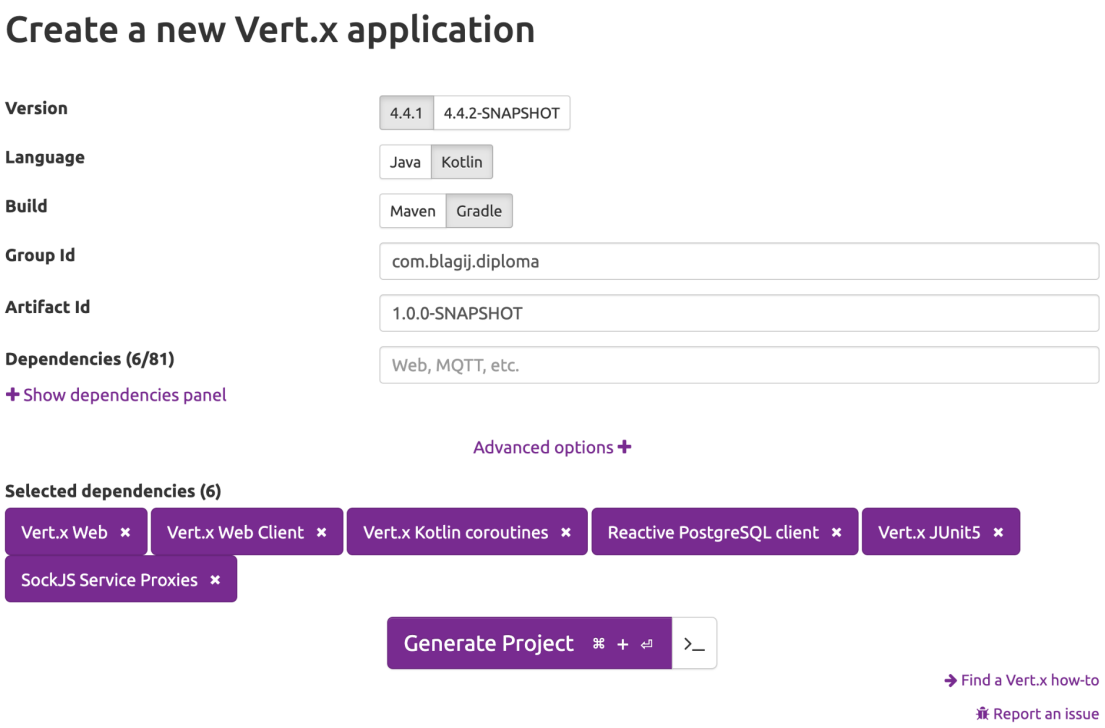

Рисунок 6 – Ініціалізація проєкту на сайті start.vertx.io

3.2 Реалізація інфраструктурного коду для подальшої розробки

Оскільки фреймворк vertx-web не є настільки зручним як конкуренти, було прийняте рішення покращити взаємодію із ним шляхом написання додаткового інфраструктурного коду.

3.2.1 Впровадження бібліотеки Kodein

Оскільки наш застосунок буде використовувати Kodein для впровадження залежностей, відразу інтегруємо його в застосунок. Проте для зручності реалізуємо можливість автоматичного підтягування залежностей в класи, щоб при додаванні нової залежності до певного

класу не довелось редагувати конфігурацію створення екземпляра цього класу. Для цього було додано наступні функції розширення (рисунок 7).

```
fun collectParams(dkodein: DKodein, construct: KFunction<Any>): Map<KParameter, Any?> {

    val className = (construct.returnType.classifier as KClass<*>).simpleName!!

    return construct.parameters.associateWith {
        val typeToken = TT(it.type.classifier as KClass<out Any>)
        if (dkodein.AllFactories(TT(String::class), typeToken).isNotEmpty()) {
            dkodein.AllFactories(TT(String::class), typeToken).first().invoke(className) ^associateWith
        } else {
            dkodein.Instance(typeToken) ^associateWith
        }
    }
}


inline fun <reified T : Any> Kodein.Builder.single(tag: Any? = null, overrides: Boolean? = null) {
    bind(tag, overrides) from singleton {
        T::class.primaryConstructor!!.callBy(collectParams(dkodein, T::class.primaryConstructor!!))
    }
}

inline fun <reified T : Any, reified K : T> Kodein.Builder.single() {
    bind<T>() with singleton {
        K::class.primaryConstructor!!.callBy(collectParams(dkodein, K::class.primaryConstructor!!))
    }
}

inline fun <reified T : Any> Kodein.Builder.single(noinline creator: NoArgSimpleBindingKodein<T>.() -> T) {
    bind<T>() with Singleton(scope, contextType, generic(), refMaker: null, sync: true, creator)
}
```

Рисунок 7 – Функції розширення Kodein Builder

Було додано три методи single з різними сигнатурами, які додають екземпляр класу в Kodein контейнер залежностей. Перший метод приймає клас, який потрібно створити, та за допомогою рефлексії отримує список параметрів для головного конструктора цього класу, і для кожного параметра за допомогою методу collectParams отримує екземпляр із контейнера залежностей. Другий метод працює аналогічно, але дозволяє зареєструвати один клас (або інтерфейс), а реалізацію використати дочірнього класу. Третій метод приймає на вхід крім класу ще й лямбда-вираз, який генерує цей екземпляр класу, і реєструє його. Він

використовується, якщо створення екземпляра не тривіальне або потрібно зробити додаткові налаштування цього екземпляра під час створення.

Після цього можна впроваджувати Vert.x в наш проєкт. Для цього створимо головний модуль, у якому зареєструємо всі необхідні для Vertx сутності (рисунок 8). У цьому модулі створимо екземпляр класу Vertx. Оскільки він створюється за допомогою статичного методу, то використаємо метод з передачею лямбда-виразу. Аналогічно для екземпляра EventBus використаємо лямбда-вираз, проте оскільки для його створення потрібно екземпляр Vertx, то отримаємо його за допомогою методу instance. Решта методів за аналогією.

```
val kodein = Kodein { this: Kodein.MainBuilder

    single<Vertx> { Vertx.vertx() }
    single<EventBus> { instance<Vertx>().eventBus() }

    single<WebClient> { WebClient.create(instance()) }

    single<PgPool> {...}

    single<WebRouter> { WebRouter(instance()) }

    importAll(
        uiModule,
        gameModule,
        authModule,
        userModule,
        plannedGameModule,
        chatModule
    )
}
```

Рисунок 8 – Основний модуль Kodein

Також в головний модуль підтягнемо всі модулі з бізнес-логікою за допомогою методу importAll. Ці модулі повинні бути створені як дочірні, і в них міститься код для кожної компоненти нашого застосунку.

3.2.2 Створення HTTP серверу з допомогою Vert.x

Після створення екземплярів усіх потрібних класів нам потрібно розгорнути вершини у Vert.x та створити HTTP сервер. Для цього до функції main додамо наступний код (рисунок 9).

```
fun main() {
    val vertx = kodein.direct.instance<Vertx>()

    setupVerticles(kodein.direct.allInstances(), vertx, kodein)
    setupPeriodics(vertx)
    registerCodecs(vertx)
}

fun setupVerticles(verticles: List<Verticle>, vertx: Vertx, kodein: Kodein) {
    vertx.deploy(verticles).onSuccess { it: CompositeFuture!
        vertx.createHttpServer(httpServerOptionsOf(compressionSupported = true))
            .requestHandler(kodein.direct.instance<WebRouter>()).listen( port: 8080) { listen ->
                if (listen.succeeded()) {
                    log.info("HTTP server running on port 8080")
                } else {
                    log.error("Failed to start HTTP server on port 8080, '${listen.cause()}'")
                    vertx.close()
                }
            }
    }.onFailure { it: Throwable!
        log.error("Failed to deploy verticle")
        vertx.close()
    }
}


fun Vertx.deploy(verticles: List<Verticle>): CompositeFuture {
    return CompositeFuture.all(verticles.map { it: Verticle
        deployVerticle(it).onSuccess { it: String!
            log.info("Deployed: ${it::class.simpleName}")
        }
    })
}
```

Рисунок 9 – Автоматизоване розгортання вершин

За допомогою цього коду Kodein поверне всі екземпляри класу Verticle (тобто наші створені вершини), а Vert.x розгорне їх. У разі успіху ми побачимо список усіх розгорнутих вершин в консолі. Такий підхід

дозволяє нам не забути запустити щойно додану вершину, а конфігурація вершин буде відбуватись в одному місці.

Також покращимо механізм створення маршрутів (route) за допомогою додаткової обробки. В цю обробку додамо зручне отримання параметрів, обробку відповіді, помилок тощо. Маршрути будемо створювати за допомогою функції розширення від рядка (String) і в ній будемо передавати метод запиту (GET, POST тощо) та власне адресу запиту (рисунок 10).

```
fun route(address: String, handler: suspend (RoutingContext) -> Any?): Route {
    val route: Route = if (address.contains( char: ' ')) {
        with(address.split( ...delimiters: ' ')) { this: List<String>
            val method = HttpMethod.valueOf(first())
            val route = route(method, last())

            if (listOf(HttpMethod.POST, HttpMethod.PATCH, HttpMethod.PUT).contains(method)) {
                route.handler(BodyHandler.create())
            }

            route ^with
        }
    } else {
        route(address)
    }
```

Рисунок 10 – Покращення механізму створення маршрутів

У контексті додаткової обробки запиту було реалізовано ParameterValueResolver (рисунок 11), який через вже згадану рефлексію отримує всі параметри лямбда-виразу обробника, і по назві та типу цих параметрів шукає їх в запиті (наприклад, тіло запиту, контекст, ідентифікатор користувача, якщо користувач автентифікований, path paramether, query paramether, значення хедерів). За бажання обробку цих параметрів можна розширити, додавши будь-який новий тип.

```
class ParameterValueResolver(private val ctx: RoutingContext) {

    private val jsonMapper = jacksonObjectMapper()

    fun resolve(it: KParameter): Any? {
        val classifier = it.type.classifier

        return when (it.name) {
            "ctx", "context" -> ctx
            "userId", "userID" -> UUID.fromString(ctx.user().principal().getString( key: "jti"))
            "body" -> {
                when (classifier) {
                    JsonObject::class -> ctx.body().asJsonObject()
                    JsonArray::class -> ctx.body().asJsonArray()
                    String::class -> ctx.body().asString()
                    else -> {
                        try {
                            jsonMapper.readValue(ctx.body().asString(), (classifier as KClass<*>).java)
                        } catch (e: Exception) {
                            throw when {...}
                        }
                    }
                }
            }
            else ->
                mapValue(
                    ctx.anyParam(it.name!!),
                    it.type.classifier as KClass<*>,
                    it.type.isMarkedNullable
                )
        }
    }
```

Рисунок 11 – Реалізація ParameterValueResolver

Після того, як ParameterValueResolver повернув всі потрібні параметри, викликається обробник. Він зазвичай повертає якесь значення, яке потрібно повернути клієнту в зручному для нього вигляді. За бажання можна додати обробку типів, які клієнт бажає отримувати, проте в нашому випадку клієнт буде завжди отримувати JSON. Тож потрібно отримане значення обробника перетворити в JSON за допомогою jsonMapper.

У випадку якщо відбулась якась помилка під час обробки запиту, клієнт повинен отримати змістовне повідомлення про помилку, а в логах серверу повинне відображатись деталізоване повідомлення про помилку із

трасуванням стеку. Для цього всю обробку обгорнемо в try-catch та в catch секції реалізуємо вищесказану обробку (рисунок 12).

```
return route.handler { ctx ->
    GlobalScope.launch(vertx.dispatcher()) { this: CoroutineScope
        try {
            val result = handler(ctx)
            val response = ctx.response()

            response.putHeader( name: "Content-Type",  value: "application/json")

            when (result) {
                null -> response.end()
                is String -> response.end(result)
                else -> response.end(jsonMapper.writeValueAsString(result))
            }

        } catch (e: Throwable) {
            log.error("Route '$address' exception", e)

            ctx.response()
                .putHeader( name: "Content-Type",  value: "application/json")
                .setStatusCode(500)
                .end( chunk: e.message ?: "Something went wrong")
        }
    }
}
```

Рисунок 12 – Створення обробника для маршруту

Тоді запит буде виглядати наступним чином (рисунок 13).

```
"GET /api/v1/game/search/:name" { name: String ->
    httpBGGClient.searchGame(name)
}
```

Рисунок 13 – Приклад простого маршруту

3.2.3 Інтеграція з базою даних

Зазвичай серверні застосунки не можуть обійтись без інтеграції з базою даних. Як уже було сказано в підрозділі 2.10, ми будемо використовувати реактивний клієнт бази даних, а саме: vertx-pg-client. Для зручності використання було реалізовано декілька стандартних методів, які отримують запит та список параметрів, і виконують цей запит та перетворюють результат виконання в сутності потрібного типу. Для цього було реалізовано абстрактний шаблонний (generic) клас Repository, який містить два абстрактних параметри PgPool, та mapper. Вони повинні бути визначені кожним дочірнім класом. Цей клас містить методи queryFirst, query та queryBatch. Розглянемо реалізацію цих методів на основі методу queryFirst (рисунок 14). Інші методи створені за аналогією.

Цей метод приймає рядок з SQL запитом, список параметрів і mapper для перетворення результату виконання запиту на екземпляр відповідного класу та повертає екземпляр класу Future, який буде обчислюватись в асинхронному режимі. У ньому за допомогою pgPool створюється підключення до бази даних та виконується запит із переданими параметрами. Коли запит завершить виконання (onComplete), якщо результат виконання буде успішний, то метод перетворить результат виконання за допомогою переданого mapper в екземпляри потрібного класу та візьме перший із них та передасть у future. Якщо ж відбудеться якась помилка, або результат виконання буде неуспішний, тоді відповідна помилка буде передана у future для подальшої обробки.

```
suspend fun queryFirst(
    @Language("PostgreSQL") sql: String,
    params: Tuple = Tuple.tuple(),
    mapper: Row.() -> T = mapper()
): T {
    return Future.future<T> { queryFirst ->
        pgPool.connection.compose { conn ->
            conn
                .preparedQuery(sql) PreparedQuery<RowSet<Row!>!>!
                .execute(params) Future<RowSet<Row!>!>!
                .onComplete { rs ->
                    try {
                        if (rs.succeeded()) {
                            queryFirst.complete(rs.result().map { it.mapper() }.first())
                        } else {
                            logger.error(
                                sqlExceptionMessage, entries(mapOf(...))
                            )
                            queryFirst.fail(rs.cause())
                        }
                    } catch (e: Exception) {
                        queryFirst.fail(e)
                    } finally {
                        conn.close()
                    }
                }
        }
    }.await()
}
```

Рисунок 14 – Код для запиту до БД, який повертає один екземпляр

3.3 Приклад розробки серверного застосунку

Відповідно до поставленої задачі, наш вебсервіс буде використовуватись як соцмережа для гравців в настільні ігри.

3.3.1 Огляд архітектури застосунку

Розроблений серверний застосунок буде взаємодіяти з базою даних PostgreSQL, стороннім поштовим сервісом для надсилання електронних

листів та з BoardGameGeeks API. Структура застосунку зображена на рисунку 15. Для кожної з цих взаємодій використаємо відповідні бібліотеки, наприклад для взаємодії із базою даних – vertx-pg-client, а для взаємодії із поштовим сервісом – vertx-mail-client, а для взаємодії зі стороннім API – vertx-web-client.

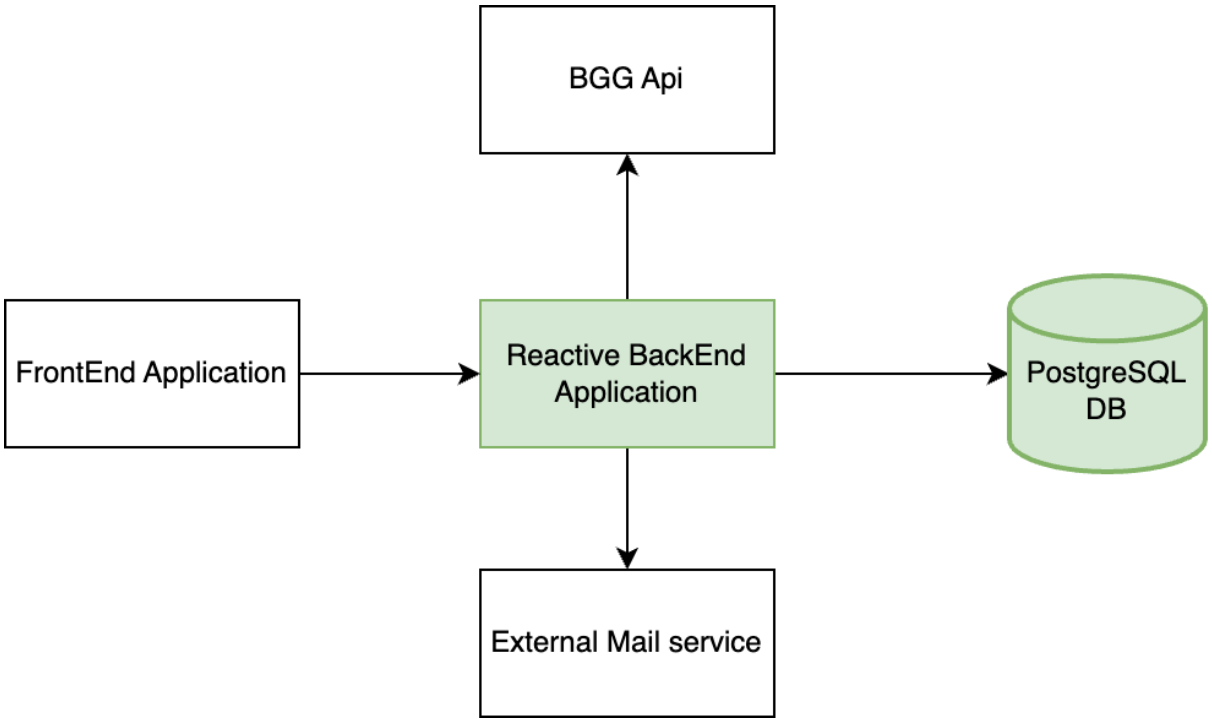

Рисунок 15 – UML діаграма розробленого застосунку

3.3.2 Взаємодія зі стороннім API для отримання інформації про настільні ігри

Інформацію про настільні ігри, як і в попередньому варіанті реалізації цього сервісу, ми будемо отримати з сервісу BoardGameGeek. Наш сервіс буде виконувати запити, приводити інформацію в зручний для клієнтської сторони вигляд та повертати її клієнту. І тут відразу скористаємось перевагою розробки на Vert.x – неблокуючі потоки. Наш

сервіс замість створення потоку, метою якого буде лише очікувати на виконання запиту, буде використовувати асинхронний HTTP client. Для цього скористаємось бібліотекою vertx-web-client.

У нас буде декілька основних запитів на BoardGameGeek, а саме: пошук настільних ігор по назві, пошук по типу або механіці та отримання детальної інформації про настільну гру. Інформацію про ці запити було отримано з документації [15] та проаналізовано запити із вебсторінки на їх сервер. Оскільки частина запитів буде в форматі JSON, а частина в XML, їх потрібно привести до одного зручного формату, а саме JSON. Для цього було використано бібліотеку org.json:json, яка надає можливість конвертувати формат XML в JSON, щоб повертати його користувачу.

Тоді запити будуть виглядати наступним чином (рисунок 16). Використовуючи створений HTTPClient, створюємо запит, передаючи повне посилання, і надсилаємо його. Після цього ми отримуємо екземпляр класу future, який обчислюється асинхронно, та очікуємо на результат за допомогою методу await. Він поверне екземпляр класу HttpResponse, в якому міститься відповідь від сервісу.

Розглянемо кожен запит трішки детальніше. Перший запит виконує пошук настільної гри по назві. Другий запит отримує настільну гру по переданому ідентифікатору, перетворює інформацію з XML в JSON, та наповнює її посиланнями на категорії та механіки. А третій запит повертає список настільних ігор по категорії або механіці.

```
suspend fun searchGame(name: String): List<Game> {
    return client.getAbs( absoluteURI: "$bggUrl/search/boardgame?nosession=1&q=$name&showcount=20") HttpRequest<Buffer!>!
        .send().await() HttpResponse<Buffer!>!
        .bodyAsString() String!
        .toGameList()
}

suspend fun getById(id: String): JSONObject {
    val response = client.getAbs( absoluteURI: "$bggUrl/xmlapi2/thing?id=$id") HttpRequest<Buffer!>!
        .send().await() HttpResponse<Buffer!>!
        .bodyAsString()

    val json: JSONObject = XML.toJSONObject(response).getJSONObject( key: "items").getJSONObject( key: "item")

    return json
        .addLinks( nameOfLink: "boardgamecategory")
        .addLinks( nameOfLink: "boardgamemechanic")
        .put("id", id)
}

suspend fun getGamesByTypes(gameTypeId: Int): JsonObject {
    return client.getAbs( absoluteURI: "$geekdoUrl&objectid=${gameTypeId}",) HttpRequest<Buffer!>!
        .send().await() HttpResponse<Buffer!>!
        .bodyAsJsonObject()
}
```

Рисунок 16 – Запити на сервіс BoardGameGeek

3.3.3 Робота з обліковими записами користувача

Користувачі можуть зареєструватись, використовуючи логін, особисту пошту та пароль. Після цього користувачу на пошту прийде посилання для активації акаунту. Тоді вершина для реєстрації буде виглядати наступним чином (рисунок 17).

```
"POST /api/v1/auth/register" { body: User ->
    userRepository.createUser(body.copy(status = "UNACTIVATED", password = encodePassword(body.password)))

    val token = Token(body.id, UUID.randomUUID().toString())
    tokenRepository.storeToken(token)

    eventBus.send( address: "EMAIL_SENDING_CHANNEL", generateMailMessage(body.mail, token.token)) ^lambda
}

"POST /api/v1/auth/activate/:token" { token: String ->
    userRepository.activateUserByToken(token)
}
```

Рисунок 17 – Вершина для реєстрації

У першому маршруті користувач передає свої облікові дані і сервер зберігає їх в базі даних. При цьому, якщо користувач з такими даними вже зареєстрований, то метод викине помилку. Після цього ми генеруємо та зберігаємо в базу даних токен для активації користувача. Цей токен будемо надсилати користувачу на пошту.

Оскільки надсилання повідомлення на пошту (а саме приєднання до SMTP серверу) може тривати певний час, щоб обробка запиту на реєстрацію не завершилась помилкою, було прийнято рішення використати шину подій. У цьому маршруті ми створюємо повідомлення, яке надсилаємо шиною подій на вказану адресу. А в іншому місці ми реєструємо обробку за даною адресою, надсилаючи лист електронною поштою (рисунок 18). Для того, щоб ми могли надіслати MailMessage шиною подій, ми під час ініціалізації застосунку реєструємо відповідний MailMessageCodec, який вміє перетворювати об'єкт MailMessage в Buffer та навпаки.

```
override suspend fun start() {
    eventBus.consumer<MailMessage>( address: "EMAIL_SENDING_CHANNEL") { message ->
        GlobalScope.launch { this: CoroutineScope
            mailClient.send(message.body())
        }
    }
}
```

Рисунок 18 – Реєстрація обробника за адресою, для надсилання листів на електронну пошту

У другому маршруті нам приходить токен, і ми активуємо користувача за цим токеном.

Також бувають ситуації, коли у токена завершився термін придатності, а користувач не підтвердив поштову адресу. У такому випадку нам потрібно видалити цей токен та користувача з бази даних. Для цього найкраще підійдуть періодичні події (periodic). Створимо такий обробник, який буде запускатися о 3 ночі та видаляти застарілі токени та користувачів з неактивованим профілем та відсутнім токеном (рисунок 19).

```
fun getListOfPeriodics(): List<Handler<Long>> {
    return listOf(
        kodein.direct.instance<CleanUpTokensPeriodic>()
    )
}

fun setupPeriodics(vertx: Vertx) {
    val timeToMidnight = (of( hour: 3,  minute: 0).toSecondOfDay() - now().toSecondOfDay()).mod( other: 86400L)
    getListOfPeriodics().forEach { periodic ->
        vertx.setPeriodic(timeToMidnight, Duration.ofDays( days: 1).toMillis()) { periodic.handle(it) }
    }
}
```

Рисунок 19 – Додавання periodic, який працює о 3 ночі

Для ідентифікації користувач надає свій логін та пароль, а ми звіряємо його із даними в базі даних, і в разі існування такого користувача, ми генеруємо JWT токен, який надсилаємо користувачу. Для цього використовується маршрут, зображений на рисунку 20. Потім, за цим токеном, ми зможемо автентифікувати користувача в системі.

```
"POST /api/v1/auth" { body: AuthRequest ->
    val user = userRepository.findUser(body)

    jwtAuth.generateToken(
        JsonObject().put("jti", user.id.toString()),
        jwtOptionsOf(
            algorithm = "HS512",
            expiresInMinutes = 60,
            subject = user.login,
        )
    ) ^lambda
}
```

Рисунок 20 – Ідентифікація користувача та генерування токена

Після того, як користувач отримав токен, він може виконувати ті запити, які недоступні неавтентифікованому користувачу. Для того, щоб наш сервіс не дозволяв неавтентифікованому користувачу користуватись відповідним функціоналом, було реалізовано наступний маршрут (рисунок 21). Як можна помітити, йому поставлений порядок мінус один. Це зроблено для того, щоб він оброблявся перед всіма іншими маршрутами. У ньому задано список публічних шляхів, і якщо шлях в запиті починається з одного із цих публічних шляхів, тоді запит можна продовжити без автентифікації користувача. В іншому випадку потрібно автентифікувати користувача, і в разі валідності токена – продовжити подальшу обробку.

```
val publicPaths = listOf(
    "/api/v1/auth",
    "/api/v1/game",
)

route().order(-1).handler { it: RoutingContext!
    val requestPath = it.request().path()

    if (publicPaths.any { path -> requestPath.startsWith(path) }) {
        it.next()
    } else {
        jwtAuthHandler.handle(it)
    }
}
```

Рисунок 21 – Реалізація маршруту для обробки автентифікації користувача

Після того, як jwtAuthHandler автентифікував користувача, він збереже інформацію про нього у контекст запиту, у поле user. Після цього у ParameterValueResolver ми додали обробку параметра userId, який буде витягувати цю інформацію із поля user із контексту (рисунок 11).

Тоді користувач зможе виконати запит на отримання інформації про себе і сервіс знатиме, про якого користувача потрібно повертати інформацію (рисунок 22)

```
"GET /api/v1/user/me" { userId: UUID ->
    userRepository.getUserByUserId(userId)
}
```

Рисунок 22 – Запит користувача, на отримання інформації про себе

ВИСНОВКИ

На сьогодні створення реактивних серверних застосунків перебуває у фазі активного розвитку, тож дослідження цього підходу, та відповідних фреймворків є дуже актуальною темою.

Під час виконання кваліфікаційної роботи було:

- досліджено стандартні технології для реалізації серверних застосунків;
- порівняно класичний та реактивний підходи для розробки серверних застосунків;
- досліджено технічні особливості фреймворків Vert.x, Vert.x-web та мови програмування Kotlin;
- імплементовано інфраструктурний код для полегшення впровадження фреймворку Vert.x
- реалізовано механізми, що забезпечують взаємодію між компонентами застосунку та їх синхронізацію;
- імплементовано реактивний серверний застосунок.

Інструментарієм було вибрано мову програмування Kotlin, фреймворки Vert.x та Vert.x-web, співпрограми (coroutine), систему автоматичної збірки Gradle, IDE Intellij IDEA, базу даних PostgreSQL.

Розроблений інфраструктурний код для створення серверного застосунку може використовуватись як основа для комерційних серверних застосунків, а також як основа для виконання лабораторних робіт з дисциплін «Об'єктноорієнтоване програмування» та «Інформаційні системи». Практичну частину можна використовувати в реальному житті для гравців в настільні ігри.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Michael P. Papazoglou. (2008). Web Services: Principles and Technology. Prentice Hall. ISBN: 978-0321155559
2. Arnaud Lauret. (2019). The Design of Web APIs. Manning. ISBN: 978-1617295102
3. Рейтинг мов програмування 2023. [Електронний ресурс] – Режим доступу: https://dou.ua/lenta/articles/language-rating-2023
4. Peter Sestoft. (2012). Programming Language Concepts (Undergraduate Topics in Computer Science). Springer. ISBN: 978-1447141556
5. Intro to reactive. [Електронний ресурс] – Режим доступу https://vertx.io/introduction-to-vertx-and-reactive/
6. The Reactive Manifesto (2014). [Електронний ресурс] – Режим доступу https://www.reactivemanifesto.org/
7. Introduction to the Kotlin Language. [Електронний ресурс] – Режим доступу https://www.baeldung.com/kotlin/intro
8. Julien Ponge. (2020). Vert.x in Action: Asynchronous and Reactive Java. Manning. ISBN 978-1617295621
9. TechEmpower. Web Framework Benchmarks . [Електронний ресурс] – Режим доступу https://www.techempower.com/benchmarks/
10. Vert.x Core Manual. [Електронний ресурс] – Режим доступу https://vertx.io/docs/vertx-core/java/
11. Common Authentication and Authorization. [Електронний ресурс] – Режим доступу https://vertx.io/docs/vertx-auth-common/java/
12. JWT Auth provider. [Електронний ресурс] – Режим доступу https://vertx.io/docs/vertx-auth-jwt/java/

13. Reactive PostgreSQL Client. [Електронний ресурс] – Режим доступу https://vertx.io/docs/vertx-pg-client/java/
14. Kotlin Dependency Injection with Kodein. [Електронний ресурс] – Режим доступу www.baeldung.com/kotlin/kodein-dependency-injection
15. BGG XML API2. [Електронний ресурс] – Режим доступу boardgamegeek.com/wiki/page/BGG_XML_API2