АНОТАЦІЯ

Кваліфікаційна робота на здобуття освітнього ступеню «бакалавр» за спеціальністю 121 – Інженерія програмного забезпечення. Тернопільський національний технічний університет ім. Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП-41, 2023 рік. Пояснювальна записка до кваліфікаційної роботи на здобуття освітнього ступеню «бакалавр» містить: 130 с., 23 рис., 1 табл., 5 додатків.

Тема: Розробка високонавантаженої системи на основі парадигми реактивного програмування з використанням мови програмування Java та застосуванням мікросервісної архітектури

Ключові слова: ВИСОКОНАВАНТАЖЕНІ ДОДАТКИ, WEB-САЙТИ, ІНТЕРНЕТ-КІНОТЕАТР, МІКРОСЕРВІСИ, ПРОТОКОЛИ, АРХІТЕКТУРА, РЕАКТИВНІСТЬ, ОНЛАЙН КІНОТЕАТР, РЕАКТИВНЕ ПРОГРАМУВАННЯ.

Метою дипломної роботи є дослідження перспектив розробки високонавантажених систем базуючись на парадигмі реактивного програмування з використанням мікросервісної архітектури. Щоб продемонструвати це, ми розробили онлайн кінотеатр. Даний вид програм дозволить нам яскраво відобразити усю суть поставленої проблеми.

У першому розділі ми проведемо аналіз програмного застосунку. Розглянемо функціональні та не функціональні вимоги. Буде розглянуто діаграми варіантів використання різних дійових осіб.

Далі, у другому розділі, буде проведено аналіз архітектури. Підберемо архітектурний стиль проєкту, використовуючи найкращі практики проектування. Також, обговоримо загальні патерни проектування, які були використані під час

розробки клієнтської та серверної частин. Також, дані частина буде відображена у вигляді діаграм.

У наступному розділі – розділі номер три – ми проаналізуємо основні технології, які було використано при розробці системи. Ми окремо розглянемо технології клієнтської та серверної частин. Також, ми проведемо аналіз парадигм при розробці серверної частини, адже це напряму стосується теми дипломної роботи. Далі, ми проаналізуємо використані бази даних та їх типи. Та на кінець, але не менш важливіше, ми обговоримо механізм взаємодії серверної та клієнтської частин.

У четвертому розділі буде розглянуто інструменти, які були використані при розробці даного проєкту. В даному розділі буде описано чому ці інструменти біли обрані та їх переваги над іншими.

Далі – у розділі номер п’ять – буде розглянуто реалізацію самого продукту, де увага буде сконцентрована на серверну частину. Будуть описані основні мікросервіси, які відповідають за базову та основну логіку застосунку. Також, буде розглянуто переваги між парадигмою реактивного програмування – парадигма, яка забезпечує асинхронне виконання – та блокуючого програмування. Будуть наведені результати порівняння двох парадигм. Додатково буде опис та приклад використання механізмів подальшої розробки та подальшої доставки системного забезпечення.

У шостому розділі буде розглянута безпека життєдіяльності та основи охорони праці, де буде розглянута роль центральної нервової системи в трудовій діяльності людини та оцінка травмонебезпеки технологічного процесу при розробці програмного забезпечення.

ANNOTATION

Qualification work for obtaining a bachelor's degree in specialty 121 - Software engineering. Ternopil National Technical University named after Ivan Pulyuya, faculty of computer and information systems and software engineering, department of software engineering, group CP-41, 2023. The explanatory note to the qualification work for obtaining the bachelor's degree contains: 130 pages, 23 figures, 1 table, 5 appendices.

Topic: Development of a highly loaded system based on the reactive programming paradigm using the Java programming language and the application of microservice architectureKeywords: HIGH LOAD APPLICATIONS, WEBSITES, INTERNET CINEMA, MICROSERVICES, PROTOCOLS, ARCHITECTURE, REACTIVITY, ONLINE CINEMA, REACTIVE PROGRAMMING.

The aim of the diploma work is to study the prospects for the development of highly loaded systems based on the paradigm of reactive programming using microservice architecture. To demonstrate this, we developed an online cinema. This type of programs will allow us to vividly reflect the essence of the problem.

In the first section, we will analyze the software application. Let's consider functional and non-functional requirements. Use case diagrams of various actors will be considered.

Further, in the second chapter, the analysis of the architecture will be carried out. We will select the architectural style of the project, using the best design practices. Also, let's discuss the general design patterns that were used during the development of the client and server parts. Also, the data part will be displayed in the form of charts.

In the next chapter, chapter number three, we will analyze the main technologies that were used in the development of the system. We will separately consider the technologies of the client and server parts. Also, we will conduct an analysis of paradigms in the development of the server part, because this is directly related to the topic of the thesis.

Next, we will analyze the used databases and their types. And finally, but no less importantly, we will discuss the mechanism of interaction between the server and client parts.

In the fourth section, the tools that were used in the development of this project will be considered. This section will describe why these tools were chosen and their advantages over others.

Next, in section number five, the implementation of the product itself will be considered, where attention will be focused on the server part. The main microservices that are responsible for the basic and core logic of the application will be described. Also, the advantages between the reactive programming paradigm – a paradigm that allows for asynchronous execution – and blocking programming will be considered. The results of the comparison of the two paradigms will be given. In addition, there will be a description and an example of the use of mechanisms for further development and further delivery of system support.

In the sixth chapter, life safety and the basics of labor protection will be considered, where the role of the central nervous system in human labor and the assessment of the risk of injury of the technological process in the development of software will be considered.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

HTTP – HyperText Transfer Protocol

SQL – Structured Query Language

URL – Uniform Resource Locator

DB – Data Base

UI – User Interface

UX – User Experience

UML – Unified Modeling Language

JSON – JavaScript Object Notation

XML – Extensible Markup Language

SPA – Single Page Application

UML – Unified Modeling Language

CI – Continues Integration

CD – Continues Delivering

AOP – Aspect-Oriented Programming

MPA – Multiple pages application

REST – Representational State Transfer

ЗМІСТ

ВСТУП

У сучасному світі, де системи стають все більшими та більшими навантаження на ці системи також ростуть. Відповідно, ціна, яка витрачається на їх обслуговування також росте. Часто ці ціни ростуть у арифметичній, а інколи і у геометричній прогресіях. Сьогодні, коли апетити користувачів ростуть, ця тема є актуальною як ніколи. Також вартість підтримки та постійного його розвитку зростає. На прикладі онлайн кінотеатру буде наведено приклад оптимізації таких систем.

Метою дипломної роботи є дослідження перспектив розробки високонавантажуваних систем базуючись на парадигмі реактивного програмування з використанням мікросервісної архітектури на прикладі створення онлайн кінотеатру. Це дозволить оптимізувати витрати на обладнання, яке призначене для підтримки даної програми та його розробку.

Ці фактори є двома найважливішими аспектами розробки програмного забезпечення, які сприяють загальному успіху та ефективності програмної системи.

Розширюваність означає легкість, з якою програмна система може бути розширена або модифікована для пристосування до нових функцій, функцій або змін у вимогах. Він передбачає розробку архітектури програмного забезпечення та кодової бази в модульній та гнучкій манері.

Програмна система з хорошою можливістю розширення може адаптуватися до мінливих потреб бізнесу, технологічного прогресу та вимог користувачів. Це дозволяє майбутні вдосконалення та дозволяє програмному забезпеченню залишатися актуальним і конкурентоспроможним у середовищі, що швидко змінюється. У міру зростання бази користувачів або збільшення робочого навантаження розширювана система програмного забезпечення може бути

масштабована, щоб впоратися з додатковим навантаженням. Це дозволяє додавати нові модулі або компоненти без серйозних збоїв у існуючій системі.

Продуктивність означає, наскільки добре працює програмна система з точки зору швидкості, оперативності, ефективності та використання ресурсів. Він охоплює такі фактори, як час відповіді, пропускна здатність, затримка, використання пам'яті та використання ЦП.

Це безпосередньо впливає на досвід користувача. Користувачі очікують, що програми будуть швидкими, оперативними та ефективними. Повільне або невідповідне програмне забезпечення може призвести до розчарування, зниження продуктивності та поганого сприйняття програмного продукту чи послуги. Високопродуктивні програмні системи створюють задоволених клієнтів, які, швидше за все, продовжуватимуть використовувати програмне забезпечення та рекомендуватимуть його іншим. Позитивний досвід користувачів покращує утримання клієнтів і лояльність, сприяючи успіху та зростанню програмного продукту чи послуги.

Створення онлайн кінотеатру неймовірно захоплююче завдання, адже це вимагає досить глибоких знань роботи мережі та великої уважності під час розробки системи, щоб спромогтись максимальної оптимізації та швидкості роботи. Створення онлайн-кінотеатру – це захоплююче заняття, яке дозволяє відтворити магію кіно до світової аудиторії. Із зростанням популярності потокових сервісів і розвитком технологій попит на платформи онлайн-кінотеатрів значно зріс.

РОЗДІЛ 1 АНАЛІЗ ВИМОГ ДО ПРГРАМНОГО ПРОДУКТУ

1.1 Аналіз функціональних вимог

Щоб відобразити функціональні вимоги дійових осіб на сайті, буде спроектовано діаграму варіантів використання. Діаграма варіантів використання — це візуальне представлення функціональних вимог системи з точки зору її користувачів. Він ілюструє взаємодію між системою та її користувачами або зовнішніми об’єктами. Діаграми варіантів використання зазвичай використовуються в розробці програмного забезпечення для фіксації та передачі інформації про поведінку та функціональність системи.

Діаграми варіантів використання є корисним інструментом для розуміння та документування вимог і функціональності системи, сприяння спілкуванню між зацікавленими сторонами та служать основою для подальшого розвитку та аналізу.

Реалізовану діаграму варіантів використання для дійової особи “Користувач” ви можете побачити на рисунку 1.1.

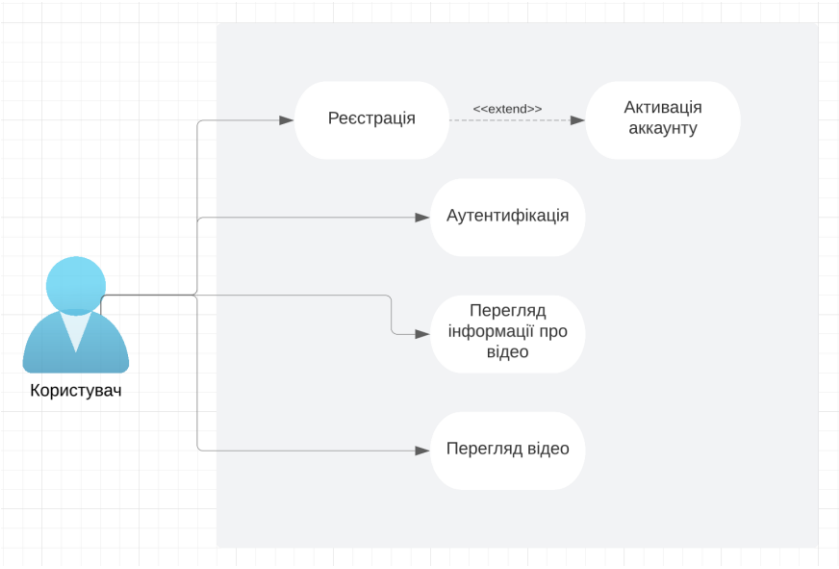

Рисунок 1.1 – Діаграма варіантів використання дійової особи “Користувач”

На рисунку 1.1 ви можете побачити діаграму варіантів використання дійової особи “Користувач”. Користувач матиме можливість пройти реєстрацію, після чого підтвердити свої данні, ввівши код, який прийде на електрону адресу. Після чого ця дійова особа зможе авторизуватись та переглядати відомості про відео або безпосередньо саме відео.

На етапі реєстрації користувач повинен ввести такі персональні дані як: електрону адресу, ім’я користувача, яка буде відображатись на сайті та використовуватиметься для звернення, пароль та підтвердження коректності паролю.

Після введення персональних даних під час реєстрації система повинна перевірити їх на правильність. Наприклад, адреса електронної пошти має бути дійсною, а пароль має відповідати вимогам безпеки, таким як мінімальна довжина, використання різних символів тощо. Якщо дані не відповідають вимогам, користувача буде попереджено з відповідними помилками.

Після успішної реєстрації на вказану електронну адресу користувача буде надіслано код підтвердження. Цей код буде використовуватися для перевірки правильності введених даних і підтвердження електронної адреси. Користувачеві необхідно буде ввести отриманий код для завершення процесу реєстрації.

Після авторизації, користувач матиме доступ до особистого облікового запису, де зможе переглядати та змінювати свої персональні дані. Крім того, в обліковому запису користувача будуть доступні відомості про відео, такі як заголовок, опис, рейтинг, коментарі та інша відповідна інформація. Користувач зможе переглядати ці відомості та за потреби коментувати або оцінювати відео. Крім перегляду відомостей про відео, користувач зможе переходити безпосередньо до перегляду самого відео.

Також, для повноцінної роботи сайту необхідно такої дійової особи як “Адміністратор”. На рисунку 1.2 ви можете побачити діаграму варіантів використання даної дійової особи.

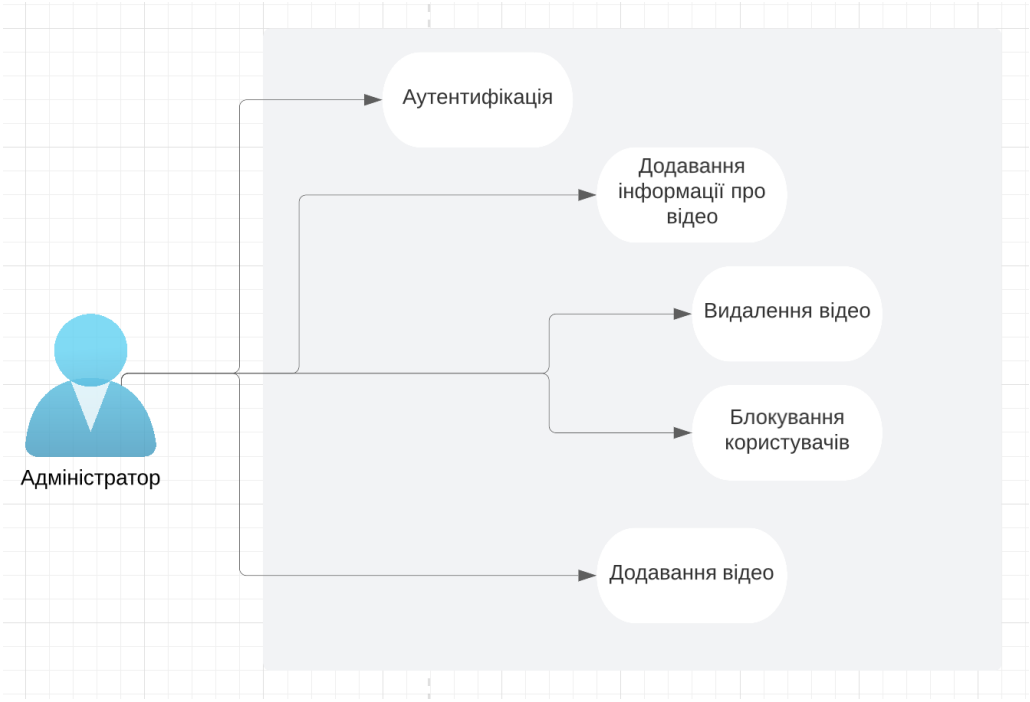

Рисунок 1.2 – Діаграма варіантів використання дійової особи “Адміністратор”

На рисунку 1.2 ви можете побачити діаграму варіантів використання дійової особи “Адміністратор”. Адміністратор матиме можливість авторизуватись та переглядати відомості про відео або безпосередньо саме відео. Також він зможе редагувати самі відео або відомості про них. Адміністратору надається можливість блокування користувачів.

Після завершення процесу авторизації адміністратор отримає доступ до свого особистого кабінету, де він зможе виконувати різні дії, пов’язані з керуванням контентом і користувачами.

У своєму обліковому записі адміністратор зможе переглядати деталі відео, такі як назва, опис, рейтинг, коментарі та інша відповідна інформація. Це дозволить адміністратору оцінювати та аналізувати вміст, який публікується на сайті чи платформі.

Дана дійова особа матиме можливість редагувати відео, зокрема змінювати вміст, додавати або видаляти відеокліпи тощо. Крім того, він зможе редагувати

інформацію про відео, таку як назва, опис та інші деталі, щоб надати відповідну та точну інформацію користувачів.

Адміністратор має можливість блокувати користувачів, які порушують правила сайту або платформи. Це може включати зловживання, неприйнятну поведінку або поширення невідповідного вмісту. Блокування дозволяє адміністратору припинити доступ таких користувачів до контенту та функцій сайту.

1.2 Аналіз нефункціональних вимог

Нефункціональні вимоги – це критерії, які визначають продуктивність, зручність використання, безпеку та інші якості системи або програмного додатку, а не його конкретні функціональні можливості. Вони описують, як система має поводитись чи працювати, а не те, що вона має робити. Нефункціональні вимоги, як правило, стосуються таких атрибутів, як надійність, масштабованість, доступність, ремонтопридатність і продуктивність.

Вимоги до продуктивності визначають очікуваний час відгуку, пропускну здатність і використання ресурсів системи. Це включає такі фактори, як максимально допустимий час відгуку для певних операцій, здатність системи обробляти певну кількість одночасних користувачів або транзакцій, а також ефективність використання ресурсів, таких як пам’ять або потужність обробки. Тестування та оптимізація продуктивності часто проводяться, щоб переконатися, що система відповідає цим вимогам і добре працює за очікуваних умов робочого навантаження.

Вимоги до зручності використання зосереджені на досвіді користувача та простоті використання системи. Вони охоплюють такі фактори, як дизайн інтерфейсу користувача, інтуїтивність, простота та доступність. Вимоги до зручності використання можуть визначати такі критерії, як кількість кроків, необхідних для

виконання завдання, чіткість повідомлень про помилки та підтримка різних мов або допоміжних технологій. Відгуки користувачів, тестування зручності використання та принципи дизайну інтерфейсу користувача зазвичай використовуються для задоволення цих вимог.

Вимоги безпеки стосуються захисту системи, її даних і користувачів від несанкціонованого доступу, зломів і вразливостей. Ці вимоги охоплюють такі сфери, як автентифікація, авторизація, шифрування даних, безпечні протоколи зв'язку та журнали аудиту. Вимоги до безпеки визначаються на основі конфіденційності інформації, що обробляється, і дотримання відповідних стандартів або правил. Моделювання загроз, оцінка вразливості та тестування на проникнення часто проводяться, щоб переконатися, що система відповідає необхідним стандартам безпеки.

Вимоги до масштабованості стосуються здатності системи обробляти зростаючі робочі навантаження, обсяги даних або кількість користувачів без значного зниження продуктивності. Це включає можливість додавати більше апаратних ресурсів, розподіляти робоче навантаження між декількома серверами або вертикально чи горизонтально масштабувати. Вимоги до масштабованості часто включають такі міркування, як балансування навантаження, розподіл даних, механізми кешування та кластеризація. Тестування, моніторинг і налаштування продуктивності необхідні для забезпечення ефективного масштабування системи.

Вимоги до підтримуваності зосереджені на простоті обслуговування, модифікації та оновлення системи протягом її життєвого циклу. Ці вимоги стосуються таких факторів, як читабельність коду, модульність, розширюваність, документація та дотримання стандартів кодування. Вимоги до ремонтопридатності спрямовані на мінімізацію часу простою під час оновлень, полегшення виправлення помилок і спрощення додавання нових функцій або інтеграції із зовнішніми системами. Належна документація, контроль версій сприяють дотриманню вимог щодо зручності обслуговування.

Вимоги до сумісності визначають здатність системи інтегруватися та працювати з іншими системами, платформами або програмними середовищами. Це включає взаємодію з певними операційними системами, базами даних, браузерами або програмними компонентами сторонніх виробників. Вимоги до сумісності можуть включати дотримання галузевих стандартів або протоколів, підтримку певних форматів файлів або можливість інтеграції через API (інтерфейси прикладного програмування) або веб-служби. Тестування на сумісність часто виконується для забезпечення безперебійної взаємодії між системою та зовнішніми об’єктами.

Нефункціональні вимоги дають рекомендації щодо продуктивності системи. Вони визначають такі фактори, як час відгуку, пропускна здатність, масштабованість і використання ресурсів. Встановлюючи цілі продуктивності та обмеження, нефункціональні вимоги допомагають забезпечити ефективну роботу системи та відповідність очікуванням щодо продуктивності користувачів і зацікавлених сторін. Нефункціональні вимоги, пов’язані з надійністю та доступністю, допомагають забезпечити послідовне функціонування системи та її доступність у разі потреби. Ці вимоги визначають такі фактори, як час роботи системи, відмовостійкість, обробка помилок і механізми аварійного відновлення. Розглядаючи ці аспекти, нефункціональні вимоги сприяють загальній стабільності та надійності системи.

Виконання цих нефункціональних вимог має вирішальне значення для розробки надійних і високоякісних програмних систем, які відповідають очікуванням користувачів, працюють оптимально та відповідають стандартам безпеки та сумісності.

РОЗДІЛ 2 АНАЛІЗ АРХІТЕКТУРИ ПРИ РОЗРОБЦІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Підбір архітектури проєкту

Підбір архітектури для програмного забезпечення є важливим етапом в процесі його розробки. Неправильний вибір архітектури може призвести до проблем зі розширюваністю, продуктивністю, безпекою та збереженням коду в майбутньому.

Для правильного вибору архітектури, необхідно провести дослідження предметної області та визначити вимоги до програмного забезпечення. Важливо враховувати, які функції повинні бути доступні для користувачів, які сервіси повинні бути використані, які дані повинні бути оброблені та збережені, і як повинні бути розгорнуті програмні компоненти.

Враховуючи те, що користувач нашої програми має повноцінно працювати з нашою системою та отримувати хороший досвід роботи з нею, клієнт-серверна архітектура підійде нам як ніколи. Даний вид архітектури є однією з найбільш поширених архітектур при розробці програмного забезпечення. Вона дозволяє розділити функціональність програми на дві.

На рисунку 2.1 наведено схематичний вигляд даної архітектури.

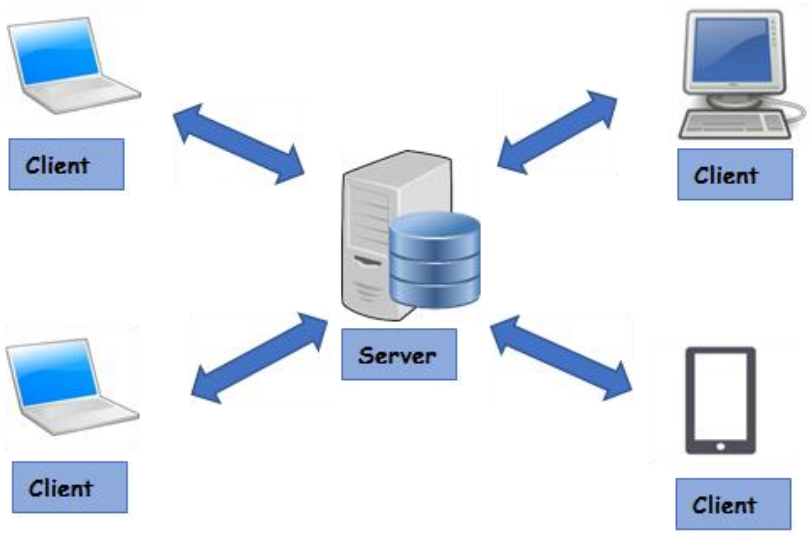

Рисунок 2.1 – Схематичний вигляд клієнт-серверної архітектури

Переглянувши рисунок 2.1 можна зрозуміти концепцію даної архітектури. Клієнти надсилають запити на сервер та отримують відповідь від нього у вигляді певного медіа контенту. У ролі клієнта може виступати веб сторінка або будь-яка додаток, який встановлений на комп’ютер чи смартфон.

Однією з головних переваг цієї архітектури є можливість розподілу ресурсів та завдань між клієнтом та сервером. Це дозволяє зменшити навантаження на клієнтську частину програми та забезпечити оптимальне використання ресурсів сервера. Крім того, використання клієнт-серверної архітектури дозволяє забезпечити більшу безпеку даних та захист від зловживань.

Основними поняттями даної архітектури є клієнтська та серверні частини — це терміни, які використовуються програмістами та комп’ютерними професіоналами для опису рівнів, які складають апаратне забезпечення, комп’ютерну програму чи веб-сайт, які поділяються на основі того, наскільки вони доступні для користувача. У цьому контексті користувач посилається на сутність, яка може бути людиною або цифровою.

Бек-енд – це серверна частина та відноситься до частин комп’ютерної програми або програмного коду, які дозволяють їй працювати та слідувати певним

алгоритмам, і до яких користувач не може отримати доступ напряму. Більшість даних і операційного синтаксису зберігаються та доступні до них у бек-енд частині комп'ютерної системи. Зазвичай код складається з однієї або кількох мов програмування. Бек-енд частину також називають рівнем доступу до даних програмного або апаратного забезпечення та включають будь-які функції, до яких потрібно отримати доступ і навігацію за допомогою цифрових засобів.

Рівень, що дозволяє користувачеві взаємодіяти з серверною частиною є фронт-енд, і він включає все програмне та апаратне забезпечення, яке є частиною інтерфейсу користувача. Люди безпосередньо взаємодіють з різними аспектами інтерфейсу програми, включаючи введені користувачем дані, кнопки, програми, веб-сайти та інші функції.

Системні архітектури розбиваються на клієнтські та серверні компоненти для різних цілей. Найпоширенішим є розробка програмного забезпечення та веб-розробка, щоб розділити проекти з точки зору необхідних навичок. Інтерфейсним аспектом проекту зазвичай займаються професіонали, наприклад веб-дизайнери, тоді як бек-ендом займаються інженери та розробники.

2.2 Аналіз серверної архітектури

Один з головних аспектів аналізу серверної архітектури - це вимоги до сервера. Це включає оцінку потреб у масштабуванні, швидкості, продуктивності та інших характеристиках, які впливають на роботу сервера. Якщо вимоги до сервера відомі заздалегідь, це допомагає вибрати правильну архітектуру та інструменти, які краще відповідають потребам системи.

Другий аспект - аналіз технологій та інструментів, використаних у розробці серверної частини. Це включає оцінку популярності та підтримки технологій, а

також їхню ефективність та можливості. Важливо також враховувати, як технології можуть бути інтегровані одна з одною та як це може вплинути на архітектуру системи.

Третій аспект - складність та масштабованість архітектури. Важливо оцінити, наскільки легко можна додати нові функції та модифікувати існуючу архітектуру, а також наскільки легко можна масштабувати систему в разі збільшення навантаження. Це включає оцінку розширюваності архітектури та здатності до реалізації багаторівневої архітектури.

На відміну від архітектури програми в цілому, серверна архітектура є більш складнішою та вимагає більшого аналізу. У нашому випадку, ми будемо обирати між манолітною та мікросервісною архітектурами. Монолітна архітектура є однією з найбільш поширеніших архітектур. Хоча, сьогодні, мікросервісна архітектура здобуває все більше і більше прихильників, створюючи вагомого конкурента. Далі ми розглянемо та спробуємо порівняти обидві.

2.2.1 Огляд монолітної архітектури при проектуванні серверної частини

Монолітна архітектура передбачає розробку всього програмного продукту як єдиного, недільного цілого, що складається з підсистем і модулів, що взаємодіють між собою за допомогою спільного коду і бази даних.

Потрібно розглянути різні типи архітектур та порівняти їх з вимогами до програмного забезпечення. Наприклад, для великих та складних проектів можна використовувати мікросервісну архітектуру, яка дозволяє розбити додаток на невеликі та незалежні компоненти, що спрощує розробку та розширення. Також її поєднати з клієнт-серверною архітектурою, яка працює на двох рівнях та дозволяє розділити логіку клієнтської та серверної частини.

Дана архітектура є простою та зручною для розробки, тестування та розгортання. Вона дозволяє зосередитися на розробці функціональності програми, не витрачаючи занадто багато часу на проектування та взаємодію між компонентами.

Приклад монолітної архітектури можна побачити на рисунку 2.2.

Рисунок 2.2 – Приклад монолітної архітектури

На рисунку 2.2 можна побачити приклад застосунку з використанням монолітної архітектури. Коли користувач намагається виконати одну з перелічених дій всі запити обробляться в одному місці – “Single instance”. Цей сервіс існує виключно в одному екземплярі.

Даний тип архітектурі використовується для розробки багатьох видів програмного забезпечення, від додатків для мобільних пристроїв до веб-додатків. У монолітній архітектурі всі компоненти додатку взаємодіють безпосередньо між собою, що може зробити систему менш гнучкою та менш масштабованою. Крім того, оновлення однієї складової може вплинути на роботу інших компонентів системи, що може призвести до складнощів у веденні та підтримці додатку.

Однак, цей тип архітектури має свої недоліки. Оскільки всі компоненти додатку розміщуються в одному моноліті, він може стати великим та незручним у

підтримці та оновленні. Крім того, у монолітній архітектурі досить складно здійснювати масштабування додатку. Оскільки всі компоненти взаємодіють між собою безпосередньо, розширення функціональності може бути непростим.

Монолітна архітектура є традиційним підходом до розробки програмного забезпечення, який є простим у реалізації та зрозумілим для розробників. Використання монолітної архітектури може дозволити розробникам швидко створювати та запускати нові функціональність.

Проте, монолітна архітектура може стати обмеженням для подальшого розвитку системи, оскільки всі компоненти системи розміщені в одному моноліті, що робить їх взаємозалежними та ускладнює розгортання та масштабування. Крім того, монолітна архітектура може бути менш гнучкою та менш стійкою до збоїв у порівнянні з мікросервісною архітектурою.

У цілому, монолітна архітектура може бути ефективною у випадках, коли потрібно створити невелику систему з низьким рівнем складності, але у випадках, коли необхідно створити складну систему з високим рівнем гнучкості та масштабованості, мікросервісна архітектура може бути більш ефективним рішенням.

2.2.2 Огляд мікросервісної архітектури при проектуванні серверної частини

Мікросервісна архітектура — це архітектурний стиль, який передбачає розробку програмного забезпечення у вигляді невеликих та незалежних частин, які називаються – мікросервіси. Кожен з них відповідає за невеликий шматок бізнес-логіку та взаємодіє з власною базою даних.

Мікросервіс – це архітектурний стиль, в якому додаток розбивається на набір незалежних компонентів, які взаємодіють між собою через API. Кожен мікросервіс виконує одну конкретну функцію, і є самодостатнім, що означає, що його можна

розглядати як окремий додаток. Кожен мікросервіс може бути написаний різними мовами програмування та мати свою власну базу даних [1].

Крім того, загалом мікросервіси спрощують для команд оновлення програми та прискорюють цикли випуску завдяки безперервній інтеграції та безперервній доставці (CI/CD). Команди можуть експериментувати з кодом і повертатися, якщо щось піде не так.

На рисунку 2.3 зображено приклад мікросервісної архітектури.

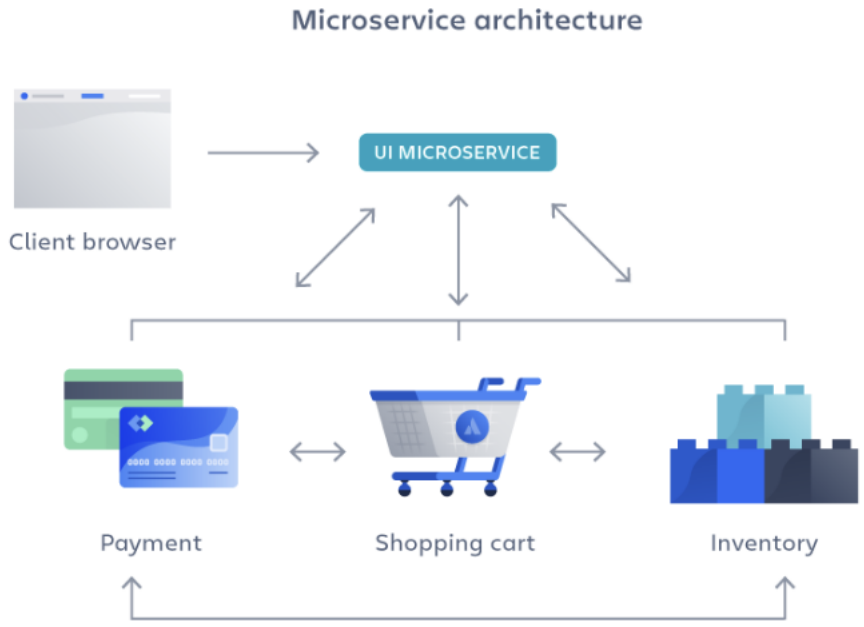

Рисунок 2.3 – Приклад мікросервісної архітектури

З рисунку 2.3 ми можемо побачити відмінності монолітної та мікросервісної архітектур. Дана архітектура передбачає розбиття програми на невеличкі, незалежні частини. Ви можете помітити, що дії, які може виконати користувач, винесені в окремі скрвіси, які можуть взаємодіяти один з одним.

До переваг мікросервісної архітектури можна віднести:

- Гнучкість – сприяйте гнучким способам роботи з невеликими командами, які часто розгортаються.

- Гнучке масштабування – якщо мікросервіс досягає свого навантаження, нові екземпляри цього сервісу можна швидко розгортати в супровідному кластері, щоб зменшити тиск. Тепер ми працюємо з декількома клієнтами та без громадянства, а клієнти розподілені між кількома інстанціями. Тепер ми можемо підтримувати набагато більші розміри екземплярів.
- Безперервне розгортання – тепер ми маємо часті та швидші цикли випусків. Раніше ми випускали оновлення раз на тиждень, а тепер ми можемо робити це приблизно два-три рази на день.
- Незалежне розгортання – оскільки мікросервіси є окремими одиницями, вони дозволяють швидко й легко незалежно розгортати окремі функції.

Недоліки також притаманні даній архітектурі. Коли ми переходимо від невеликої кількості монолітних кодових баз до більшої кількості розподілених систем і служб, що забезпечують наші продукти, виникають складність. Мікросервіси можуть збільшити складність, що призведе до розповсюдження розвитку або швидкого та некерованого зростання. Може бути важко визначити, як різні компоненти пов’язані один з одним, кому належить окремий програмний компонент або як уникнути втручання в залежні компоненти.

Мікросервісна архітектура є потужним інструментом у розробці програмного забезпечення. Проте, використання мікросервісної архітектури вимагає певного рівня досвіду та навичок у розробці програмного забезпечення, оскільки реалізація цієї архітектури може бути складнішою порівняно з традиційною монолітною архітектурою. Крім того, необхідно забезпечити ефективне управління сервісами та контроль за їхньою безпекою та доступністю.

У цілому, мікросервісна архітектура є потужним інструментом для розробки програмного забезпечення, який може допомогти підвищити ефективність розробки та забезпечити більшу гнучкість та стабільність системи.

Багато проектів спочатку починаються як моноліт, а потім розвиваються в архітектуру мікросервісів. Оскільки до моноліту додаються нові функції, багато

розробників, які працюють над єдиною кодовою базою, можуть стати громіздким. Конфлікти коду стають частішими, і ризик оновлень однієї функції вносить помилки в непов'язану функцію. Коли виникають ці небажані моделі, можливо, настав час розглянути можливість переходу на мікросервіси.

На рисунку 1.3 зображено монолітну архітектуру в порівнянні з мікросервісною.

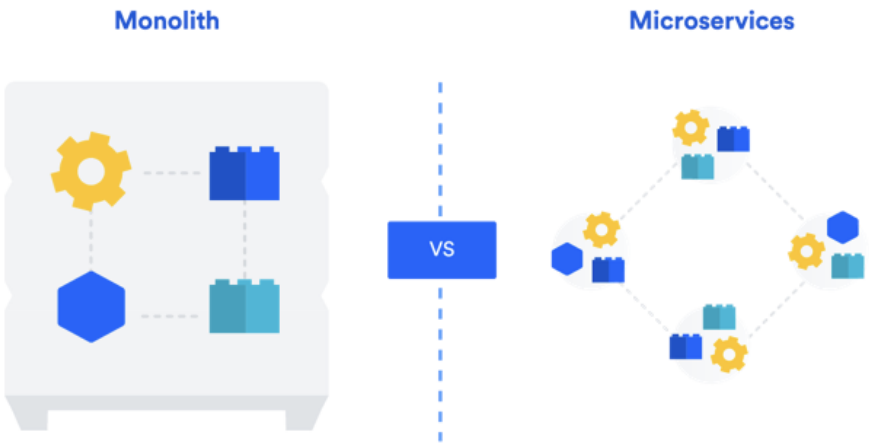

Рисунок 2.4 – Порівняння монолітної та мікросервісної архітектур

Ліворуч, на рисунку 2.4, зображено монолітну архітектуру. Можна помітити, що всі її частини, як і весь функціонал, знаходиться в одній частині програми. Її модулі взаємодіють одна з одною напряму, використовуючи одну кодову базу та мають доступ до ресурсів інших модулів. Праворуч, на тому ж рисунку, зображено мікросірвісну архітектуру. На відміну від попередньої, всі її частини розділені, знаходяться в різних кодових базах та використовують різні ресурси. Самі сервіси взаємодіють один з одним використовуючи API.

Мікросервіси можуть бути не для всіх. Застарілий моноліт може працювати чудово, і його руйнування може не коштувати клопоту. Але оскільки організації ростуть і вимоги до їхніх програм зростають, архітектура мікросервісів може бути корисною.

Обидві архітектури є підходами до розробки програмного забезпечення. Кожен з них має свої переваги та недоліки, тому при виборі архітектури необхідно зважати на конкретні потреби проекту та вимоги до нього.

Мікросервіси розроблені як невеликі самостійні сервіси, які зосереджені на певних бізнес-функціональних можливостях. Кожен мікросервіс можна розробляти, розгортати та масштабувати незалежно, що забезпечує модульну розробку та полегшує обслуговування. Цей модульний підхід сприяє гнучкості та гнучкості, оскільки зміни або оновлення одного мікросервісу не обов’язково впливають на всю систему. Мікросервіси дозволяють горизонтальне масштабування, тобто окремі мікросервіси можна масштабувати незалежно на основі їхніх конкретних моделей використання та вимог. Це забезпечує ефективне використання ресурсів і надає можливість масштабувати лише необхідні компоненти, що призводить до покращення продуктивності та швидкості реагування. Крім того, оскільки мікросервіси слабко пов’язані, вони можуть використовувати різні технології та інфраструктури, які найкраще відповідають їхнім конкретним вимогам.

Монолітна архітектура передбачає, що усі компоненти додатку об'єднуються в один великий блок, який виконує всі функції. У цьому випадку, всі функціональність додатку знаходиться в одному місці і взаємодіє між компонентами здійснюється через локальні виклики. Основна перевага монолітної архітектури полягає у тому, що вона дозволяє розробникам швидко створювати додатки, зменшуючи кількість залежностей між компонентами. Крім того, монолітна архітектура забезпечує простоту у встановленні, тестуванні та розгортанні.

РОЗДІЛ 3 АНАЛІЗ ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

3.1 Огляд використаних технологій при розробці клієнтської частини

Клієнтська частина у вигляді веб-сайту є тим елементом, з яким користувач взаємодіє безпосередньо. Вона відповідає за візуальне представлення даних та надає зручний та легкий доступ до функцій, які надає програмний продукт.

Один з головних критеріїв вибору технологій для клієнтської частини є швидкість та продуктивність. Також важливим критерієм є зручність та простота використання технологій для розробки та підтримки веб-сайту. Для цього використовуються різні мови програмування, фреймворки та бібліотеки, які надають зручний та простий інтерфейс для розробки клієнтської частини веб-сайту.

3.1.1 Використана архітектура при розробці клієнтської частини

Сьогодні мало кого можна здивувати сайтом, який базується на Multiple pages application (MPA) архітектурі, що використовує статичні документи для подання інформації та взаємодії з ним. Про постійне повне перезавантаження сторінки після будь-якої дії годі й говорити. На щастя - або на жаль – іноваційний прогрес не оминув web-розробників. З кожним днем виходить все більше і більше нових технологій, одною з яких є Single Page Application (SPA) архітектура або архітектура односторінкової програми [3].

Односторінкові програми взаємодіють з відвідувачами шляхом автоматичного завантаження поточної сторінки, уникаючи необхідності завантажувати декілька веб-сторінок із сервера. Веб-сайти складаються з одного URL-посилання. Вміст завантажується динамічно, а окремі компоненти інтерфейсу користувача (UI)

оновлюються після автоматично. Взаємодія з користувачем покращується, оскільки користувач може взаємодіяти з поточною сторінкою, поки новий вміст отримується із сервера. Коли відбувається оновлення, частини поточної сторінки оновлюються новим вмістом.

Початковий запит клієнта в SPA завантажує сайт та всі його відповідні ресурси, такі як HTML, CSS і JavaScript. Початковий файл завантаження може бути значним для складних застосунків і призвести до повільнішого часу завантаження. Інтерфейс прикладного програмування отримує нові дані, коли користувач переміщується в SPA. сервер відповідає лише даними у форматі нотації об’єктів JSON. Отримавши ці дані, браузер оновлює вигляд програми, який бачить користувач, не перезавантажуючи сторінку.

Архітектура односторінкових додатків включає технології відтворення на стороні сервера та клієнта. Сайт відображається та представляється користувачеві за допомогою рендеринга на стороні клієнта, рендеринга на стороні сервера або генераторів статичних сайтів.

Це дозволяє уникнути перерв у подорожі користувача, що є життєво важливим для веб-сайтів, особливо в цифровій торгівлі. Скорочуючи час затримки між послідовними сторінками, це робить сайт більш схожим на настільну програму, забезпечуючи більш плавний і комфортний досвід.

Рішення SPA мають багато переваг як на стороні клієнтського досвіду, так і на стороні внутрішнього дизайну рівняння. Покращена продуктивність додатків, узгодженість, скорочений час розробки та нижчі витрати на інфраструктуру не тільки допомагають вам запропонувати більш приємний досвід для ваших користувачів, вони також допомагають командам розробників працювати ефективніше.

Сьогодні односторінкові програми є всюди. Вони є чудовим інструментом для створення неймовірно привабливих та унікальних вражень для користувачів веб-сайту. Односторінкова програма — це веб-сайт або веб-програма, яка динамічно

переписує поточну веб-сторінку новими даними з веб-сервера замість методу за замовчуванням, коли веб-браузер завантажує цілі нові сторінки.

Ви легко впізнаєте деякі популярні приклади односторінкових програм, як-от Gmail, Google Maps, Airbnb, Netflix, Pinterest, Paypal та багато інших. Компанії в усьому Інтернеті використовують SPA для створення плавного, масштабованого досвіду.

Даний вид архітектури виглядає найкращим рішенням для всіх веб сторінок, хоча має свої переваги та недоліки. Ці програми виконують один запит до сервера під час початкового завантаження та зберігають всі отримані дані. Споживачі можуть використовувати отримані дані для роботи в автономному режимі, якщо це необхідно, що робить його більш зручним для користувачів, оскільки вони дають змогу споживати менше ресурсів даних. Крім того, коли клієнт має погане підключення до Інтернету, якщо це дозволяє підключення до локальної мережі.

Використання SPA може підвищити швидкість веб-сайту, оскільки оновлюється лише необхідний вміст, а не вся сторінка. SPA завантажують другорядний файл JSON, а не нову сторінку. Файл JSON забезпечує вищу швидкість завантаження та ефективність. Це забезпечує миттєвий доступ до всіх функцій і функцій сторінки без затримок. Це величезна перевага, враховуючи, що час завантаження веб-сайту може суттєво вплинути на доходи та продажі. Хоче потрібно враховувати, що з цим збільшується кількість запитів на сервер. Якщо сторінка є невеликою, але містить багато елементів, які можуть змінитись під час однієї дії – всі ці елементи зроблять запит на сервер, щоб оновити свій стан.

SPA дозволяють здійснювати плавні переходи, миттєво надаючи всю інформацію на сторінці. Веб-сайт не потребує оновлення, тому його процеси ефективніші, ніж звичайні онлайн-програми.

Крім того, такі ресурси як HTML, CSS і JavaScript, будуть отримані лише один раз протягом життя програми. Обмінюються лише необхідними даними.

Користувачі отримують доступ до сторінок, які миттєво відображаються з усім вмістом одночасно. Це зручніше, оскільки користувачі можуть зручно та безперервно прокручувати. Таке відчуття, ніби користуєшся мобільним або нативним додатком для комп’ютера.

Даний підхід забезпечують позитивний UX з чітким початком, серединою та завершенням. Крім того, користувачі можуть отримати бажаний вміст, не натискаючи кілька посилань, як у MPA. Нижчі показники відмов спостерігаються, оскільки користувачі отримують миттєвий доступ до інформації, на відміну від MPA, де користувачі розчаровуються, оскільки завантаження сторінок займає значну кількість часу. Навігація також швидша, оскільки елементи сторінки використовуються повторно.

Також, дана архітектура дозволяє користувачам виконувати навігацію швидше завдяки кешуванню та зменшенню обсягів даних. Лише необхідні дані передаються туди й назад, а лише відсутні частини оновленого вмісту завантажуються.

Неварто забувати про складову розробки. Розробники можуть повторно використовувати серверний код і відокремлювати SPA від зовнішнього інтерфейсу користувача. Роз’єднана архітектура в SPA розділяє зовнішні дисплеї та серверні служби. Це дозволяє розробникам змінювати світогляд, створювати та експериментувати, не впливаючи на вміст і не турбуючись про технологію серверної частини.

Розробникам, які планують перейти на програми для iOS і Android, слід використовувати SPA, оскільки їх відносно легше конвертувати. Вони можуть використовувати той самий код для переходу від SPA до мобільних додатків. Оскільки весь код надається в одному екземплярі, SPA легко прокручувати, що робить їх ідеальними для мобільних додатків.

Також можна використовувати єдину кодову базу для створення програм, які можуть працювати на будь-якому пристрої, браузері та операційній системі. Це покращує досвід споживачів, оскільки вони можуть використовувати SPA будь-де. Це

також дозволяє розробникам та інженерам DevOps створювати багатофункціональні програми.

Незважаючи на всі переваги односторінкових додатків, при використанні фреймворків SPA виникають деякі недоліки. На щастя, триває робота з подолання цих проблем за допомогою SPA. Нижче наведено проблеми односторінкових додатків.

Поширена думка, що односторінкові програми погано взаємодіють з пошуковими системами. Більшість пошукових систем деякий час не могли сканувати веб-сайти SPA на основі взаємодії Ajax із серверами. В результаті більшість таких сайтів залишилися неіндексованими. Наразі ботів Google навчили використовувати JavaScript замість звичайного HTML для індексування таких веб-сайтів.

3.1.1 Вибір мови програмування для розробки клієнтської частини

Для розробки клієнтської частини буде використовуватись JavaScript. З моменту появи JavaScript він став найпопулярнішою мовою програмування для веб-розробки сьогодні. Ви можете зрозуміти важливість JavaScript як мови веб-розробки з того факту, що згідно з останніми звітами, JavaScript наразі використовується понад 94 відсотками всіх веб-сайтів. Хоча, на мою думку, популярність цією мови програмування в розробці web-застосунків полягає в тому, що аналогів, на жаль, не існує.

JavaScript — це мова програмування на стороні клієнта, яка допомагає веб-розробникам розробляти веб-додатки та створювати динамічні та інтерактивні веб-сторінки, впроваджуючи користувацькі сценарії на стороні клієнта. Розробники також можуть використовувати кросплатформні механізми виконання, як-от Node.js, для написання серверного коду на JavaScript. Розробники також можуть створювати

веб-сторінки, які добре працюють у різних браузерах, платформах і пристроях, поєднуючи JavaScript, HTML5 і CSS3 [5].

Однією з переваг JavaScript є те, що він підтримується багатьма веб-браузерами, такими як Google Chrome, Internet Explorer, Firefox, Safari та Opera тощо. Таким чином, користувачі можуть легко отримати доступ до веб-програм у будь-якому веб-переглядачі на свій вибір. Вони можуть просто ввімкнути мову JavaScript, якщо її вимкнено, і користуватися всіма функціями сайту.

Сьогодні більшість розробників створюють адаптивний веб-дизайн, щоб зробити веб-сайт доступним і добре працювати в різних браузерах і на таких пристроях, як смартфони та комп'ютери. Адаптивний веб-дизайн дозволяє розробникам оптимізувати сайт як для комп'ютерів, так і для мобільних пристроїв за допомогою одного коду. Розробники повинні поєднати CSS3, HTML5 і JavaScript, щоб зробити веб-сторінки адаптивними. Розробники можуть використовувати JavaScript для оптимізації веб-сторінок для мобільних пристроїв.

JavaScript фреймворки і бібліотеки полегшують веб-розробникам на основі JavaScript. Існує багато динамічних фреймворків JS, таких як AngularJS, ReactJS, EmberJS тощо. Використовуючи ці фреймворки, розробники можуть легко, швидко та ефективно розробляти високоінтерактивні та професійно виглядаючі веб-програми. У JavaScript також є багато бібліотек, які ви можете використовувати відповідно до своїх вимог. Наприклад, ви можете розробляти GUI за допомогою таких віджетів, як Bootstrap, jQuery або AngularJS [8].

Як ми вже обговорювали, є кілька причин, чому JavaScript стала найпопулярнішою мовою програмування у світі серед різних типів розробників. JavaScript — це надійна мова програмування, яка допомагає розробникам легко та швидко створювати великі веб-додатки.

JavaScript не тільки дозволяє створювати високоінтерактивну веб-програму, але також допомагає підвищити швидкість, продуктивність, функціональність, зручність використання та функції програми без будь-яких проблем. Зараз розробники та

програмісти все частіше використовують цю мову, щоб зробити свої веб-додатки найкращими та оптимальними для користувачів на різних пристроях, у браузерах та операційних системах. Розробникам також слід ознайомитися з різними бібліотеками, фреймворками та інструментами JavaScript і об’єднати декілька бібліотек і фреймворків, щоб використовувати її відповідно до вимог проектів.

3.1.4 Огляд фреймворку для розробки клієнтської частини

У наш час для будь-якого бізнесу дуже важливо бути онлайн. Для споживачів наявність веб-сайту є його авторитетом. Це також працює навпаки: якщо компанія не має веб-сайту, то в очах споживачів такої компанії не існує. Не дивно, що послуги розробки веб-додатків стають популярними і все більш доступними для широкої аудиторії.

Творці веб-додатків дуже часто зосереджуються лише на візуальному аспекті, але забувають про очікування користувачів. Водночас хороший продукт має поєднувати UX/UI дизайн та продуктивність, адже лише таке поєднання гарантує впізнаваність в очах споживачів. Знання про дизайн продукту необхідні для будь-якого проекту розробки програмного забезпечення!

Іншим важливим фактором є вибір належного стека технологій, який буде підібрано відповідно до вашого проекту. Як компанія, що займається розробкою веб-додатків, ми віримо, що поєднання технологій і пристрасті може мати чудовий ефект. Отже, якщо вам потрібен легкий і швидкий веб-додаток - Angular один із найкращих фреймворків для створення цього [2].

Оскільки ми вирішили розробляти клієнтську частину використовуючи методологію односторінкових застосунків, нам необхідно обрати правильний

фреймворк для побудови нашого додатку. На ринку є багато різноманітних рішень для даного завдання. Кожен з них має низьку переваг та недоліків.

Фреймворки загалом підвищують ефективність і продуктивність веб-розробки, забезпечуючи узгоджену структуру, щоб розробникам не доводилося перебудовувати код з нуля. Фреймворки економлять час і пропонують розробникам безліч додаткових функцій, які можна додати до програмного забезпечення без додаткових зусиль.

Користувальницький інтерфейс (UI) програми Angular створено за допомогою HTML. У порівнянні з JavaScript, HTML є більш простою мовою. Це також декларативна та проста у використанні мова з такими директивами, як ng-app, ng-model тощо.

При використанні серверних можливостей візуалізації Angular можна писати односторінкові веб-додатки (SPA). Це також допомагає швидко завантажувати домашню сторінку та покращує продуктивність веб-сайту на мобільних і малопотужних пристроях [6].

За даними Statista, Angular вже є найкращим фреймворком веб-додатків. Це свідчить про те, що Angular є одним із лідерів у галузі платформ для створення веб-сайтів і популярним вибором для великих підприємств. Це структура, призначена для розробників і компаній, які прагнуть створювати передові програми [7].

3.2 Аналіз використаних технологій при розробці серверної частини

Основною частиною будь якого високонавантаженого застосунку є його серверна частина. На відміну від клієнтської частини, де використовується виключно JavaScript для побудови UI частини, серверну частину можна розробляти на десятках різних мовах програмування. Відповідно до цього важливо дуже уважно та детально підійти до цього питання [4].

Оскільки бекенд відповідає за те, як сайт працює, оновлюється та змінюється, вибір бекенд-технологій, що стоять за програмою, є першим і найважливішим кроком у цій довгій подорожі. Веб-розробка бекенда пов’язана з бізнес-логікою, яка працює на сервері та взаємодіє безпосередньо з базою даних, надаючи на стороні клієнта інтерфейс для доступу до цих даних.

Його також називають розробкою на стороні сервера, оскільки серверна частина виконує все, що відбувається в програмі. Тепер давайте зануримося глибше та подивимося, як працює бекенд і які мови використовуються для розробки бекенда.

Є багато різних мов програмування, які може використовувати розробник серверної частини. Однак певні мови програмування будуть більш вигідними для вашого проекту, ніж інші. Перш ніж вибрати мову для свого внутрішнього коду, ви повинні зрозуміти, що вимагає ваш веб-сайт, програма чи проект програмного забезпечення, як вони використовуватимуться та як ваш вибір мови програмування зрештою вплине на зовнішніх користувачів.

Є кілька основ внутрішньої розробки програм, які повинні бути в центрі вашої уваги при виборі мови програмування, сервера та бази даних. Вибрана вами мова впливатиме на те, наскільки добре функціонує ваш веб-сайт або додаток, наскільки легко їх оновлювати та наскільки легко чи складно керувати вашою базою даних.

Ми збираємося дослідити деякі з найпопулярніших мов програмування. Ми опишемо, що робить кожну з цих мов унікальною, і коли кожна з них стане ідеальним вибором для використання як базової мови. Цей список аж ніяк не є вичерпним. Є багато інших мов програмування, які можна використовувати у внутрішній частині вашої веб-або мобільної програми. Ми просто виділимо найпопулярніші.

Список популярних серверних мов програмування був би надзвичайно неповним без включення Java. Java широко використовується програмістами вже понад 20 років і продовжує залишатися одним із найпопулярніших варіантів для веб-розробки.

Java не тільки надзвичайно стабільна, але й об’єктно-орієнтована, що дає програмістам широкий контроль над мовою. Java працює на будь-якій платформі, вона має можливість запускати декілька завдань одночасно та має неймовірне керування пам’яттю.

Пошук найкращої серверної мови програмування здебільшого залежатиме від ваших потреб. Якщо ви не веб-розробник, вам слід довіряти інтуїції та досвіду веб-розробника, який працює над вашим проектом.

Якщо ви веб-розробник, але не впевнені, яка мова програмування найкраще підходить для вашого останнього проекту, почніть з мови, з якою вам зручніше працювати. Найкраща мова серверного програмування не буде однаковою для всіх. Певні мови працюють краще в певних ситуаціях і програмах, але ви завжди можете працювати з мовою, яка вам найлегша або найкраща для роботи.

3.2.1 Вибір основної мови програмування

Мова програмування для проєкту має базуватися на потребах вашого бізнесу не лише тому, що вона має певний синтаксичний цукор або розкручена. Можливо, ви вважаєте, що, оскільки ви розробник, вибір будь-якої мови має бути виключно вашим вибором. Лише у вас є повна свобода вибору технології, яку ви хочете, але це не працює в IT-організації, і це може мати негативні наслідки для вас.

Як технічний менеджер, ви, по-перше, повинні звернути увагу на всі рухомі частини вашого проекту. Ми повинні знати всі компоненти для кращого перегляду, і це допоможе нам вибрати конкретну мову програмування. Гарне уявлення на початку вашого проекту допомагає вибрати розумну мову програмування, що зменшує час, витрачений на підтримку проекту, розширення проекту та захист проекту в подальшому.

У програмуванні, якщо ви можете писати хороше програмне забезпечення на Java, C#, Python, PHP або будь-яких інших мовах, ви також можете писати погане програмне забезпечення, використовуючи ці мови. Жодна мова не є найкращим вибором для будь-якого програмного забезпечення. Деякі мови та фреймворки краще підходять для проектів, ніж інші. Розглянемо приклад Java. Це не була хороша мова в момент її створення. Просто це було зручніше, ніж у конкурентів. Вибираючи мову, потрібно мислити саме так [9].

Щоб вибрати правильну мову програмування для свого проекту, нам потрібно врахувати різні фактори, такі як продуктивність, тип програми, безпека тощо, які ми обговорювали в розділі запитань, які нам потрібно поставити. Тепер давайте обговоримо ці фактори детально, щоб використовувати відповідну мову для проекту, але пам’ятайте, що завжди є певний компроміс.

Іншим фактором, який потрібно враховувати, є цільова платформа, на якій ви хочете запускати свою програму. Скажімо, у вас є дві мови Java і C. Якщо ви написали програму на C і хочете запустити її в Windows і Linux. У цьому випадку нам потрібні компілятори платформи та два різних виконуваних файли.

У випадку мови Java буде згенеровано байт-код, і щоб запустити його на будь-якій машині, у вас повинна бути встановлена віртуальна машина Java. Подібне відбувається і з веб-сайтами. Він має виглядати та працювати однаково в усіх браузерах.

Яку б технологію та мову ми не обрали, ми отримуємо екосистему бібліотек і підтримку постачальників. Нам потрібно звернути увагу на ремонтопридатність програми, і це є причиною того, що ми завжди повинні дивитися на останню версію мовного або технологічного стеку. Переконайтеся, що все, що ми виберемо, актуальне і залишається актуальним протягом тривалого часу.

Створюючи програму, нам потрібно думати про взаємодію зі своїми клієнтами в довгостроковій перспективі, і в якийсь момент нам доведеться передати свою кодову базу іншій команді. Нам доведеться найняти внутрішніх розробників у

вашому регіоні залежно від технології, над якою ми працюватимемо, і нам доведеться заплатити за це вартість обслуговування.

Кілька особливостей Java можна віднести до простоти, безпеки та портативності мови. У наведеному нижче списку описано дев’ять із цих ключових функцій.

Надійність: Java робить наголос на допомозі користувачам досягти програмування без помилок. Одним із прикладів процесу, який підтримує цю мету, є перевірка під час виконання (RTC). RTC автоматично виявляє та сповіщає користувачів про помилки виконання.

Java створена для легкого вивчення. При належній підготовці та практиці це також може бути легко освоїти. Його поєднання автоматичних і структурно стабільних процесів дозволяє розробникам-початківцям відносно легко створювати програми.

Об’єктно-орієнтований: у мові програмування Java все розглядається як об’єкт. Кожен об'єкт належить до класу і унікально характеризується своєю ідентичністю, станом і поведінкою.

Програми та додатки Java виграють від її багатопотокової природи. Багатопоточний процес дозволяє запускати програми окремо, але виконувати їх одночасно. Крім того, потоки спільно використовують загальну область пам’яті, що зменшує навантаження на центральний процесор (CPU).

Безпечний характер мови програмування Java є однією з її найбільш хвалених особливостей. За замовчуванням Java надає кілька рівнів безпеки, які дозволяють вам як розробнику створювати та запускати середовища кодування без вірусів. Компоненти безпеки включають Java без явного вказівника, розділення локальних і імпортованих пакетів класів завантажувачем класів і компіляцію програм Java у байт-код, щоб назвати декілька.

Ця мова вважається архітектурно нейтральною, оскільки її інтерпретатор байт-коду можна використовувати на будь-якій платформі. Код не містить залежностей і

варіантів, а інструкції з коду Java не виконуються безпосередньо на платформі, на якій він працює.

Java використовує розподілену мовну систему, яка дозволяє безпечно переміщувати та отримувати доступ до коду між різними машинами. Це робить Java повністю сумісною з будь-яким середовищем програмування. Це також дозволяє одночасно підтримувати високі вимоги до пропускної здатності, зменшувати затримку та збільшувати продуктивність.

3.2.2 Вибір фреймворку для пришвидшення розробки

Щоб пришвидшити нашу розробку ми підберемо фреймворт, який у цьому нам допоможе. Мало того, що він дозволить зекономити наш час, а ще й надасть захист від базових хакерських атак.

Фреймворки надають структурований і попередньо визначений набір інструментів, бібліотек і шаблонів, які спрощують процес розробки. Вони часто пропонують такі вбудовані функції, як маршрутизація, інтеграція бази даних, автентифікація та керування сеансами, які можуть значно прискорити розробку та зменшити кількість шаблонного коду. Фреймворки також містять угоди та найкращі практики, які допомагають підтримувати узгодженість у кодовій базі та полегшують розробникам співпрацю над проектом.

Бекенд-фреймворки створені для вирішення складних завдань створення масштабованих і продуктивних програм. Вони часто містять такі функції, як механізми кешування, асинхронна обробка запитів і оптимізовані запити до бази даних. Вибір фреймворка, який відомий своєю масштабованістю та продуктивністю, може гарантувати, що ваша програма зможе обробляти зростаючий трафік і забезпечить хорошу взаємодію з користувачем навіть за високого навантаження.

Популярні фреймворки зазвичай мають великі та активні спільноти, що означає, що доступна обширна документація, навчальні посібники та ресурси. Сильна спільнота також гарантує, що фреймворк добре підтримується, часто оновлюється виправленнями помилок і виправленнями безпеки, а також має багату екосистему сторонніх бібліотек і розширень. Це може заощадити час на розробку та запропонувати рішення для поширених проблем без повторного винаходу колеса.

Майбутнє технічне обслуговування та підтримка. Вибір широко поширеної структури збільшує шанси знайти розробників із досвідом роботи в цій структурі, полегшуючи пошук ресурсів і підтримки, коли це необхідно. Це також зменшує ризик того, що фреймворк стане застарілим або залишеним у майбутньому, забезпечуючи довгострокове обслуговування та підтримку вашої програми.

Зрештою, вибір серверної системи залежить від таких факторів, як конкретні вимоги вашої програми, масштабованість і потреби в продуктивності, розмір і досвід вашої команди розробників, а також довгострокові цілі проекту. Щоб прийняти обґрунтоване рішення, важливо оцінити різні фреймворки, розглянути їхні сильні та слабкі сторони та узгодити їх із вимогами вашого проекту.

Spring Boot — це потужний фреймворк для бекенд-розробки на Java. Він створений на основі Spring Framework, який є широко використовуваним і надійним фреймворком для корпоративної розробки Java. Spring Boot має на меті спростити налаштування та конфігурацію додатків Spring, полегшуючи та пришвидшуючи створення готових до виробництва серверних систем.

Spring Boot дотримується принципу конвенції над конфігурацією, що означає, що вона забезпечує розумні параметри за замовчуванням і автоматичну конфігурацію на основі загальноприйнятих передових практик. Це зменшує потребу в ручному налаштуванні та шаблонному коді, дозволяючи розробникам зосередитися більше на бізнес-логіці, а не на налаштуванні інфраструктури.

Spring Boot містить потужний механізм автоматичного налаштування, який аналізує шлях до класів і автоматично налаштовує різні компоненти та бібліотеки на

основі виявлених залежностей. Це значно спрощує процес конфігурації, усуваючи необхідність ручного підключення та налаштування. За потреби розробники можуть змінити автоматичне налаштування та надати власну конфігурацію.

Spring Boot містить вбудований контейнер сервлетів (Tomcat, Jetty або Undertow), який дозволяє вам запакувати вашу програму як окремий виконуваний файл JAR. Це полегшує розгортання та запуск вашої програми без необхідності використання зовнішніх серверів. Він також забезпечує зручне середовище розробки, оскільки ви можете швидко запустити та перевірити свою програму без встановлення окремого веб-сервера.

Spring Boot поставляється з потужною системою керування залежностями, яка спрощує керування залежностями та версіями. Це гарантує, що різні компоненти та бібліотеки, які використовуються у вашій програмі, сумісні між собою. Spring Boot також надає підібраний набір початкових залежностей, які включають бібліотеки, що часто використовуються для різних функцій, таких як доступ до бази даних, безпека та обмін повідомленнями.

Spring Boot включає декілька готових до виробництва функцій із коробки. Він забезпечує вбудовану підтримку метрик, перевірок працездатності, журналювання та зовнішньої конфігурації. Ці функції спрощують моніторинг і керування програмою у робочому середовищі. Spring Boot також добре інтегрується з іншими проектами Spring та інструментами сторонніх розробників, забезпечуючи комплексну екосистему для розвитку підприємства.

Spring Boot має велике й активне співтовариство розробників, а це означає, що доступні численні ресурси, навчальні посібники та підтримка. Він також має багату екосистему розширень, бібліотек та інтеграцій, які можна легко інтегрувати у вашу програму Spring Boot. Розробка, керована спільнотою, гарантує, що Spring Boot постійно розвивається, з регулярними оновленнями, виправленнями помилок і новими функціями.

Загалом, Spring Boot спрощує та прискорює розробку серверних програм, надаючи добре розроблену та впевнену структуру. Він широко використовується в промисловості для створення надійних і масштабованих систем, що робить його потужним вибором для бекенд-розробки на Java.

Архітектура мікросервісів є популярним підходом для розробки великих і складних програм, розбиваючи їх на менші незалежні служби, які можна розгортати та масштабувати окремо. Spring Boot — це фреймворк, який добре поєднується з мікросервісами завдяки своїй легкості, простоті використання та розгалуженій екосистемі. У поєднанні вони пропонують кілька переваг для розробки програм на основі мікросервісів.

Spring Boot надає повний набір функцій та інструментів, які спрощують розробку мікросервісів. Він включає підтримку створення RESTful API, доступ до даних через Spring Data, безпеку через Spring Security, обмін повідомленнями за допомогою Spring Integration та багато іншого. Конвенційний підхід Spring Boot зменшує зусилля, необхідні для налаштування загальних функцій, дозволяючи розробникам більше зосереджуватися на бізнес-логіці своїх служб.

Зв’язок між мікросервісами є критично важливим аспектом архітектури мікросервісів. Spring Cloud, фреймворк на основі Spring Boot, надає кілька функцій для полегшення виявлення служб і спілкування. Наприклад, він включає Netflix Eureka для реєстрації та виявлення служб, стрічку для балансування навантаження на стороні клієнта та Feign для декларативних клієнтів REST. Ці інструменти спрощують керування зв’язком між службами в екосистемі мікросервісів.

Архітектури мікросервісів дозволяють незалежне масштабування сервісів на основі їхніх індивідуальних потреб. Програми Spring Boot можна легко розгортати та горизонтально масштабувати за допомогою технологій контейнеризації, таких як Docker, і інструментів оркестровки, таких як Kubernetes. Це забезпечує ефективне використання ресурсів і покращує загальну масштабованість і стійкість системи.

Межі мікросервісів допомагають ізолювати несправності, дозволяючи збоям в одній службі мати обмежений вплив на решту системи. Spring Boot разом із такими інструментами, як Spring Cloud Circuit Breaker (наприклад, Netflix Hystrix), надає механізми для обробки та відновлення після збоїв служби. Автоматичні вимикачі та резервні механізми можуть бути реалізовані для елегантної обробки погіршеної продуктивності обслуговування або тимчасової недоступності.

Легкий характер Spring Boot і прості параметри розгортання добре узгоджуються з практиками DevOps і безперервними конвеєрами доставки. Використання технологій контейнеризації та оркестровки разом із автоматизованими процесами збирання та розгортання дозволяє ефективно розробляти, тестувати та розгортати програми на основі мікросервісів.

Варто зазначити, що хоча архітектура мікросервісів із Spring Boot пропонує кілька переваг, вона також створює складності з точки зору управління розподіленою системою, узгодженості даних і тестування. Розробка та впровадження мікросервісів вимагає ретельного розгляду різних факторів, включаючи межі сервісів, протоколи зв’язку, керування даними та стратегії стійкості.

Таким чином, поєднання архітектури мікросервісів із Spring Boot забезпечує потужну та гнучку основу для створення масштабованих, стійких і незалежних служб. Простота та розгалужена екосистема Spring Boot роблять його чудовим вибором для розробки додатків на основі мікросервісів, що дозволяє командам ефективно вирішувати виклики розробки сучасних додатків.

3.3 Аналіз використаних парадигм при розробці серверної частини

У розробці програмного забезпечення існують різні парадигми програмування та підходи, які розробники можуть використовувати для структурування та

проектування свого коду. Парадигми програмування — це різні способи або стилі, в яких може бути організована дана програма або мова програмування. Кожна парадигма складається з певних структур, особливостей і думок про те, як слід вирішувати типові проблеми програмування.

Питання про те, чому існує багато різних парадигм програмування, схоже на те, чому існує багато мов програмування. Певні парадигми краще підходять для певних типів проблем, тому має сенс використовувати різні парадигми для різних типів проектів. Крім того, практики, які складають кожну парадигму, розвивалися з часом. Завдяки прогресу програмного та апаратного забезпечення з’явилися різні підходи, яких раніше не було.

І останнє, я думаю, це людська творчість. Як вид, ми просто любимо створювати речі, вдосконалювати те, що створили інші в минулому, і адаптувати інструменти до наших уподобань або до того, що здається нам більш ефективним.

Існує велика кількість парадигм програмування, які передбачають свої плюси та мінуси, такі як об’єктно орієнтована парадигма, функціональна парадигма, парадигма реактивного програмування, аспектно-орієнтована парадигма та інші. Варто зазначити, що кожна з цих парадигм може існувати в парі з іншими парадигмами. Далі варто підсвітити парадигми, які будуть найбільше використовуватись у нашому проєкті.

3.3.1 Огляд об’єктно орієнтованої парадигми

Основна концепція ООП полягає в тому, щоб розділити проблеми на сутності, які кодуються як об’єкти. Кожна сутність згрупує певний набір інформації (властивостей) і дій (методів), які може виконувати сутність.

ООП активно використовує класи (які є способом створення нових об’єктів, починаючи з плану або шаблону, який встановлює програміст). Об’єкти, створені з класу, називаються екземплярами [10].

Концепції ООП почали з’являтися ще в 60-х роках з мовою програмування під назвою Simula. Незважаючи на те, що свого часу розробники не повністю сприйняли перші досягнення в ООП-мовах, методології продовжували розвиватися.

Перенесімося у 80-ті роки, і редакційна стаття, написана Девідом Робінсоном, була одним із перших знайомств з ООП, оскільки багато розробників не знали про його існування. На сьогоднішній день такі мови, як C++ і Eiffel, стали більш популярними та основними серед програмістів. Визнання продовжувало зростати протягом 90-х років, і з появою Java ООП привернув величезну кількість прихильників.

У 2002 році разом із випуском .NET Framework Microsoft представила нову мову ООП під назвою C#, яку часто називають найпотужнішою мовою програмування.

Цікаво, що через покоління концепція організації вашого коду в значущі об’єкти, які моделюють частини вашої проблеми, продовжує спантеличувати програмістів. Багато людей, які не мають уявлення про те, як працює комп’ютер, вважають думку про об’єктно-орієнтоване програмування цілком природною. Навпаки, багато людей, які мають досвід роботи з комп’ютерами, спочатку думають, що в об’єктно-орієнтованих системах є щось дивне.

Інкапсуляція визначається як загортання даних в єдиний блок. Це механізм, який поєднує код і дані, якими він маніпулює. В інкапсуляції змінні або дані класу приховані від будь-якого іншого класу, і до них можна отримати доступ лише через будь-яку функцію-член свого класу, в якому вони оголошені. Як і в інкапсуляції, дані в класі приховані від інших класів, тому це також відомо як приховування даних.

Розглянемо реальний приклад інкапсуляції: у компанії існують різні розділи, як-от розділ рахунків, відділ фінансів, відділ продажу тощо. Фінансовий відділ

обробляє всі фінансові операції та зберігає записи всіх даних, пов’язаних з фінансами. Подібним чином відділ продажів займається всією діяльністю, пов’язаною з продажами, і веде облік усіх продажів. Зараз може виникнути ситуація, коли чиновнику з фінансового відділу з якихось причин потрібні всі дані про продажі в конкретному місяці. У цьому випадку він не має прямого доступу до даних розділу продажів. Спочатку йому доведеться зв’язатися з іншим співробітником відділу продажів, а потім попросити його надати конкретні дані. Ось що таке інкапсуляція. Тут дані відділу продажів і співробітників, які можуть ними маніпулювати, об’єднані під єдину назву «розділ продажів».

Спадкування є важливою опорою ООП. Здатність класу отримувати властивості та характеристики від іншого класу називається успадкуванням. Коли ми пишемо клас, ми успадковуємо властивості від інших класів. Отже, коли ми створюємо клас, нам не потрібно писати всі властивості та функції знову і знову, оскільки вони можуть бути успадковані від іншого класу, який має ці властивості. Успадкування дозволяє користувачеві повторно використовувати код, коли це можливо, і зменшити його надмірність.

Протягом 70-х і 80-х років процедурно-орієнтовані мови програмування, такі як C і Pascal, широко використовувалися для розробки бізнес-орієнтованих програмних систем. Але в міру того, як програми виконували більш складну бізнес-функціональність і взаємодіяли з іншими системами, почали виявлятися недоліки методології структурного програмування. Через це багато розробників програмного забезпечення звернулися до об'єктно-орієнтованих методологій і мов програмування для вирішення виниклих проблем. Це означає, що команді не потрібно писати той самий код кілька разів. Покращена інтеграція з сучасними операційними системами. Підвищена продуктивність — розробники можуть легко й швидко створювати нові програми за допомогою кількох бібліотек. Поліморфізм дозволяє одній функції адаптуватися до класу, у якому вона розміщена. Легко оновлювати, і програмісти також можуть самостійно впроваджувати системні функції. Це також спрощує пошук

несправностей і співпрацю в розробці. Завдяки використанню інкапсуляції та абстракції складний код прихований, обслуговування програмного забезпечення стає легшим, а Інтернет-протоколи захищені.

Сьогодні більшість мов дозволяють розробникам змішувати парадигми програмування. Це часто тому, що вони будуть використовуватися для різних методів програмування.

Наприклад, візьміть JavaScript – ви можете використовувати його як для ООП, так і для функціонального програмування. Коли ви кодуєте об’єктно-орієнтований JS, вам потрібно ретельно продумати структуру програми та план на початку кодування. Ви можете зробити це, подивившись, як ви зможете розбити необхідні речі на прості, багаторазово використовувані класи, які будуть звичними екземплярами планів об’єктів.

Розробники, які працюють з ООП, зазвичай погоджуються, що загалом його використання дозволяє покращити структуру даних і можливість повторного використання коду. Це економить час у довгостроковій перспективі.

3.3.2 Аналіз функціональної парадигми

Функціональне програмування — це парадигма програмування, в якій ми намагаємося пов’язати все в стилі чистих математичних функцій. Це декларативний тип стилю програмування. Його основна увага зосереджена на тому, «що вирішити», на відміну від імперативного стилю, де основна увага приділяється «як вирішити». Він використовує вирази замість тверджень. Вираз обчислюється для отримання значення, тоді як оператор виконується для призначення змінних. Ці функції мають деякі особливості, які обговорюються нижче [11].

Функціональне програмування базується на лямбда-численні, структурі, розробленій Алонзо Черчем для вивчення обчислень із функціями. Її можна назвати найменшою мовою програмування в світі. Він дає визначення того, що можна обчислити. Все, що можна обчислити за допомогою лямбда-числення, можна обчислити. Він еквівалентний машині Тьюрінга за своєю здатністю обчислювати. Він забезпечує теоретичну основу для опису функцій та їх оцінки. Він є основою майже всіх сучасних функціональних мов програмування.

Переваги функціонального програмування полягають у тому, що чисті функції легше зрозуміти, оскільки вони не змінюють жодних станів і залежать лише від наданих їм вхідних даних. Незалежно від результату, який вони виробляють, вони повертають значення. Сигнатура їх функції надає всю інформацію про них, тобто тип повернення та аргументи. Здатність функціональних мов програмування розглядати функції як значення та передавати їх функціям як параметри робить код більш читабельним і зрозумілим. Тестування та налагодження легше. Оскільки чисті функції приймають лише аргументи та видають вихідні дані, вони не виробляють жодних змін, не приймають вхідні дані або створюють прихований вихід. Вони використовують незмінні значення, тому стає легше перевіряти деякі проблеми в програмах, написаних з використанням чистих функцій. Він використовує відкладене обчислення, яке дозволяє уникнути повторного обчислення, оскільки значення оцінюється та зберігається лише тоді, коли це необхідно.

Різниця між функціональною парадигмою та ООП полягає в тому, що ООП допускає змінний стан, коли об’єкти можуть змінювати свій внутрішній стан з часом. У функціональному програмуванні перевага віддається незмінності, а зміни стану уникаються. Функціональне програмування спирається на концепцію чистих функцій, які не змінюють стан. ООП обертається навколо об’єктів, які інкапсулюють як дані, так і поведінку. У функціональному програмуванні основна увага приділяється функціям, які працюють з даними, і дані часто представлені в незмінних структурах даних. ООП використовує успадкування, коли класи можуть

успадковувати та розширювати властивості та поведінку інших класів. Функціональне програмування наголошує на композиції, де функції поєднуються для створення більш складної поведінки. ООП часто використовує керуючі структури, такі як цикли та умови для потоку керування. Функціональне програмування покладається на рекурсію та функції вищого порядку для потоку керування та уникає змінних циклів. ООП допускає побічні ефекти. Функціональне програмування спрямоване на мінімізацію або усунення побічних ефектів, сприяючи чистим функціям і незмінності.

Обидві парадигми мають свої сильні сторони та підходять для різних сценаріїв. ООП часто використовується при моделюванні складних систем зі змінним станом і фокусом на взаємодії об’єктів. Функціональне програмування добре підходить для завдань, пов’язаних зі складними перетвореннями даних, паралельним і одночасним програмуванням, а також сценаріями, де незмінність і чистота важливі для коректності та зручності обслуговування.

3.3.3 Огляд парадигми реактивного програмування

Зараз, ми підійшли до однієї із найцікавіших парадигм – це парадигма реактивного програмування. Дана парадигма рахується досить новою у порівнянні з вище згаданими парадигмами. Хоча, вона зарекомендувала себе досить перспективною та швидко розвиваючою парадигмою розробки високонавантажених систем. Дана парадигма базується на принципах неблокуючого програмування, що робить її однією з найефективніших для використання [12].

Як ми вже зазначали у вступі, реактивне програмування базується на ідеї асинхронної обробки подій. Асинхронна обробка означає, що обробка події не блокує обробку інших подій.

У перших програмах із графічним інтерфейсом інтерфейс користувача оновлювався лише після певної дії користувача, наприклад натискання кнопки. Серцем програми був так званий цикл подій. Це був нескінченний цикл і відповідав за обробку введених користувачем даних і оновлення інтерфейсу користувача. Але оскільки програма синхронно чекала (тобто блокувала) введення користувача, нічого іншого не могло статися. А синхронне очікування події, яка не настає, означає, що вся програма зависає, а графічний інтерфейс перестає реагувати.

Рішення полягає в тому, щоб відокремити обробку подій від оновлення GUI. Відокремлення здійснюється шляхом введення черги подій і паралельної обробки подій. Цей підхід добре відомий як модель виробника та споживача. Потік, який обробляє дані користувача, розміщує події в черзі, а потік-споживач бере події з голови черги та обробляє їх. Потік споживача також може діяти як виробник, розміщуючи запити на оновлення графічного інтерфейсу користувача в чергу, оброблену потоком оновлення графічного інтерфейсу користувача. Такий ланцюжок виробник/споживач призводить до реактивних потоків.

Сьогодні парадигма реактивного програмування також поширилася на хмарні програми. Хмарні сервіси часто реалізуються як набір мікросервісів. Це невеликі компоненти, які слабко пов’язані та взаємодіють один з одним через асинхронну передачу повідомлень. Парадигма реактивного програмування ідеально підходить для таких мікросервісів.

Для того, щоб побачити всю перспективу реактивного програмування, давайте порівняємо її з потокоблокуючим програмуванням. Приклад ви можете побачити на рисунку 3.1.

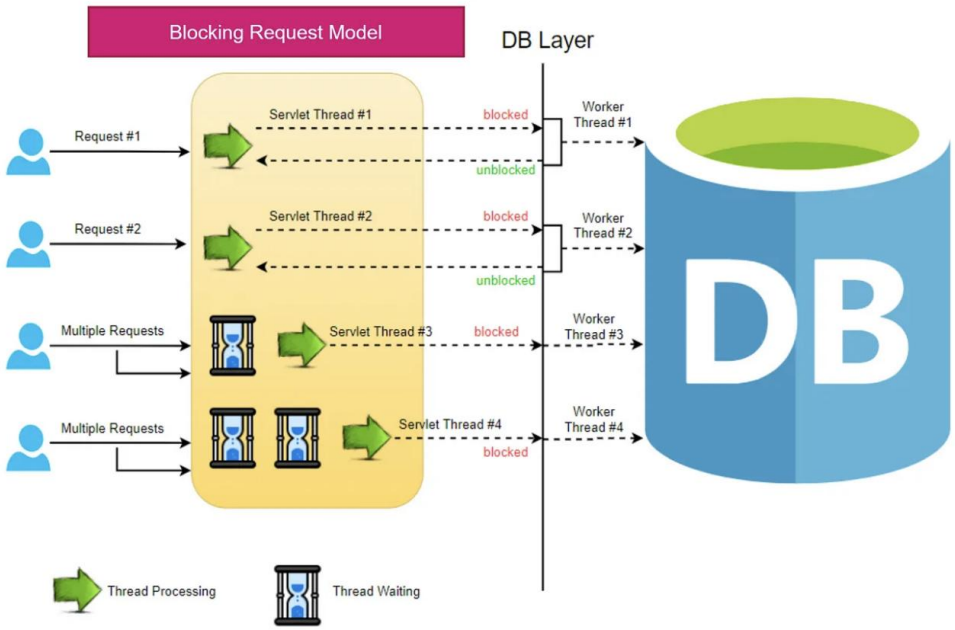

Рисунок 3.1 – Приклад блокуючого програмування

На рисунку 3.1 можна побачити приклад блокуючого програмування. З кожним запитом на сервер створюються новий потік, який очікує на відповідь від бази даних. Саме у цей момент потік простоює без роботи, та не може приймати запити від інших користувачів.

Програмування блокування дотримується моделі послідовного виконання, де кожна операція блокується або чекає, поки вона завершиться, перш ніж перейти до наступної операції. У програмі блокування, коли виконується завдання, воно займає потік виконання, доки завдання не завершиться. Така поведінка блокування може призвести до неефективності ресурсів, оскільки потоки можуть залишатися неактивними під час очікування операцій вводу-виводу, таких як читання з файлу або очікування відповіді мережі.

Програмування блокування зазвичай використовується в моделях синхронного програмування, де потік програми блокується до завершення поточної операції. Цей підхід легко зрозуміти та реалізувати, оскільки виконання програми є детермінованим і слідує передбачуваному порядку. Однак програмування блокування може бути погано масштабованим для обробки великої кількості одночасних завдань

або коли завдання мають різний час відповіді. Це може призвести до вузьких місць продуктивності та поганого використання ресурсів у сценаріях із високим рівнем паралелізму.

Далі давайте розглянемо роботу програми, яка базується на принципах реактивного програмування, де ми використовуємо потоконеблокуючий підхід. Приклад неблокуючого підходу наведено на рисунку 3.2.

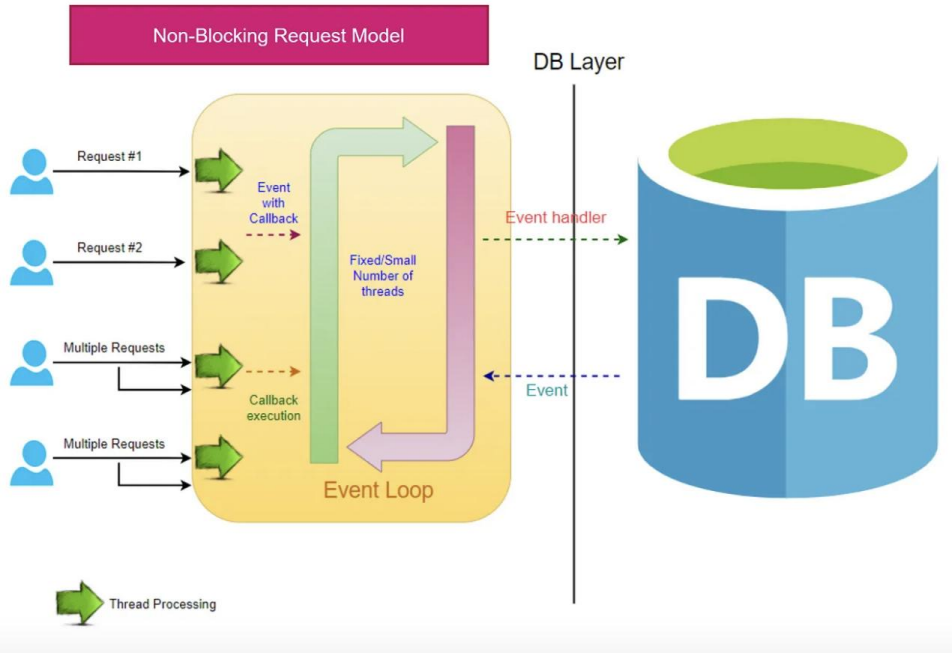

Рисунок 3.2 – Приклад неблокуючого підходу

На рисунку 3.2 наведено приклад потоконеблокуючого застосунку. Коли користувач робить запит на сервер вже не створюється новий потік. Замість того цей запит підбирає існуючий потік, а коли черга догодить до видобутку даних з бази даних, що займає багато часу, потік не очікує завершення видобутку, а бере за той час обробляє запит іншого користувача. Далі, коли вже отримані дані з бази, потік продовжує обробку попереднього запиту.

Реактивне програмування — це парадигма асинхронного програмування, яка зосереджена на обробці потоків даних і подій реактивним і неблокуючим способом.

Він заснований на принципах реактивних систем, які розроблені таким чином, щоб бути чуйними, стійкими, еластичними та керованими повідомленнями.

Реактивне програмування використовує такі конструкції, як реактивні потоки, спостережувані та моделі програмування, керовані подіями, для обробки та реагування на потоки даних і події, коли вони відбуваються. Замість блокування та очікування завершення завдання реактивне програмування використовує неблокуючі операції вводу-виводу, зворотні виклики та архітектури, керовані подіями, щоб максимізувати паралелізм і швидкість реагування [13].

Давайте розглянемо приклад ланцюгу дій, які обробляє програма, яка базується на принципі реактивного програмування. Приклад наведено на рисунку 3.3.

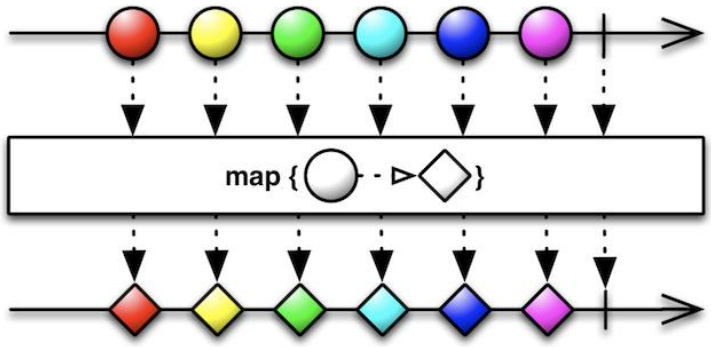

Рисунок 3.3 – Приклад ланцюгу дій на основі реактивного програмування

На рисунку 3.3 наведено приклад ланцюг дій, де кожен кружечок символізує певний об’єкт, а ромбік перетворений об’єкт. Кожен з цих перетворених об’єктів є незалежним і може переходити на наступну стадію обробки, поки готується наступний об’єкт. Програма, яка базується на потокоблокуючому програмуванні очікує перетворення всіх об’єктів та перенасить їх у іншу фазу вже тоді, коли всі готові. В цьому і є ключова особливість даної парадигми.

Реактивне програмування дозволяє розробникам ефективно виконувати велику кількість одночасних завдань. Замість виділення окремого потоку для кожного

завдання, реактивне програмування використовує цикли подій і пули потоків для керування та планування завдань. Коли завдання потребує введення-виведення, наприклад читання з бази даних або виконання запиту HTTP, виконуваний потік звільняється для виконання іншої роботи, що дозволяє системі обробляти більше завдань одночасно.

Реактивний підхід особливо корисний у сценаріях системи, керовані подіями, і високопродуктивні програми, де оперативність і масштабованість є важливими. Реактивне програмування може забезпечити краще використання ресурсів, покращену пропускну здатність системи та зменшену затримку порівняно з блокуючим програмуванням.

Важливо відзначити, що як реактивне програмування, так і блокуюче програмування мають своє місце в розробці програмного забезпечення. Вибір між ними залежить від завдань, що виконуються. Реактивне програмування добре підходить для сценаріїв, які вимагають високого рівня паралелізму, швидкості реагування та масштабованості, тоді як блокуюче програмування може бути більш доречним для простіших послідовних операцій або ситуацій, коли потрібен суворий контроль над потоком програми.

Тестування програми за допомогою реактивних потоків дещо відрізняється від тестування традиційної. Тим не менш, загалом підхід той самий: ми хочемо перевірити, чи програма видає очікуваний вихід для даного введення. У традиційній програмі ми можемо легко перевірити програму, викликавши методи класів. Але в реактивній програмі ми не можемо викликати методи класів безпосередньо. Замість цього ми повинні використовувати API реактивних потоків.

Стратегія тестування для реактивних додатків полягає в тому, щоб перевірити, чи конвеєр потокової обробки видає очікувані результати для заданого потоку вхідних даних. Звичайно, ми повинні протестувати конвеєр у трьох випадках: коли потік завершується нормально, коли потік завершується з помилкою, і коли потік

скасовано. Гарний приклад тестування програми Reactive доступний у Тестуванні реактивних потоків за допомогою StepVerifier і TestPublisher.

Налагодити реактивні потоки важко. У той час як у синхронній та блокуючій системі трасування стека помилок вказує на першопричину проблеми, у потоці асинхронного реактора помилка реєструється в абоненті, але виникає в оператора під час обробки потоку.

3.4 Аналіз використаних баз даних

Невідмінною частиною будь-якого застосунку є база даних. Сьогодні важко уявити програмний продукт, який не використовує базу даних. Всі данні користувачів, починаючи датою входу до в системо, закінчуючи персональними даними, є дуже важливими для програмного продукту і повинні десь надійно зберігатись.

У розробці програмного забезпечення база даних — це структурована сукупність даних, яка організована, керована та доступна комп’ютерною системою. Він призначений для зберігання, отримання та обробки даних таким чином, щоб полегшити ефективне керування даними та пошук для програми або програмної системи.

Вони зберігають дані в структурованому форматі, зазвичай організованому в таблиці (у випадку реляційних баз даних) або документи (у випадку баз даних, орієнтованих на документи). Дані зберігаються постійно, гарантуючи, що вони зберігаються навіть тоді, коли програма не працює активно.

Також забезпечують постійний механізм зберігання даних, гарантуючи, що вони можуть бути надійно збережені та отримані з часом. Це особливо важливо для додатків, які вимагають тривалого зберігання даних і цілісності даних.

У міру того, як програма та її дані зростають, бази даних потрібно масштабувати, щоб обробляти зростаючі обсяги даних і трафік користувачів. Цього можна досягти шляхом вертикального масштабування (збільшення апаратних ресурсів) або горизонтального масштабування (розподіл даних між декількома серверами). Бази даних можуть пропонувати певні механізми масштабування або інтегруватися з масштабованими архітектурами, такими як шардинг або кластеризація.

Бази даних часто забезпечують механізми для резервного копіювання та відновлення даних, що дозволяє розробникам створювати регулярні резервні копії та відновлювати дані у разі збоїв, аварій або пошкодження даних. Це допомагає забезпечити надійність і доступність даних у разі непередбачуваних подій.

Таким чином, бази даних є фундаментальними компонентами розробки програмного забезпечення, які служать репозиторіями для постійного зберігання, пошуку та маніпулювання даними. Вони забезпечують структуроване зберігання, моделювання даних, індексування, безпеку, масштабованість та інші функції, необхідні для створення надійних і ефективних програмних систем. Вибір відповідної технології бази даних і ефективне проектування схеми бази даних є ключовими міркуваннями при розробці програмного забезпечення.

3.4.1 Основна база даних для збереження довгострокових даних

Для нашого проєкту основною базою даних буде PostgreSQL. Вона підтримує стандартний SQL і забезпечує надійну підтримку реляційного моделювання даних, транзакцій, обмежень цілісності та посилальної цілісності. Він дотримується принципів ACID (атомність, узгодженість, ізоляція, міцність), забезпечуючи послідовність і надійність даних.

Вона пропонує широкий спектр функцій, які роблять його придатним для різних випадків використання. Він підтримує розширені типи даних, включаючи масиви, JSON, XML і геопросторові дані. Він надає широкі можливості індексування, включаючи B-дерево, хеш-індекси та GiST (узагальнене дерево пошуку), що забезпечує ефективний пошук даних. PostgreSQL також пропонує підтримку повнотекстового пошуку, складних запитів, збережених процедур і тригерів, надаючи розробникам потужні інструменти для обробки та аналізу даних.

PostgreSQL розроблено для роботи з великими робочими навантаженнями та додатками з високим трафіком. Він підтримує одночасні операції читання та запису, а його архітектура MVCC (Multi-Version Concurrency Control) забезпечує ефективну обробку одночасних транзакцій. PostgreSQL може горизонтально масштабуватися за допомогою таких методів, як шардинг і реплікація, що дозволяє обробляти збільшені обсяги даних і одночасну роботу користувачів.

Дана база даних пропонує ряд функцій для забезпечення цілісності та надійності даних. Він підтримує обмеження (наприклад, первинні ключі, унікальні ключі, зовнішні ключі) для забезпечення дотримання правил цілісності даних. Він забезпечує підтримку транзакцій, дозволяючи розробникам групувати операції бази даних у атомарні одиниці для підтримки узгодженості. PostgreSQL також пропонує відновлення на певний момент часу (PITR) і протоколювання попереднього запису (WAL), надаючи механізми резервного копіювання даних, відновлення та аварійного відновлення.

PostgreSQL має яскраву та активну спільноту з відкритим кодом. Він отримує переваги завдяки регулярним оновленням, виправленням помилок і покращенню продуктивності. Спільнота надає розширену документацію, навчальні посібники та форуми, що полегшує пошук ресурсів і допомоги. PostgreSQL має багату екосистему розширень, інструментів і бібліотек, які ще більше розширюють його можливості та інтегрують його з різними фреймворками та мовами програмування.

Ліцензування є відкритим вихідним кодом і випущений згідно з ліцензією PostgreSQL, що дозволяє вільно використовувати його як для комерційних, так і для некомерційних цілей. Це робить його рентабельним вибором порівняно з пропрієтарними базами даних, для яких може знадобитися дорога плата за ліцензування.

Загалом, багатий набір функцій, надійність, масштабованість, розширюваність і активне співтовариство PostgreSQL роблять його чудовим вибором для широкого діапазону додатків, від малих проектів до великих корпоративних систем. Його дотримання принципів реляційної бази даних у поєднанні з розширеними функціями та оптимізацією продуктивності робить його надійним і гнучким варіантом для зберігання та керування даними.

3.4.2 Опис бази даних для зберігання кешів програми

Оскільки нашою метою є зберігання кешів тоді нам необхідна база даних, яка дозволятиме записувати та читати дані з великою швидкістю. Також важливо не забувати про безпеку та цілісність даних яку буде забезпечувати наша база даних. Саме через ці критерії Redis вибивається у лідери.

Звичайні випадки використання досвідченими користувачами Redis можуть застосовуватися в таких завданнях, як виконання ряду операцій, як-от збільшення хеш-значення, додавання до рядка, обчислення перетину набору, об’єднання та різниці, переміщення елемента до списку або збирання член із найвищим рейтингом у відсортованому наборі. Redis керує такими типами завдань на оптимальному рівні, що може призвести до надзвичайної продуктивності. Це тому, що Redis працює з вбудованим набором даних у пам’яті.

Залежно від варіанту використання можна зберегти те саме, час від часу створюючи дамп на диску або додаючи кожну з команд до журналу. Постійність можна легко вимкнути, якщо вам потрібен мережевий багатофункціональний кеш у пам’яті.

Як і в інших основних базах даних, Redis також підтримує асинхронну реплікацію головний-підлеглий, а також дуже швидку неблокуючу синхронізацію, автоматичне повторне підключення з частковою повторною синхронізацією, коли зв’язок між головною та реплікою розривається, наприклад через проблеми з мережею або тайм-аут. Підключення. Якщо часткова повторна синхронізація має сенс, репліка повторно підключається та намагається продовжити часткову повторну синхронізацію: це означає, що вона намагатиметься просто отримати частину потоку команд, яку вона пропустила під час відключення [14].

Redis пропонує п’ять можливих варіантів даних для значень. Це хеші, списки, набори, рядки та відсортовані набори. Операції, які є унікальними для цих типів даних, наведені та супроводжуються добре задокументованою часовою складністю.

Стійкість даних означає, що дані зберігаються після завершення процесу створення певних даних. Іншими словами, збережені дані мають зберігатися навіть у разі збою сервера. Щоб сховище даних вважалося постійним, воно має записуватись у постійне сховище (тобто енергонезалежне сховище, таке як жорсткий диск або SSD).

Постійності в Redis можна досягти двома різними методами. Спочатку за допомогою знімка, коли набір даних асинхронно передається з пам’яті на диск через регулярні проміжки часу як двійковий дамп за допомогою формату файлу дампа Redis RDB. Альтернативно, шляхом ведення журналу, коли запис кожної операції, яка змінює набір даних, додається до файлу лише для додавання (AOF) у фоновому процесі. Redis може переписати файл лише для додавання у фоновому режимі, щоб уникнути нескінченного зростання журналу. Ведення журналу було представлено у версії 1.1 і зазвичай вважається безпечнішим підходом.

За замовчуванням Redis записує дані у файлову систему принаймні кожні 2 секунди, за потреби доступні більш-менш надійні параметри. У разі повного збою системи на налаштуваннях за замовчуванням буде втрачено лише кілька секунд даних.

Продуктивність у Redis надзвичайно ефективна. Завдяки своїй природі в пам’яті, відданості керівника проекту забезпеченню мінімальної складності, а також моделі програмування на основі подій програма може похвалитися винятковою продуктивністю для операцій читання та запису.

Якщо ви використовуєте Redis Cluster, вертикальне масштабування є надзвичайно ефективним і дуже простим у управлінні. Існує купа команд, якими ви можете скористатися, наприклад перешардинг і ребалансування хеш-слотів. Якщо ви збираєтеся використовувати традиційне налаштування первинної репліки з Redis, використання Sentinel має бути інтегрованим і використовувати правильні клієнти Redis, але з узгодженим хешуванням – це ваш вибір, якщо ви хочете розділити та масштабувати свої основні вузли для розповсюдження запитів на запис.

Коли говорити про варіанти використання Redis, найбільш загальний варіант використання, який трапляється багатьом, це кешування даних і зберігання сеансів (тобто веб-сеансів). Redis має багато варіантів використання, які можна застосувати та бути корисними для будь-яких ситуацій, особливо це стосується швидкості та масштабованості, легко керувати як збільшенням, так і зменшенням масштабу.

РОЗДІЛ 4 АНАЛІЗ ІНСТРУМЕНТІВ ПРИ РОЗРОБЦІ ПРЄКТУ

4.1 Аналіз інструментів для постанови робочого процесу

Зберігання та аналіз вимог є дуже важливим при розробці програмного забезпечення. Завдяки цьому ми можемо організувати пріоритетність та важливість функціоналу, який має бути імплементований. Саме тут нам прийде на допомогу інструменти керування проєктом.

Інструмент керування проектами забезпечує централізовану платформу для зберігання й доступу до всієї інформації, пов’язаної з проектом. Це включає плани проектів, графіки, завдання, документацію, комунікацію та ресурси. Наявність єдиного джерела правди гарантує, що всі члени команди можуть отримати доступ до необхідної інформації, коли вона їм потрібна, сприяючи кращій координації та співпраці.

це дозволяє створювати та підтримувати структуровані плани проекту. Ви можете визначати цілі проекту, розбивати їх на завдання, встановлювати залежності та терміни виконання, призначати обов’язки та відстежувати прогрес. Цей рівень організації допомагає вам бути в курсі графіків і результатів проекту, полегшуючи планування та ефективний розподіл ресурсів.

Крім того, інструменти управління проектами дозволяють створювати та призначати завдання членам команди, встановлювати пріоритети та відстежувати їх прогрес. Ви також можете оцінити та розподілити час для кожного завдання, гарантуючи дотримання термінів і оптимізацію ресурсів. Це допомагає в ефективному управлінні завданнями, зменшуючи ризик пропуску термінів і покращуючи загальну продуктивність.

Саме тут нам приходить на допомогу Jira. Jira — широко використовуваний інструмент управління проектами та відстеження проблем, розроблений Atlassian. Він надає потужну та гнучку платформу для планування, відстеження та керування

своїми проектами та завданнями. Спочатку розроблений для команд розробників програмного забезпечення, Jira розширився для використання в різних галузях і типах проектів.

Це дозволяє командам створювати та відстежувати проблеми, які можуть представляти завдання, помилки, покращення або будь-які інші робочі елементи. Кожну проблему можна призначити членам команди, визначити пріоритети, позначити та пов’язати з пов’язаними проблемами.

Jira надає можливості керування проектами за допомогою таких функцій, як призначення завдань, відстеження прогресу та планування проекту. Це дозволяє командам створювати та керувати планами проекту, визначати робочі процеси, встановлювати терміни та розподіляти ресурси. Він пропонує підтримку гнучких методологій, таких як Scrum і Kanban. Він надає такі функції, як керування невиконаними документами, планування спринтів, відображення історій і гнучкі дошки для візуалізації та керування поточною роботою.

Jira надає функції звітності та аналітики для відстеження прогресу проекту, вимірювання продуктивності команди та отримання інформації про дані проекту. Він пропонує настроювані інформаційні панелі, вбудовані звіти та можливість створювати власні звіти та візуалізації.

Функціональність можна розширити за допомогою різних доповнень і плагінів, доступних на Atlassian Marketplace. Ці розширення пропонують додаткові функції та інтеграції, щоб адаптувати Jira до конкретних потреб команди.

Загалом Jira — це універсальний інструмент керування проектами, який дає змогу командам співпрацювати, відстежувати прогрес і ефективно керувати проектами. Його широко застосовують у різних галузях, включаючи розробку програмного забезпечення, IT-операції, маркетинг, кадрову службу та багато інших, завдяки своїй гнучкості, можливостям налаштування та надійному набору функцій.

4.2 Аналіз інструментів для подальшої розробки

Інструменти для зберігання коду або інструменти CI/CD є неменш важливими для великих та надійних проєктів. Саме тому, ми не можемо згадати про такі інструменти як Bitbucket, TeamCity та Artifactory.

Bitbucket — це веб-сервіс для розміщення репозиторіїв із керування версіями, який переважно використовується для розміщення та керування сховищами Git або Mercurial. Він надає платформу для спільної роботи розробників для роботи над кодом, керування версіями та спільної роботи над проектами. Це гарантує, що команди мають центральне сховище для зберігання вихідного коду та керування ним. Він забезпечує контроль версій, дозволяючи розробникам відстежувати зміни, співпрацювати над кодом і ефективно керувати гілками. Це сприяє узгодженості коду, співпраці та полегшує перегляд коду.

Bitbucket пропонує функції для співпраці та перегляду коду. Це дозволяє розробникам створювати запити на отримання та керувати ними, забезпечуючи структурований і організований спосіб перегляду та обговорення змін коду. Це покращує якість коду, полегшує обмін знаннями та покращує командну співпрацю.

TeamCity — це потужна система безперервної інтеграції та управління збіркою. Він автоматизує процеси збирання, тестування та розгортання, забезпечуючи централізовану платформу для керування конвеєром безперервної інтеграції. Це гарантує, що розробники можуть часто інтегрувати свої зміни коду в спільне сховище. Він автоматично створює та перевіряє зміни коду, надаючи ранні відгуки про можливі проблеми. Це призводить до швидшого виявлення та вирішення помилок, забезпечуючи якість і стабільність коду.

TeamCity спрощує процес керування збіркою, надаючи централізовану платформу для налаштування та керування конвеєрами збірки. Він підтримує різноманітні програми для створення, тестування фреймворків і агентів для

створення, що дозволяє командам ефективно створювати, тестувати та пакувати програми.

Масштабованість і паралельні збірки: TeamCity пропонує функції масштабованості, що дозволяє командам розподіляти робоче навантаження між кількома агентами збірки. Це дозволяє виконувати паралельні збірки, скорочуючи час збирання та покращуючи загальну швидкість розробки та продуктивність. Ця інтеграція спрощує робочий процес розробки, покращує видимість і забезпечує безперебійну співпрацю між різними інструментами та системами.

Artifactory — це менеджер сховища, який служить центральним центром для зберігання бінарних артефактів, таких як бібліотеки, залежності та пакети випусків, і керування ними. Він діє як безпечне та масштабоване рішення для керування артефактами. Він забезпечує надійне та безпечне сховище для зберігання бінарних артефактів. Це гарантує, що артефакти впорядковані, версії та легкодоступні для команди розробників. Це сприяє повторному використанню, зменшує дублювання та покращує загальну якість програмного забезпечення.

Artifactory дозволяє командам ефективно керувати залежностями та контролювати їх. Він підтримує кешування залежностей, проксі-сервер і вирішення, зменшуючи зовнішню мережеву залежність і прискорюючи процеси збирання.

Він допомагає керувати процесом випуску, надаючи репозиторій для зберігання кандидатів на випуск і остаточних збірок. Це гарантує, що артефакти випуску мають правильні версії, контролюються та можуть бути легко розповсюджені в різних середовищах. Artifactory інтегрується з такими популярними інструментами збирання, як Maven, Gradle і Jenkins. Це забезпечує плавне вирішення артефактів і розгортання, сприяючи надійним і ефективним процесам збірки.

Загалом такі системи, як Bitbucket, TeamCity та Artifactory, відіграють вирішальну роль у підтримці сучасних методів розробки програмного забезпечення. Вони сприяють автоматизації, ефективності та якості процесів розробки та випуску програмного забезпечення, сприяють співпраці та цілісності коду.

РОЗДІЛ 5 РОЗРОБКА ОНЛАЙН КІНОТЕАТРУ

5.1 Визначення завдань

Як згадувалось раніше, для зберігання вимог, ми будемо використовувати Jira. Він дозволить зберігати всю необхідну інформацію про набір завдань та їх пріоритизацію. Це дозволить раціонально використати час та якісніше розробляти програмний продукт.

Також буде використано Scrum як фреймворк для побудови робочого процесу. Scrum — це гнучка структура управління проектами, яка допомагає командам структурувати та керувати своєю роботою за допомогою набору цінностей, принципів і практик. Подібно до команди з регбі (звідки вона отримала свою назву), яка тренується до великої гри, Scrum заохочує команди вчитися через досвід, самоорганізовуватися під час роботи над проблемою та розмірковувати про свої перемоги та поразки для постійного вдосконалення.

Було створено новий проєкт та визначено набір завдань і їх опис. Еа рисунку 5.1 можна побачити приклад описаного.

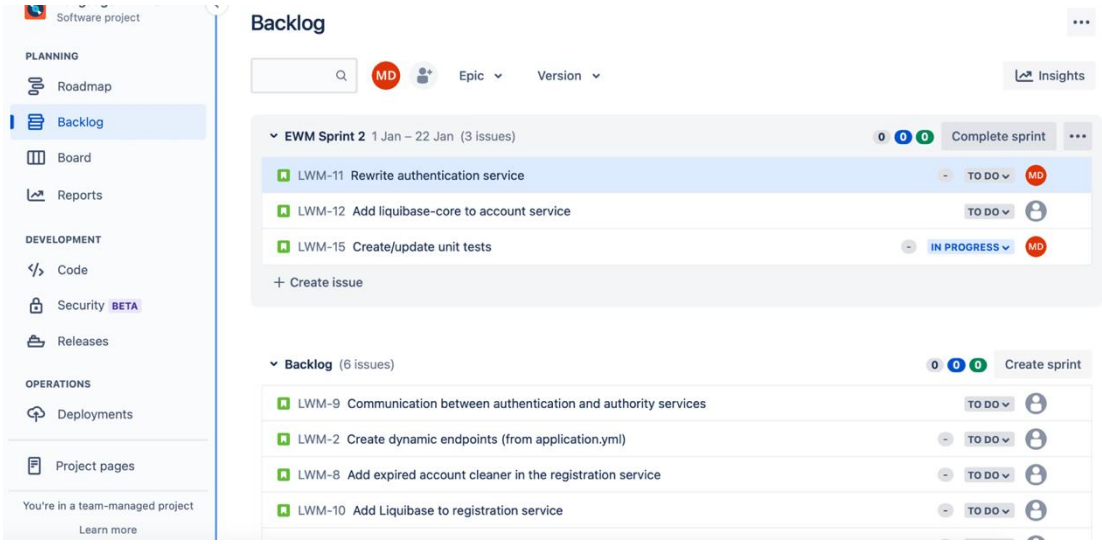

Рисунок 5.1 – Приклад Jira

На рисунку 5.1 зображено приклад завдань в Jira. На рисунку можна побачити текучий спринт з набором завдань та список неактивних завдань. Також можна побачити статус завдання та особу, яка працює з ним. Перейшовши на одне з завдань (рисунок 5.2) можна побачити розширений опис завдання та додатковий функціонал.

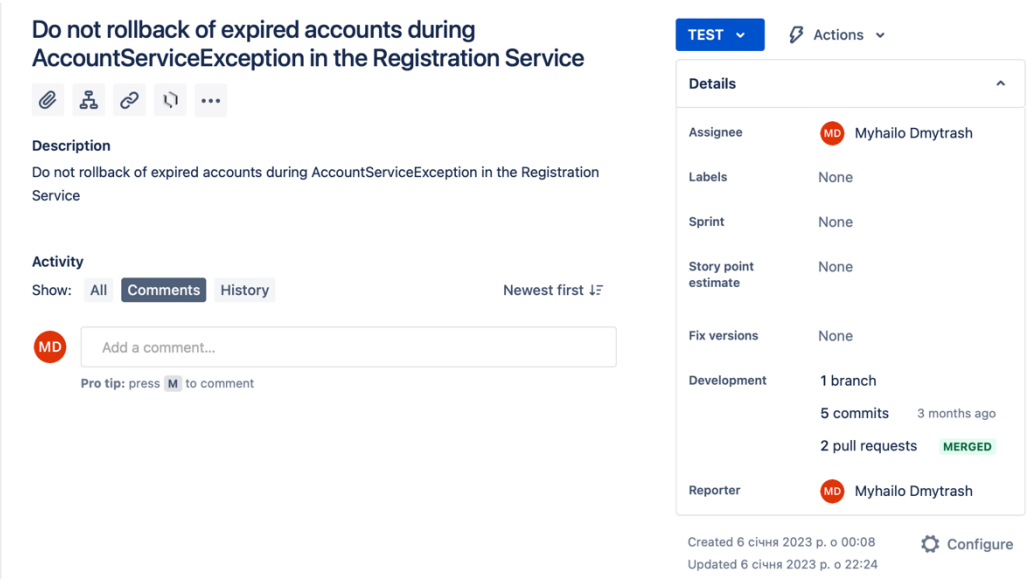

Рисунок 5.2 – Опис одного із завдань

На рисунку 5.2 наведено приклад розгорнутого завдання. На ньому можна побачити додатковий опис та додаткову інформацію. Також реалізований зручніший механізм для роботи з гілками.

5.2 Архітектурний стиль проєкту

Як було вже описано у розділі вище, проєкт буде використовувати клієнт-серверний архітектурний стиль. Це передбачає, що онлайн кінотеатр скрадатиметься з двох частин – система, з якою буде працювати безпосередньо клієнт та сервер, який

буде обробляти запити від клієнтської системи. На рисунку 4.1 наведено діаграму даної архітектури.

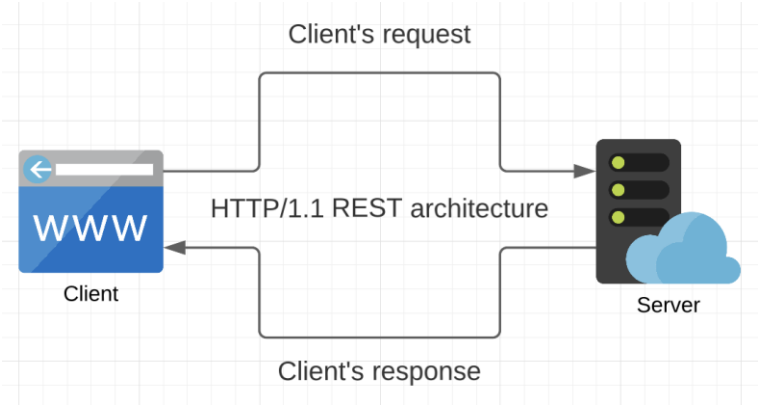

Рисунок 5.1 – Архітектурний стиль проєкту

На рисунку 5.1 наведено приклад архітектури проєкту. На даній рисунку ви множите побачити діаграму взаємодії клієнтської частини із серверною. Обидві системи взаємодіють одна з одною завдяки REST архітектурі, яка базується на HTTP/1.1 протоколі.

REST – це архітектурний стиль для проектування мережевих програм і систем, зокрема веб-сервісів. Він надає набір принципів і обмежень, які дозволяють створювати масштабовані веб-сервіси без збереження стану та сумісні веб-сервіси.

Він не має стану, тобто кожен запит від клієнта до сервера повинен містити всю необхідну інформацію для розуміння та обробки запиту. Сервер не зберігає жодного стану сеансу клієнта між запитами, що спрощує масштабованість і забезпечує кращу відмовостійкість.

REST розділяє клієнтські та серверні компоненти, наголошуючи на чіткому розподілі обов’язків. Клієнт відповідає за користувальницький інтерфейс і взаємодію з користувачем, тоді як сервер відповідає за бізнес-логіку, зберігання даних і управління ресурсами.

Він підтримує механізми кешування для підвищення продуктивності та масштабованості. Кешування дозволяє клієнту або посередникам, таким як проксі-

сервери, зберігати відповіді від сервера для повторного використання для наступних ідентичних запитів.

Ця архітектура передбачає багаторівневу архітектуру, де між клієнтським і серверним компонентами можна додавати проміжні сервери або проксі. Це забезпечує масштабованість, балансування навантаження та підвищену безпеку за рахунок додавання додаткових рівнів абстракції.

Щоб забезпечити взаємодію між клієнтською та серверною частинами, було реалізовано кінцеві точки, основна частина з яких наведена у таблиці 5.1.

Таблиця 5.1 – Кінцеві точки серверної частини

/accounts/register	POST	Реєстрація аккаунту
/accounts/confirm	POST	Активація аккаунту
/registration/resend-code	POST	Перенадсилання коду
/movies/{movie_id}/{language}	GET	Одержання відео
/movies/descriptions/{movie_id}/{language}	GET	Одержання опису відео

У таблиці 5.1 наведено ключові кінцеві точки, якими може скористатись клієнтська частина та їх опис. Це не весь перелік кінцевих точок, адже більшість з них будуть доступні в середині мікросервісної прхітектури.

5.2 Огляд мікросервісів та їх архітектури

Ми підібрались до однієї із найцікавіших частин. Далі, будуть розглянуті самі мікросерміси та зв’язки між ними. Спершу, розглянемо архітектурні патерни, які були використані під час розробки. Їх ви можете побачити на рисунку 5.1.

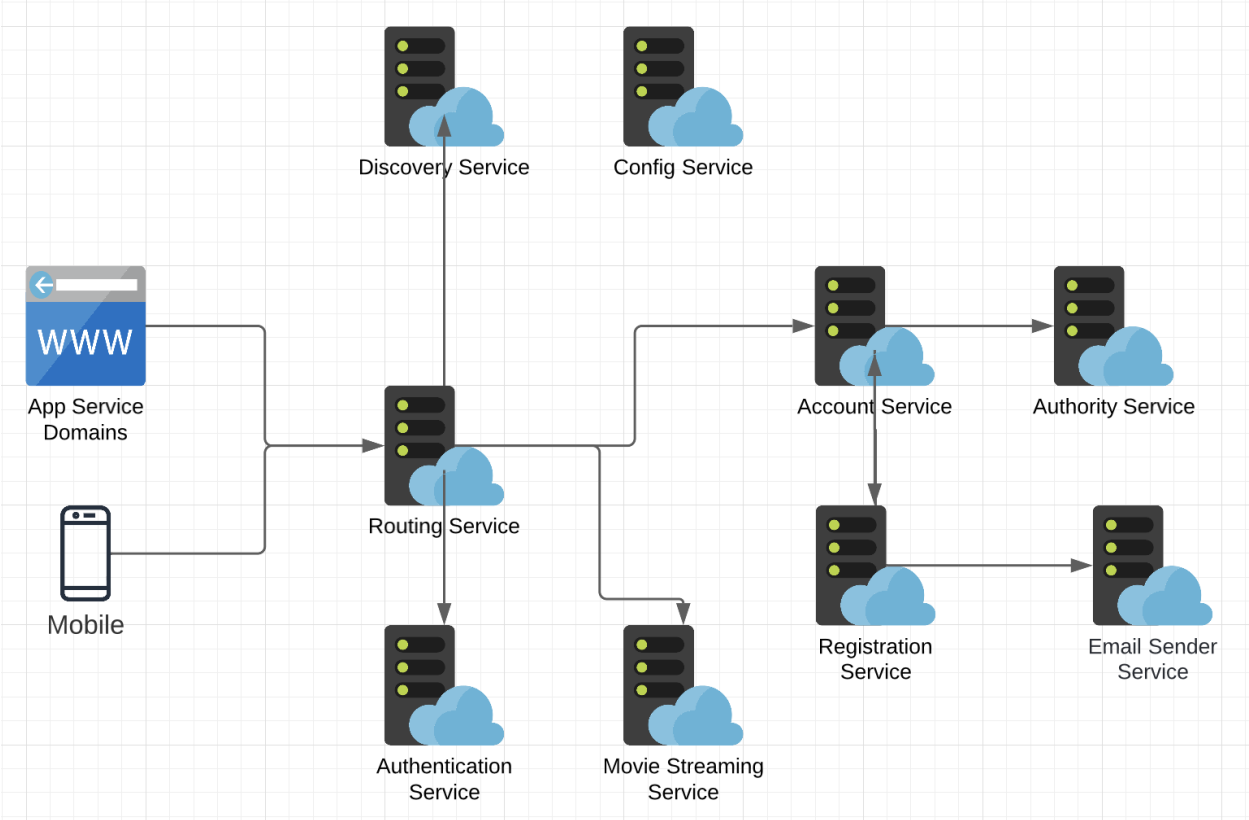

Рисунок 5.1 – Діаграма мікросікрвісім та зв’язків між ними

На рисунку 5.1 можна побачити діаграма мікросікрвісім та зв’язків між ними. Розглянувши даний рисунок, можна зрозуміти, що Routing Service – це сервіс, який делегує всіма запитами на сервер. Він вирішуватиме куди перенаправляти наш запит і чи взагалі кудись його перенаправляти. Також, він буде містити перший рівень захисту.

Щоб Routing Service мав можливість знати, на які IP адреси та порти перенаправляти запити, він це запитує у Discovery Service. Даний сервіс зберігає відомості про кожен мікросервіс, його місцезнаходження та стан.

Наступний неменш важливіший мікросервіс – Configuration Service. Він дозволяє кожному з сервісів дізнатись специфічні налаштування та може їх оновляти у режимі реального часу. Це дозволяє отримати максимальну незалежність так гнучкість системи.

Далі йде перелік сервісів, які відповідають за специфічні можливості самої системи. Account Service відповідає з зберігання даних користувача та все, що з цим пов’язано. За обробку самої можливості збереження та валідацію даних було створено Registration Service. Він перевіряє введені дані, наявність аккаунту та створює заявку на активацію, дані якої передає на Email Sender Service. Цей сервіс дозволяє надсилати код активації на пошту, яку вказав користувач.

Після активації аккаунту, Authority Service створює початковий набір повноважень користувача. Також він дозволяє додавати нові та встановлювати термін їхньої актуальності. Далі, пройшовши авторизацію, дані сесії і інші неохідні дані будуть закріплені за користувачем в Authentication Service.

І на кінець, але не менш важливий – Movie Streaming Service. Цей мікроскрвіс відповідає за передачу медіа контенту користувачеві та попередню обробку медіа файлів. Цей так інші мікросервіцси будуть детальніше розглянуті далі.

5.2.1 Сервіс маршрутизації

В архітектурі мікросервісів служба маршрутизації є компонентом, відповідальним за обробку маршрутизації та пересилання запитів від клієнтів до відповідних мікросервісів. Він діє як центральна точка входу для вхідних запитів і

спрямовує їх до відповідного мікросервісу на основі попередньо визначених правил маршрутизації.

Концепція служби маршрутизації тісно пов’язана з принципами виявлення служби та динамічної маршрутизації в мікросервісах. Оскільки кількість мікросервісів зростає та вони розвиваються незалежно, клієнтам стає складно відстежувати розташування та доступність кожної служби. Служба маршрутизації допомагає вирішити цю проблему, надаючи клієнтам єдину уніфіковану точку входу для взаємодії з екосистемою мікросервісів.

Даний сервіс отримує вхідні запити від клієнтів і перевіряє параметри запиту, заголовки або шаблони URL-адрес, щоб визначити відповідну мікрослужбу для обробки запиту. Він використовує правила маршрутизації та конфігурацію для прийняття рішень щодо маршрутизації.

Служба маршрутизації може застосувати заходи безпеки, такі як автентифікація та авторизація, перед тим, як пересилати запити мікросервісам. Це допомагає централізувати питання безпеки та забезпечує послідовний рівень безпеки в екосистемі мікросервісів. Служба маршрутизації часто включає стратегії балансування навантаження для розподілу запитів між кількома примірниками мікросервісу. Це допомагає досягти кращої масштабованості, продуктивності та відмовостійкості шляхом рівномірного розподілу навантаження між примірниками.

Щоб створити нам даний сервіс ми будемо використовувати Spring Cloud Gateway. Spring Cloud Gateway, — це потужне рішення для маршрутизації та фільтрації, побудоване на основі Spring Boot framework. Він служить шлюзом API, який обробляє вхідні запити від клієнтів і направляє їх до відповідних служб у архітектурі мікросервісів.

Spring Cloud Gateway надає ряд функцій, які полегшують маршрутизацію, фільтрацію, балансування навантаження, безпеку та моніторинг запитів. Він діє як точка входу для клієнтів і пропонує уніфікований інтерфейс для доступу до кількох служб, спрощуючи зв’язок клієнт-сервер.

Це дозволяє визначати правила маршрутизації на основі різних критеріїв, таких як URL-шлях, заголовки та параметри запиту. Він підтримує динамічну конфігурацію маршруту та може направляти запити до різних служб або URL-адрес на основі цих правил.

Використовуючи Spring Boot Gateway, ви можете централізувати та керувати наскрізними проблемами, такими як маршрутизація, фільтрація та безпека, у масштабований та гнучкий спосіб. Це дає змогу створювати стійкі та адаптовані архітектури мікросервісів, одночасно сприяючи слабкому зв’язку та роз’єднанню між клієнтами та службами.

Spring Boot Gateway є частиною екосистеми Spring Cloud, яка надає повний набір інструментів і бібліотек для створення розподілених систем і хмарних програм в екосистемі Java.

Щоб реалізувати даний сервіс, буде використано IntelliJ IDEA Initilazer. Після ініціалізації проєкту, нам необхідно додати залежність Spring Boot Gateway у файл builg.gragle, яка наведена у лістингу 5.1 додатку А.

Далі, нам необхідно налаштувати переадресацію запитів, які надходять з клієнтської частини на відповідний мікросервіс. Шматок коду можна побачити на лістингу 5.2 додатку А.

Також, варто підмітити, що даний мікросервіс використовує парадигму реактивного програмування. Spring Boot Gateway використовує реактивне програмування як фундаментальну частину свого дизайну, щоб забезпечити ефективну та масштабовану обробку одночасних запитів без блокування. Реактивне програмування добре підходить для побудови високочутливих і стійких систем, що відповідає цілям мікросервісів і шлюзів API.

Реактивні моделі програмування, такі як та, що надається бібліотекою Reactor у Spring, уможливлюють неблокуючі операції введення-виведення. Це дозволяє шлюзу ефективно обробляти велику кількість одночасних запитів без блокування потоків,

тим самим максимізуючи використання ресурсів і покращуючи загальну продуктивність.

Ця парадигма полегшує асинхронну модель програмування, керовану подіями. Це дозволяє шлюзу одночасно обробляти кілька одночасних запитів, використовуючи цикл подій і реактивні потоки. Цей підхід гарантує, що ресурси не будуть завантажені під час очікування операцій введення-виведення, що робить систему більш чуйною та масштабованою.

Реактивні потоки підтримують зворотний тиск, що дозволяє шлюзу контролювати швидкість вхідних запитів на основі потужності та доступності низхідних служб. Це гарантує, що шлюз не перевантажує базові служби та допомагає керувати навантаженням і підтримувати стабільність системи.

Використовуючи реактивне програмування, Spring Boot Gateway може ефективно справлятися з високими навантаженнями, підтримувати велику кількість одночасних з’єднань і забезпечувати швидкість реагування та масштабованість. Це дозволяє шлюзу легко інтегруватися з іншими реактивними компонентами в системі та сприяє розробці стійких і високопродуктивних архітектур мікросервісів.

5.2.2 Сервіс виявлення мікросервісів

В архітектурі мікросервісів виявлення служб — це механізм, який забезпечує динамічне виявлення та реєстрацію окремих служб у системі. Це дозволяє службам визначати місцезнаходження та спілкуватися одна з одною без попередньої інформації про розташування чи конфігурації мережі. Виявлення служб відіграє важливу роль у сприянні масштабованості, стійкості та гнучкості додатків на основі мікросервісів.

Реєстр служб діє як централізована база даних або реєстр, де кожен мікросервіс реєструє себе з відповідними метаданими, включаючи своє розташування в мережі (наприклад, IP-адресу та порт) та іншу інформацію, що стосується послуги. Реєстр відстежує доступні служби та їхній поточний статус (наприклад, працездатні, непрацездатні або перебувають на обслуговуванні). Коли мікросервіс запускається або під час виконання, він реєструється в реєстрі сервісів, надаючи необхідну інформацію для ідентифікації та виявлення. Ця реєстрація зазвичай передбачає надсилання сигналів серцебиття або періодичних оновлень до реєстру, щоб вказати доступність служби.

Інші мікросервіси (або клієнти) можуть запитувати реєстр сервісів, щоб виявити та знайти служби, від яких вони залежать. Вони можуть отримати інформацію про доступні служби, такі як їхні мережеві адреси, кінцеві точки або інші метадані, необхідні для зв’язку. Виявлення служб часто інтегрується з методами балансування навантаження, щоб розподілити вхідні запити між кількома примірниками служби. Балансування навантаження забезпечує рівномірний розподіл трафіку запиту та запобігає перевантаженню будь-якого окремого екземпляра.

Service Discovery дозволяє динамічно оновлювати та сповіщати. Коли мікросервіс стає недоступним або його статус змінюється, реєстр може оновити інформацію про сервіс і повідомити зацікавлені сторони, щоб відобразити новий стан. Це дає змогу службам адаптуватися до змін у системі, наприклад збільшення чи зменшення масштабу, не перериваючи загальної функціональності. Виявлення служб може включати перевірку працездатності, щоб перевірити доступність і оперативність служб. Періодично перевіряючи працездатність зареєстрованих служб, механізм виявлення може видаляти або позначати непрацездатні служби як недоступні, гарантуючи, що лише справні служби перераховані та доступні.

Мікросервіси можуть знаходити та спілкуватися один з одним, не знаючи конкретних деталей мережі. Вони покладаються на Service Discovery для динамічного отримання необхідної інформації. Сервіси можна додавати або видаляти

динамічно, не впливаючи на систему в цілому. Щойно додані служби реєструються автоматично, а видалені служби видаляються з реєстру, що дозволяє легко масштабувати та підтримувати. Якщо служба дає збій або стає недоступною, служба виявлення служб може усунути збої, позначивши службу як недоступну та перенаправляючи запити до інших справних екземплярів. Це підвищує стійкість системи, уникаючи окремих точок відмови. Послуги можуть розвиватися незалежно, не впливаючи на інші служби. Їх можна додавати, оновлювати або замінювати з мінімальним впливом на загальну систему, якщо вони зареєстровані та відповідають механізму виявлення служб.

Механізми виявлення служб можуть відрізнятися залежно від реалізації та стеку технологій. Деякі популярні інструменти та фреймворки для виявлення служб включають Netflix Eureka, Consul тощо, а також вбудовану функцію виявлення служб Kubernetes. Ці інструменти надають такі функції, як автоматична реєстрація, динамічні оновлення, балансування навантаження та перевірка працездатності, щоб спростити зв’язок служби в архітектурі мікросервісів.

Служба Spring Eureka, також відома як Netflix Eureka, — це реалізація сервера виявлення послуг, що надається платформою Spring Cloud. Він призначений для полегшення реєстрації, виявлення та балансування навантаження мікросервісів у розподіленій системі.

Сервіс Eureka діє як центральний реєстр, де мікросервіси можуть реєструватися та отримувати інформацію про інші сервіси. Це забезпечує динамічне виявлення служб, дозволяючи службам знаходити та спілкуватися одна з одною без попередньої інформації про розташування чи конфігурації мережі.

Щоб реалізувати даний сервіс, ми знову скористаємось IntelliJ IDEA Initilazer. Також, наведено приклад налаштування класу, щоб реалізувати даний механізм.

5.2.3 Сервіс налаштувань

В архітектурі мікросервісів служба конфігурації — це спеціалізований компонент, який відповідає за керування та розповсюдження властивостей або параметрів конфігурації іншим мікросервісам у системі. Він централізує процес керування конфігурацією, дозволяючи мікросервісам динамічно отримувати свою конфігурацію під час виконання без необхідності повторного розгортання.

Служба конфігурації діє як центральне сховище або сервер, де зберігаються властивості конфігурації. Він містить пари ключ-значення або структуровані конфігураційні дані, до яких можуть отримати доступ мікросервіси. За допомогою зовнішньої конфігурації мікросервіси можна розробляти без жорсткого кодування їхніх властивостей конфігурації. Натомість вони отримують необхідну конфігурацію зі служби конфігурації динамічно під час виконання. Мікросервіси можуть отримати свою конфігурацію зі служби конфігурації під час запуску або під час роботи. Це дозволяє динамічно оновлювати конфігурацію без необхідності повторного розгортання мікросервісів.

Служба конфігурації може отримувати властивості конфігурації з різних джерел, таких як файли, бази даних, змінні середовища або зовнішні постачальники конфігурації. Він підтримує гнучкість у виборі різних джерел конфігурації на основі потреб мікросервісів. Служба конфігурації може підтримувати керування версіями та історію властивостей конфігурації, уможливлюючи відкат або аудит змін. Це допомагає в управлінні змінами конфігурації та збереженні запису історичних станів конфігурації.

Ця служба може надавати механізми захисту конфіденційних властивостей конфігурації, наприклад шифрування паролів або ключів API. Це гарантує, що конфіденційна інформація не буде розкрита в сховищі конфігурацій або під час передачі. Крім того, він може підтримувати механізми сповіщення мікросервісів про зміни конфігурації та запуску оновлення конфігурації під час виконання. Це гарантує, що мікросервіси можуть адаптуватися до оновлень конфігурації без перерв.

Параметрами конфігурації можна керувати та оновлювати їх централізовано, що полегшує підтримку та зміну конфігурацій у кількох мікросервісах. Мікросервіси можуть бути більш гнучкими та гнучкими, оскільки зміни конфігурації можуть бути застосовані без необхідності повторного розгортання. Це дозволяє швидко регулювати поведінку системи та налаштування. Проблеми конфігурації відокремлені від самих мікросервісів, сприяючи принципу єдиної відповідальності та модульності.

Служба конфігурації може керувати конфігурацією великої кількості мікросервісів, забезпечуючи узгодженість і зменшуючи накладні витрати на керування конфігурацією для кожної окремої служби. Зміни конфігурації можна контролювати й перевіряти, забезпечуючи видимість того, хто та коли вніс зміни. Це допомагає у вирішенні проблем і збереженні історії конфігурації.

Популярні інструменти для впровадження служб конфігурації в архітектурі мікросервісів включають Spring Cloud Config, HashiCorp Consul, Apache ZooKeeper тощо. Ці інструменти надають функції для керування, розповсюдження та захисту конфігурацій між мікросервісами.

У проєкті буде використовуватись імплементація від Spring Cloud Config. Spring Cloud Config — це модуль у рамках Spring Cloud, який забезпечує підтримку зовнішнього керування конфігурацією в архітектурах мікросервісів. Це дозволяє централізувати та керувати властивостями конфігурації ваших мікросервісів у розподіленому середовищі.

Він забезпечує центральне сховище конфігурацій, де ви можете зберігати властивості конфігурації для своїх мікросервісів. Це сховище може бути репозиторієм Git із керуванням версіями, файловою системою чи іншим підтримуваним серверним модулем. Конфігураційний сервер є компонентом, наданим Spring Cloud Config. Він діє як центральна точка для надання властивостей конфігурації клієнтським мікросервісам. Він отримує властивості конфігурації з центрального сховища та надає їх запитуючим мікросервісам.

За допомогою Spring Cloud Config ви можете зовнішні конфігурації своїх мікросервісів, тобто властивості конфігурації зберігаються окремо від коду програми. Це дозволяє динамічно оновлювати конфігурацію, не вимагаючи повторного розгортання мікросервісів. Він підтримує концепцію профілів, що дозволяє визначати різні набори властивостей конфігурації для різних середовищ або профілів додатків. Ви можете мати конфігурації, специфічні для середовищ розробки, тестування, постановки або виробництва, а сервер конфігурації може обслуговувати відповідну конфігурацію на основі активного профілю.

Spring Cloud Config легко інтегрується з Spring Boot, який є популярним фреймворком для створення мікросервісів. Програми Spring Boot можуть легко використовувати властивості конфігурації з сервера конфігурації за допомогою клієнтських бібліотек Spring Cloud Config.

Використовуючи Spring Cloud Config, ви можете ефективно керувати та розподіляти властивості конфігурації між вашими мікросервісами. Він сприяє розділенню питань конфігурації, дозволяє змінювати конфігурацію під час виконання та забезпечує централізований і безпечний підхід до керування конфігурацією в розподіленому середовищі.

Щоб реалізувати цей сервіс, нам потрібно створити новий проєкт та додати необхідні залежності до файлу build.gradle (лістинг 5.5 додатку Б). Далі, щоб задати необхідні конфігурації для кожного з мікросервісів необхідно внести зміни до файлу application.yml, який відображений в лістингу 5.6 додатку Б та, при необхідності,

створити нові yml файли, які будуть містити специфічні конфігурації для кожного з мікросервісів.

Далі, щоб активувати Spring Cloud Config нам необхідно додати аннотацію “EnableConfigServer” над класом, який забезпечує запуск проєкту. Приклад коду наведено в лістингу 5.7 додатку Б.

5.2.4 Сервіс реєстрації

Наступним сервісом, з яким у першу чергу взаємодіє користувач – це сервіс реєстрації користувачів. Даний сервіс відповідає за перевірку даних введених користувачем та перевірку їх унікальності. Якщо всі критерії успішної реєстрації виконані – даний сервіс передає данні користувача на сервіс по обробці даних аккаунтів.

Давайте детальніше розглянемо роботу даного мікросервісу на прикладі діаграми послідовності, яка зображена на рисунку 5.2.

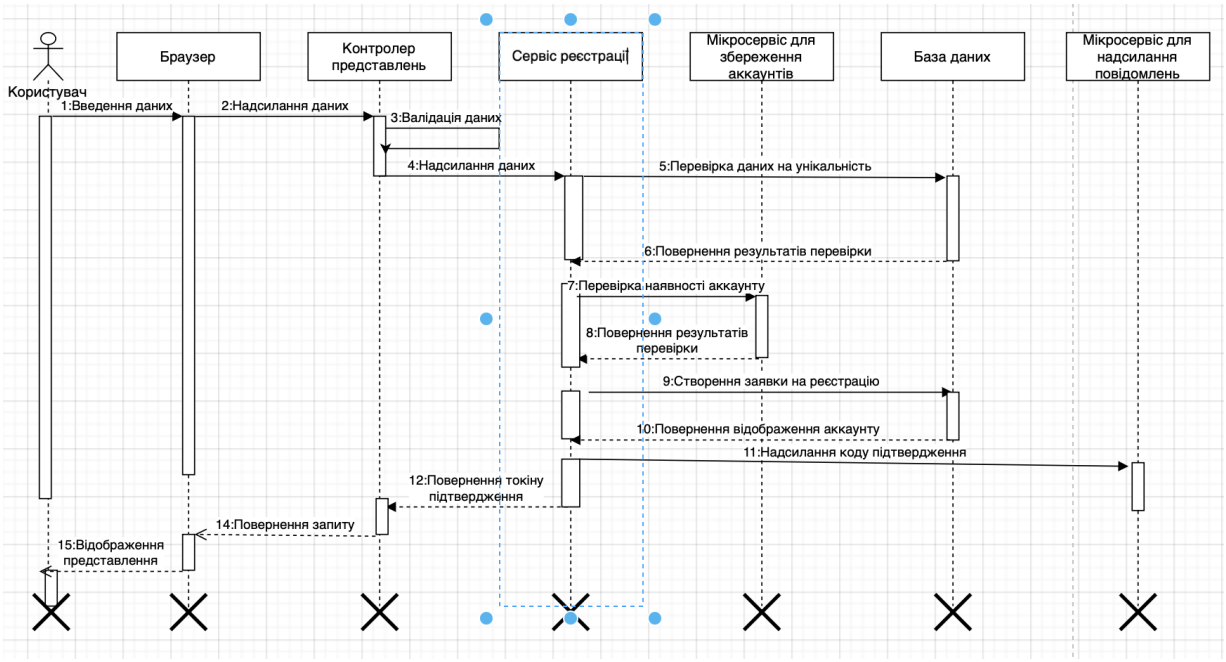

Рисунок 5.2 – Зображення діаграми послідовності дій «Авторизація»

На рисунку 5.2 дія «Авторизація». На ній зображено як користувач першим кроком надсилає дані через браузер на контролер представлення, де дані валідуються. Далі, ці дані попадають у сервіс реєстрації та перевіряються на унікальність та на наявність існуючого аккаунту у базі даних. Після успішної перевірки даних заявка на реєстрацію зберігається у базі даних і сервіс авторизації передає пошту та код підтвердження мікросервісу по розсилці повідомлень. Також, даний сервіс повертає токін авторизації через контролер назад до користувача. Відповідно дані відображаються користувачеві.

Далі розглянемо діаграму послідовності дії “Підтвердження електронної адреси”, яка наведена на рисунку 5.3.

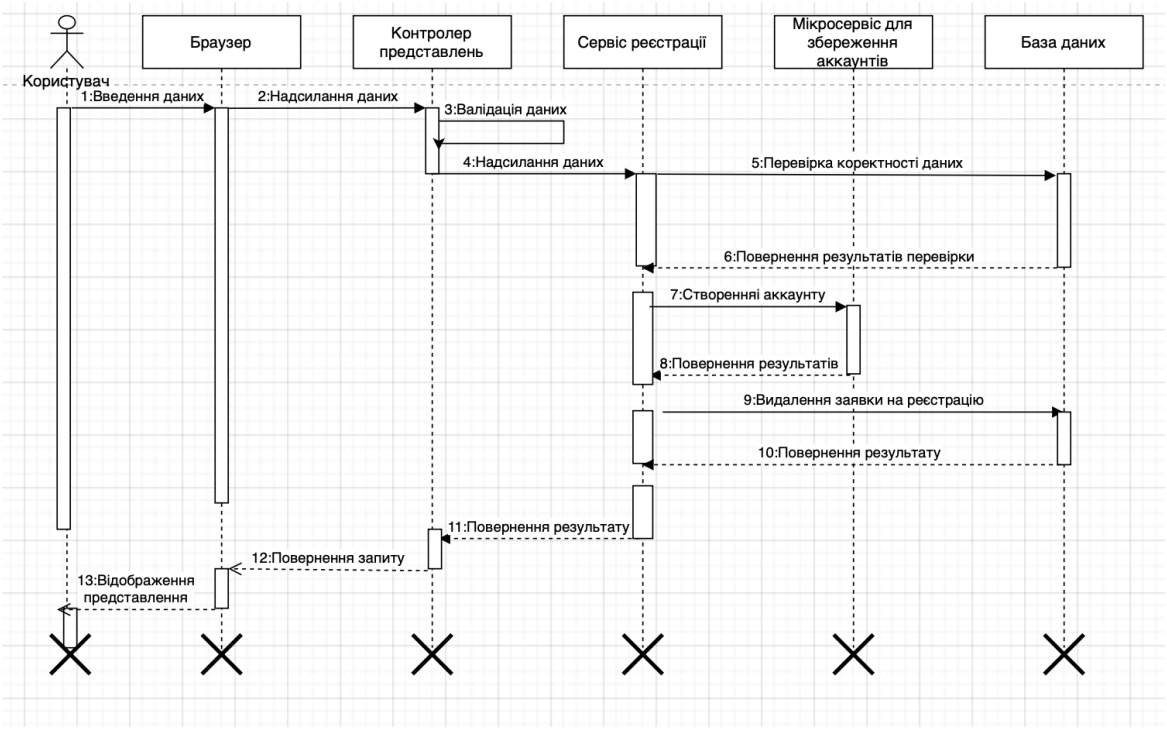

Рисунок 5.3 – Діаграма послідовності дії «Підтвердження пошти»

На рисунку 5.3 зображена дію «Підтвердження пошти». На ній зображено як користувач надсилає дані за допомогою браузер та дані попадають на контролер представлення, де вони валідуються. Далі, ці дані попадають у сервіс реєстрації та перевіряються на коректність. Після успішної перевірки даних заявка на реєстрацію зберігається надсилаються на сервіс по роботі з аккаунтами та видаляться із базі даних.

Далі розглянемо діаграму класів даного мікросрвісу. Вона зображена на рисунку 5.4.

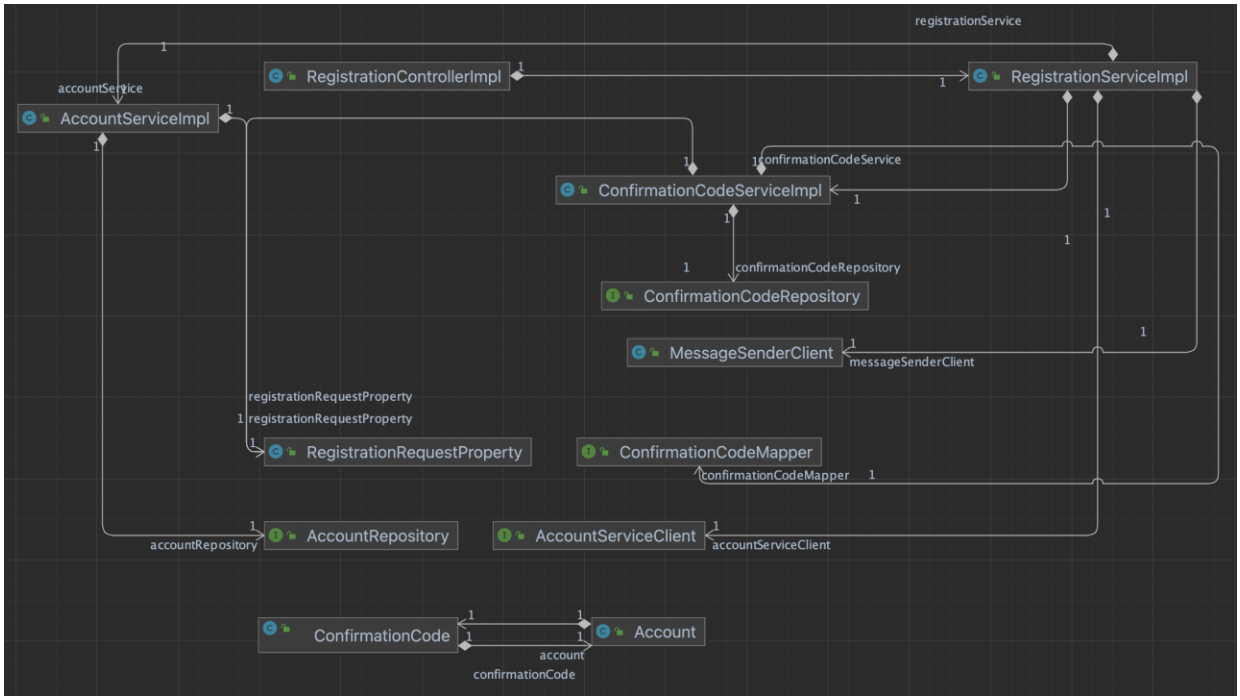

Рисунок 5.4 – Діаграма класів мікросервісу реєстрації

На рисунку 5.4 зображено діаграму класів мікросервісу реєстрації. На даній діаграмі можна побачити клас RegistrationControllerImpl (зображений в лістингу 5.8 додатку В) який містить сервіс по обробці заявок на реєстрацію – RegistrationServiceImpl, зображений в лістингу 5.9 додатку В. У свою чергу цей сервіс містить інтерфейс, який реалізує клас для взаємодії зі мікросервісом по обробці аккаунтів. Також він містить сервіс відповідальний за роботу з кодами підтвердження.

У свою чергу сервіс по роботі з аккаунтами містить репозиторій, який відповідає за взаємодію з базою даних. Також він містить клас по обробці параметрів.

Наступний – ConfirmationCodeServiceImpl, який відповідає за обробку кодів підтвердження заявок на реєстрацію та відображений в лістингу 5.10 додатку В. Він містить репозиторій, який відповідає за роботу з базою даних, клас який відповідає за налаштування. Та інтерфейс, що зображено в лістингу 5.11 додатку В, який реалізовуватиме клас, що перетворюватиме один тип об’єктів у інший.

Також можна побачити сутність Account, що зображена в лістингу 5.12 лістингу В, який відповідає за відображення аккаунту. Додатково клас ConfirmationCode, який зображено в лістингу 5.13 додатку В, що відповідає за відображення коду підтвердження.

5.2.6 Сервіс керування аккаунтами

Наступний сервіс відповідатиме за збереження та обробку даних користувача. Він забезпечує дані в цілісності та безпеці. Далі, буде розглянуто діаграму послідовності для даного мікросервісу, що зображено на рисунку 5.5.

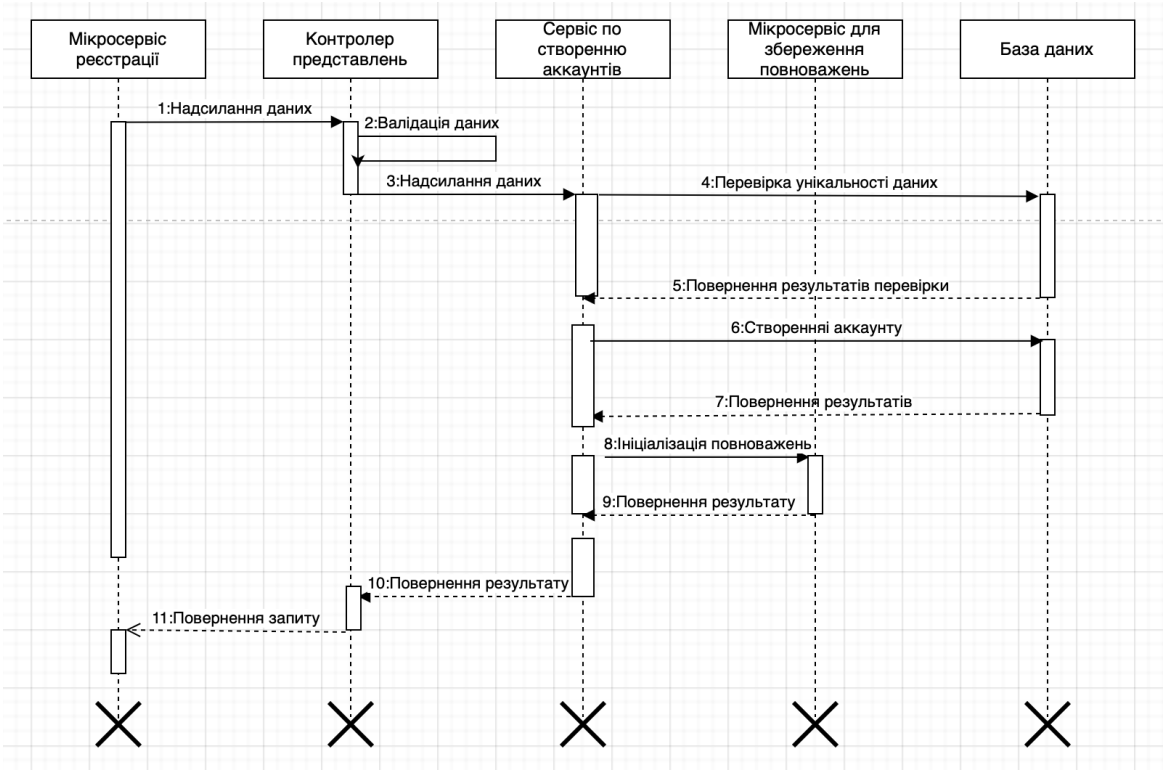

Рисунок 5.5 – Діаграма послідовності дії «Створення аккаунту»

На рисунку 5.5 зображено діаграму послідовності дії «Створення аккаунту». Мікросервіс реєстрації передає дані на мікросервіс по роботі з аккаунтами. Дані валідуються, передаються на сервіс по створенню аккаунтів, де перевіряються на унікальність в базі даних, а далі зберігаються. Наступним кроком буде ініціалізація повноважень для даного клієнту. В кінці цієї дії результат повертається на мікросірвіс реєстрації.

Далі розглянемо діаграму класів для даного мікросервісу, яка зображена на рисунку 5.6.

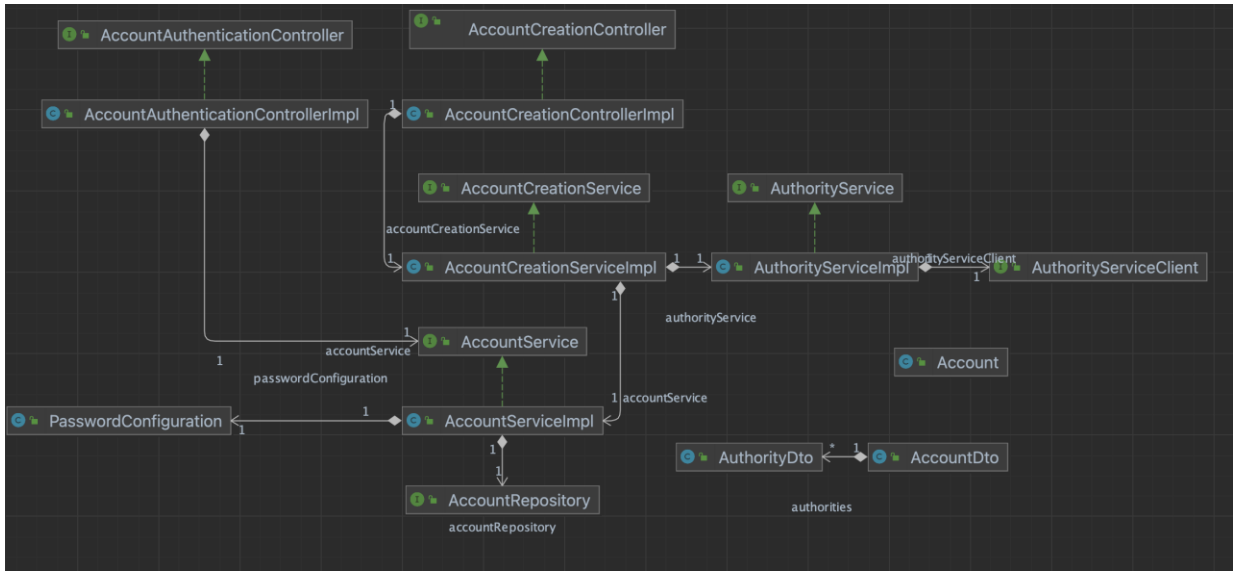

Рисунок 5.6 – Діаграма класів мікросервісу по роботі з аккаунтами

На рисунку 5.6 зображено діаграму класів даного мікросервісу. На діаграмі можна побачити клас AccountCreationControllerImpl, який наведено в лістингу 5.14 додатку Г, що реалізовує інтерфейс AccountCreationController та містить AccountCreationServiceImpl, який наведений в лістингу 5.15 додатку Г. У свою чергу клас AccountCreationServiceImpl містить AuthorityServiceImpl (лістинг 5.16 додптку Г) та AccountServiceImpl, який наведено в лістингу 5.17 додатку Г.

Також, можна побачити сутності такі, як клас Account, що наведений у лістингу 5.18 додатку Г та AccountDto, що наведений в лістингу 5.19 додатку Г. Ці класи представлення необхідні для отримання та передачі даних між мікросервісами.

5.2.9 Сервіс обробки медіа контенту

Наступним мікросервісом буде мікросервіс по обробці медіа контенту. Це мікросервіс дозволятиме отримувати відео контент користувачеві. Данні будуть передачатись частинами, щоб зменшити навантаження на користувача. Даний сервіс буде використовувати парадигму реактивного програмування. Далі, буде розглянуто діаграму послідовності запиту відео контенту користувачем. Її можна побачити на рисунку 5.7.

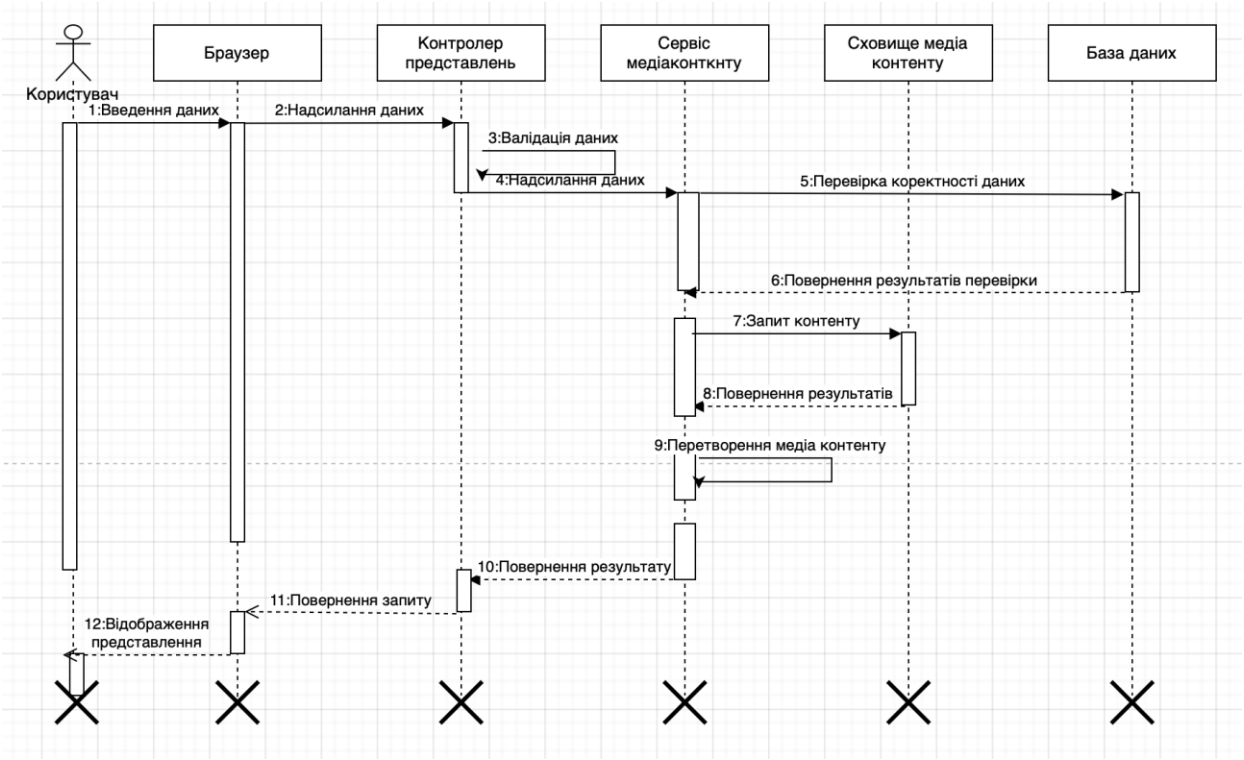

Рисунок 5.7 – Діаграма послідовності дії «Запит медіа контенту»

На рисунку 5.7 наведено діаграму послідовності дії «Запит медіа контенту». На ній можна побачити як користувач робить запит запит, щоб отримати медіа контент. Дані проходять валідацію, після чого сервіс медіа контенту перевіряє наявність даного медіа контенту в базі даних після чого робить запит на сервіс, де зберігається даний медіа контент. Після чого дані перетворюються відповідно до вимог користувача та передаються йому назад.

Дані буде розглянута діаграма класів даного мікросервісу на рисунку 5.8.

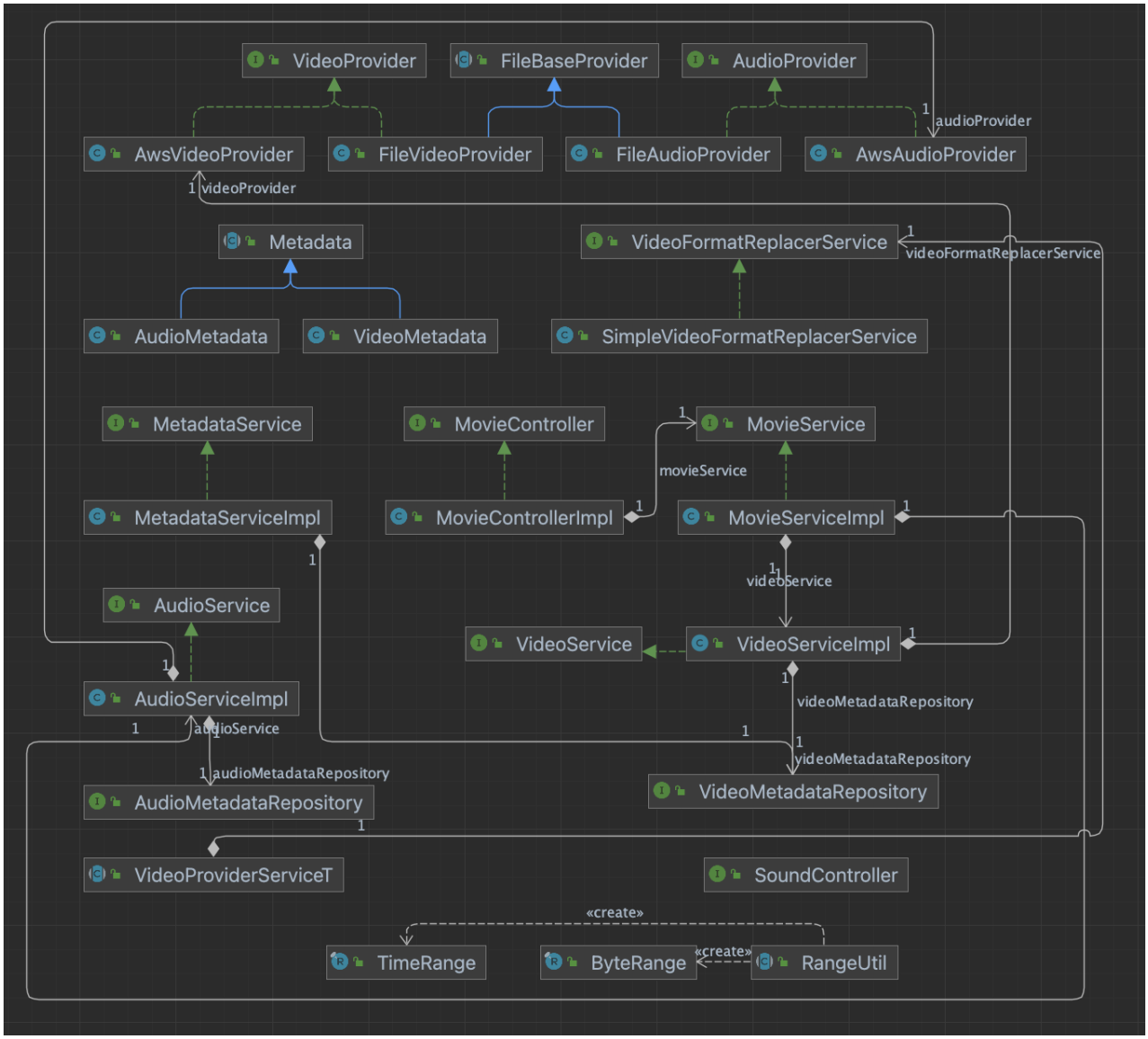

Рисунок 5.8 – Діаграма класів мікросервісу по обробці медіа контенту

На рисунку 5.8 наведено діаграму класів мікросервісу по обробці медіа контенту. У першу чергу необхідно звернути увагу на клас MovieControllerImpl, який відтворено в лістингу 5.20 додатку Д, який на відміну він раніше показаних повертає Flux<DataBuffer>, що представляє собою тип, який слідує парадигмі реактивного програмування. Даний клас містить MovieService інтерфейс, який реалізовує MovieServiceImpl. Даний клас наведено в лістингу 5.21 додатку Д. Він також оперує даними типами даних. У свою чергу, він містить VideoServiceImpl, який відповідає за обердання сомого відео без звуку та AudioServiceImpl, який відповідає за одержання аудіо до певного відео. Також на діаграмі навенено постачальники відео та медіа контенту. Їх ви можите побачити в лістингу 5.22 та 5.23 додатку Д відповідно.

5.3 Порівняння блокуючого та неблокуючого підходу

Як вже було описано – два найбільш навантажені мікросервіси використовують парадигму реактивного програмування – Routing Service та Movie Service. Це забезпеч високу продуктивність та мінімальну кількість затрачених ресурсів.

Розпочнімо огляд з найбільш навантаженого – Routing Service. Даний сервіс повинен не тільки обробляти вхідні запити, але перевіряти їх та обробляти відповіді до клієнтів. Основна відмінність між Spring Gateway на основі блокування потоків і Spring Gateway на основі реактивної парадигми (неблокуючі потоки) полягає в їхній основній архітектурі та підході до обробки запитів і паралельності.

У реактивній парадигмі використовуються неблокуючі потоки. Він використовує керований подіями та асинхронний підхід, коли потоки не блокуються під час очікування операцій введення-виведення. Натомість вони можуть обробляти кілька запитів одночасно, використовуючи функції зворотного виклику або реактивні

потоки. Ця модель забезпечує кращу масштабованість і використання ресурсів, оскільки потоки не простоюють в очікуванні завершення операцій введення-виведення.

У моделі потоку блокування обробка великої кількості одночасних запитів вимагає значної кількості потоків. Зі збільшенням кількості потоків система може зіткнутися з проблемами масштабованості через накладні витрати на створення потоку, перемикання контексту та конкуренцію за ресурси. Реактивна модель ефективніше обробляє паралелізм, використовуючи неблокуючі операції введення-виведення. Він може ефективно обробляти велику кількість одночасних запитів з меншою кількістю потоків. Реактивні фреймворки, такі як Spring WebFlux, використовують модель циклу подій, яка максимізує використання потоку та зменшує потребу у створенні потоку та перемиканні контексту.

У моделі потоку блокування кожен потік споживає значний обсяг пам’яті, як правило, порядку мегабайтів. Оскільки кількість потоків для обробки одночасних запитів збільшується, споживання пам’яті додатком також збільшується. Реактивна модель споживає менше системних ресурсів, зокрема пам’яті, через меншу кількість необхідних потоків. Реактивні фреймворки оптимізують використання пам’яті шляхом спільного використання потоків і використання легких моделей програмування, керованих подіями.

Важливо враховувати конкретні вимоги, характеристики та аспекти продуктивності вашої програми, вибираючи між моделлю потоку блокування та реактивною парадигмою. Кожен підхід має свої компроміси, і рішення має ґрунтуватися на таких факторах, як очікувана паралельність, потреби в масштабованості та сумісність з існуючими системами та бібліотеками.

Оскільки Movie Service базується на тій ж самій парадигмі, він отримує всі переваги та недоліки, що було описано вище. Даний мікросервіс одержує відео конткнт, у нашому випадку з AWS S3, який перетворює у реактивний потік масивів байтів, то реактивно його обробляє і так само надсилає користувачеві.

Далі розглянемо діаграму на якій зображено порівняння виконання коду з використанням реактивної парадигми та потокоблокуючої. Середовище на якому було виконано тестування це AWS EC2 t2.small 1vCPU / 2GB RAM. Дані можна переглянути на рисунку 5.9.

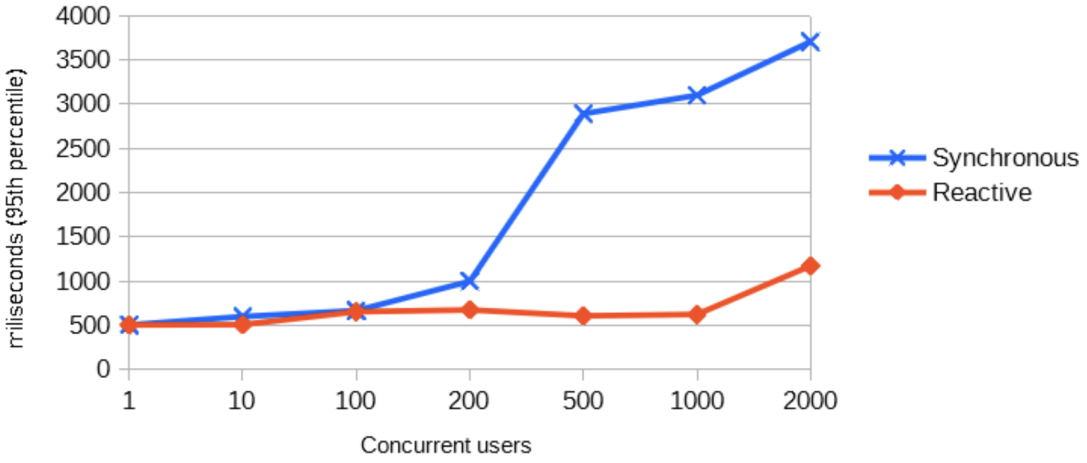

Рисунок 5.9 – Діаграма результатів порівняння

На рисунку 5.9 наведено діаграму результатів порівняння реактивного підходу та потокоблокуючого. На ній можна побачити кількість користувачів, які одночасно роблять запит на сервер та середній час очікування на виконання цих запитів. Можна побачити невелику перевагу потокоблокуючого підходу, коли кількість одночасних запитів не перевищує 100 в один момент. Далі, коли кількість запитів перевищує цю поділку, час очікування на виконання всіх запитів стрімко зростає у потокоблокуючого підходу, а у реактивного зменшується або несильно зростає [15].

Таким чином можна побачити несумнівну перевагу реактивного підходу у випадку високонавантажених систем.

5.3 CI/CD процеси під час розробки

Безперервна інтеграція — це практика розробки програмного забезпечення, яка передбачає регулярне об’єднання змін коду від кількох розробників у центральне сховище та виконання автоматизованих збірок і тестів для виявлення проблем інтеграції на ранніх стадіях процесу розробки. TeamCity — це популярний сервер CI, який допомагає полегшити процес CI та керувати ним.

Він заохочує використання автоматизованих тестів і інструментів аналізу коду. Проводячи ці тести та аналізи під час процесу CI, потенційні помилки, вразливості або запахи коду можна виявити на ранній стадії. Це допомагає підтримувати вищу загальну якість коду та зменшує накопичення технічної заборгованості.

Далі, буде розглянуто створений TeamCity проєкт для даного додатку. Даний TeamCity проєкт буде збирати кожен мікросервіс та запускати юніт тести. Також, щоб результат збірки був успішний кожен мікросервіс необхідний бути покритий юніт тестами мінімум на 90%. Приклад TeamCity проєкту наведено на рисунку 5.10.

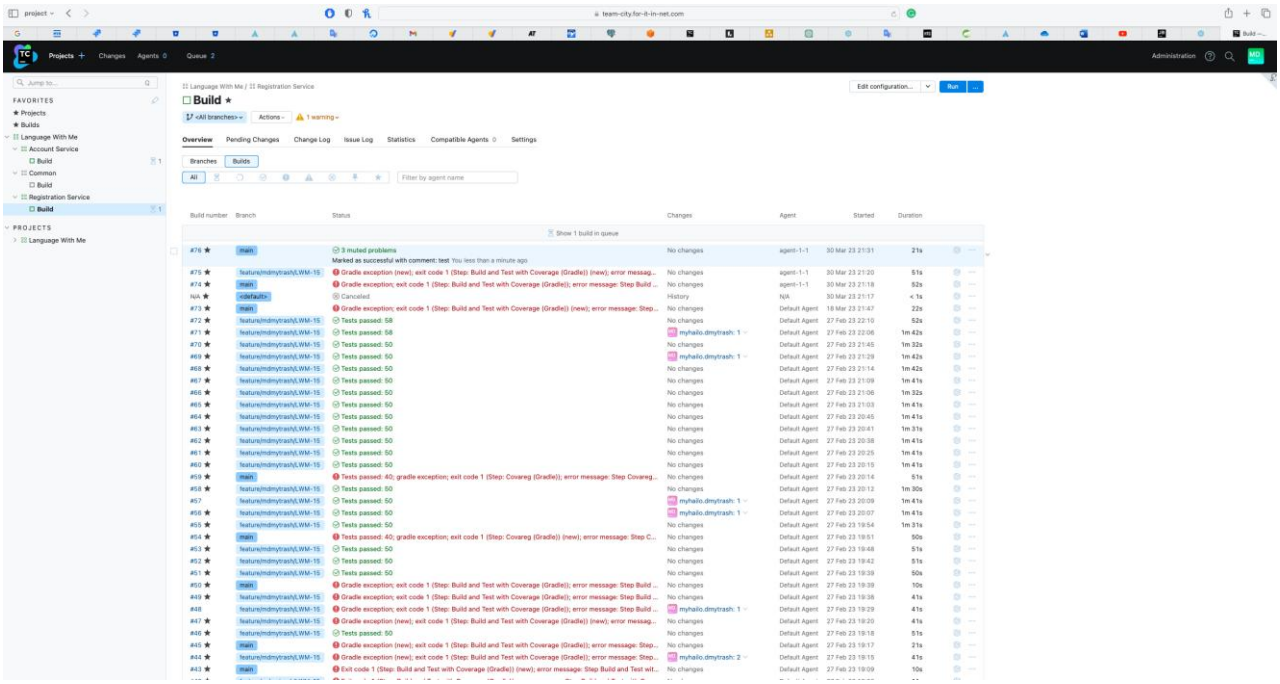

Рисунок 5.10 – Приклад проєкту TeamCity

На рисунку 5.10 зображено приклад проєкту TeamCity. На ньому можна побачити історію збірки мікросервісу, який відповідає за реєстрацію.

Також, після збірки, jar файл проєкту розміщається на Artifactory автоматично, завдяки механізмам TeamCity. Зображення проєкту Artifactory зображено на рисунку 5.11.

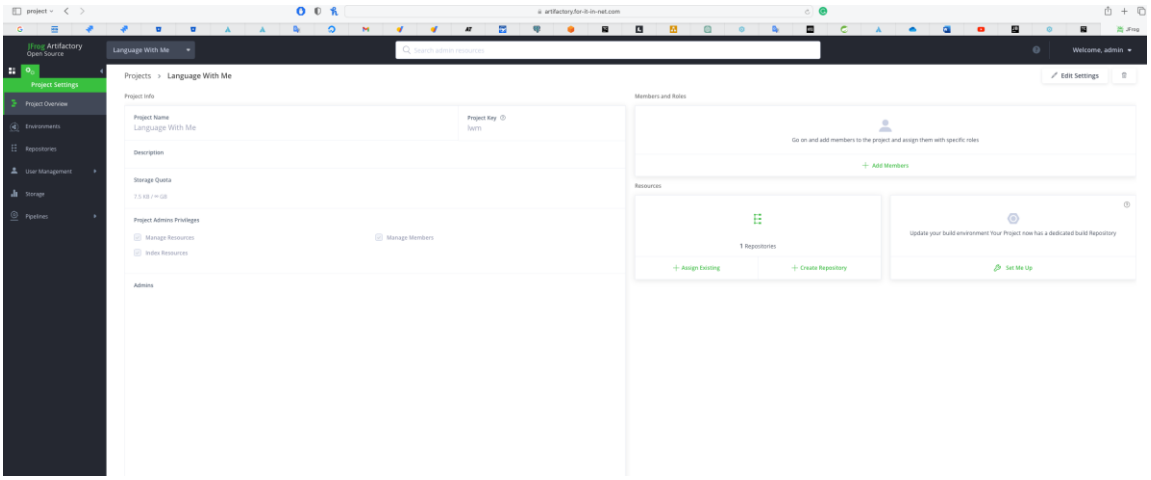

Рисунок 5.11 – Приклад проєкту Artifactory

На рисунку 5.11 зображено приклад проєкту Artifactory. На даному CD інструменті будуть розміщуватись jar файли мікросервісів для подальшого їх використання. Це все забезпечується механізмами TeamCity.

РОЗДІЛ 6 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ

6.1 Роль центральної нервової системи в трудовій діяльності людини

Центральна нервова система (ЦНС) відіграє важливу роль у трудовій діяльності людини. ЦНС складається з головного мозку та спинного мозку і відповідає за контроль та координацію рухів, сприйняття і обробку інформації, планування та виконання складних завдань.

Однією з головних функцій ЦНС є координація рухів. Головний мозок, зокрема, відповідає за планування та виконання точних рухів, необхідних для виконання різних видів роботи. Відправляючи сигнали до м'язів через спинний мозок, головний мозок дозволяє нам виконувати рухи рук, ніг, м'язів тулуба тощо, що необхідно під час виконання різноманітних робочих завдань [16].

Крім того, ЦНС відіграє ключову роль у сприйнятті та обробці інформації. Головний мозок забезпечує сприйняття зовнішніх подразників, таких як зорова, слухова, нюхова і тактильна інформація, які можуть бути важливими під час виконання роботи. Він також відповідає за обробку цієї інформації та прийняття рішень щодо відповідних дій.

ЦНС також відіграє роль у плануванні та виконанні складних завдань. Головний мозок дозволяє нам розробляти стратегії та плани дій, вирішувати проблеми, аналізувати інформацію та приймати рішення. Це особливо важливо для професій, які вимагають високого рівня когнітивних навичок, творчого мислення та прийняття швидких рішень.

Розробники ПЗ повинні аналізувати складну інформацію, розуміти її сутність та перетворювати на код програми. Центральна нервова система допомагає в сприйнятті інформації з різних джерел і обробці її для подальшого використання. Розробники ПЗ використовують комп'ютерну клавіатуру, мишу та інші пристрої для

взаємодії з комп'ютером. ЦНС забезпечує координацію рухів, необхідних для введення коду, навігації по програмі та інших дій.

Розробка програмного забезпечення вимагає високої концентрації і уваги до деталей. ЦНС допомагає підтримувати увагу на завданні, фільтрувати непотрібну інформацію та зосереджуватися на потрібних елементах. Розробники ПЗ повинні знати і використовувати різноманітні концепції, мови програмування, бібліотеки та інші інструменти. ЦНС забезпечує пам'ять, яка дозволяє зберігати та використовувати цю інформацію при роботі.

Розробка програмного забезпечення вимагає прийняття рішень та творчого підходу до вирішення проблем. ЦНС забезпечує когнітивні функції, необхідні для аналізу ситуації, генерації альтернативних рішень та оцінки їх ефективності.

Загалом, ЦНС відіграє ключову роль у трудовій діяльності розробника програмного забезпечення, забезпечуючи обробку інформації, контроль рухів, концентрацію, пам'ять, прийняття рішень та творчість. Збереження здоров'я і підтримка функцій ЦНС є важливими аспектами для ефективної праці розробника ПЗ.

Здоров'я центральної нервової системи є критичним для успішної трудової діяльності розробника програмного забезпечення. Розробка програмного забезпечення може бути викликом і супроводжуватися високим рівнем стресу. Постійний стрес може негативно впливати на ЦНС і призводити до проблем з концентрацією, пам'яттю та прийняттям рішень. Важливо займатися стрес-менеджментом, включаючи регулярний відпочинок, фізичну активність та практики релаксації.

Центральна нервова система також відіграє роль в регуляції емоцій та стресу під час трудової діяльності. Головний мозок має вплив на вироблення емоційних реакцій і вміє регулювати рівень стресу. Це особливо важливо в робочих ситуаціях, де може виникати підвищений рівень напруги та стресові ситуації. ЦНС допомагає управляти емоційним станом та адаптуватися до вимог робочого середовища.

Крім того, ЦНС має вплив на когнітивні функції, такі як пам'ять, увага, концентрація та рішення проблем. Головний мозок дозволяє нам обробляти інформацію, зберігати спогади, використовувати попередні знання та застосовувати їх у робочих ситуаціях. Це допомагає нам ефективно працювати, вирішувати завдання та розв'язувати проблеми, що виникають у робочому процесі.

Вона впливає на регуляцію фізіологічних функцій, таких як дихання, серцево-судинна система та регуляція температури тіла. Вона підтримує необхідну енергетичну активність тіла під час виконання трудових завдань та забезпечує гомеостаз організму.

Розробникам програмного забезпечення важливо дбати про свою ЦНС, забезпечувати їй достатній відпочинок, фізичну активність, підтримувати психологічний комфорт та здоровий спосіб життя. Це допоможе зберегти продуктивність, творчість та здоров'я під час роботи розробником ПЗ.

Центральна нервова система також впливає на мотивацію та настрій під час трудової діяльності. Головний мозок виробляє нейротрансмітери, такі як серотонін, дофамін та ендорфіни, які впливають на наші емоції, настрій та відчуття задоволення. Високий рівень мотивації та позитивний настрій сприяють продуктивності та задоволенню в робочому середовищі.

Крім того, ЦНС відіграє роль у сприйнятті болю. Вона допомагає нам реагувати на потенційно небезпечні сигнали та відчувати біль, що може бути важливим у роботі, де можуть виникати ризиковані ситуації або фізичні перевантаження.

Також, ЦНС має вплив на вироблення і регулювання сну та бодрощів. Вона контролює циркадіанний ритм та розподіл енергії протягом дня, що може впливати на робочу продуктивність та здатність до концентрації.

Вона грає важливу роль у реакції на стресові ситуації під час трудової діяльності. Коли ми зазнаємо стресу, ЦНС активує реакцію "боротьба або втеча" за допомогою вивільнення гормонів стресу, таких як адреналін та кортизол. Це може

підвищити наше фізичне та психологічне бодрствування, покращити концентрацію та швидкість реакції, що може бути корисним у ситуаціях, коли потрібно швидко реагувати на небезпеку або виконати важке завдання.

Однак, тривала експозиція до стресу може мати негативний вплив на ЦНС та трудову діяльність. Високі рівні стресу можуть спричиняти втому, зниження уваги та концентрації, погіршення пам'яті та прийняття рішень, а також збільшення ризику виникнення помилок.

Дотримання здорового способу життя та стратегій керування стресом, таких як фізична активність, релаксаційні техніки, планування та організація робочого часу, може допомогти зменшити негативний вплив стресу на ЦНС та покращити продуктивність у роботі.

Отже, ЦНС взаємодіє зі стресовими факторами трудової діяльності та впливає на нашу реакцію на стрес, що може мати як позитивний, так і негативний вплив на продуктивність та добробут у робочому середовищі.

За допомогою центральної нервової системи, людина може розвивати та вдосконалювати свої навички та вміння. Головний мозок має здатність до нейропластичності, що означає, що він може змінюватися та адаптуватися до нових умов та вимог. Це дозволяє нам набувати нові знання, виробляти навички та покращувати робочі вміння [17].

Наприклад, коли ми навчаємося нової професії або впроваджуємо нові технології у свою роботу, наша ЦНС пристосовується до нових викликів шляхом створення нових нейронних зв'язків та зміни сильності існуючих зв'язків між нейронами. Це дозволяє нам покращувати свою ефективність та продуктивність у виконанні професійних завдань.

Крім того, ЦНС контролює інтелектуальні процеси, такі як мислення, уява, творчість та інноваційність. Головний мозок забезпечує нам здатність до аналітичного мислення, проблемного вирішення та генерації нових ідей, що може бути важливим у творчій та інноваційній роботі.

Узагалі, центральна нервова система виконує комплексні функції, що допомагають людині ефективно функціонувати у трудовій діяльності. Вона забезпечує координацію рухів, сприйняття і обробку інформації, регулювання емоційного стану, когнітивні функції, фізіологічну регуляцію, мотивацію та інші аспекти, що впливають на нашу працездатність та результативність у роботі.

6.2 Оцінка травмонебезпеки технологічного процесу

Оцінка травмонебезпеки технологічного процесу є важливим кроком у забезпеченні безпеки та охорони здоров'я працівників. Цей процес передбачає ідентифікацію потенційних небезпек та ризиків, пов'язаних з виконанням конкретного технологічного процесу, а також оцінку їх впливу на працівників. Основною метою є забезпечення належних заходів безпеки та запобігання нещасним випадкам та професійним захворюванням.

Ідентифікація потенційних небезпек включає визначення можливих джерел травм та небезпечних ситуацій, пов'язаних з технологічним процесом. Це можуть бути фізичні небезпеки, такі як рухомі частини машин, високі температури, шум, вібрація, або хімічні небезпеки, пов'язані з використанням шкідливих речовин. Також оцінюються ризики, пов'язані з виявленими небезпеками. Враховуються фактори, такі як інтенсивність та тривалість експозиції, ймовірність виникнення нещасних випадків або захворювань, а також вразливість працівників.

Визначення заходів з управління ризиком. На основі оцінки ризиків приймаються заходи з мінімізації травмонебезпеки. Це можуть бути технічні заходи, такі як використання захисних установок або автоматизація процесу, організаційні заходи, наприклад, встановлення процедур безпеки та навчання працівників їх дотриманню, або особисті заходи, які включають використання індивідуальних

засобів захисту. Після впровадження заходів з управління ризиком необхідно встановити систему моніторингу, щоб перевіряти їх ефективність та вчасно вносити необхідні зміни, якщо виявляться нові небезпеки або ризики.

Важливо пам'ятати, що оцінка травмонебезпеки технологічного процесу є процесом, який вимагає фахової експертизи. Кваліфіковані фахівці з охорони праці повинні бути залучені для проведення детальної оцінки та розробки відповідних заходів безпеки. Виявлення можливих джерел травм у технологічному процесі розробки програмного забезпечення. Це можуть бути фізичні чинники, такі як погана ергономіка робочого місця, поганий освітлювальний устаткування, погана вентиляція, або психологічні фактори, такі як високий рівень стресу або монотонна робота [17].

Визначення і оцінка ризику, пов'язаного з ідентифікованими небезпеками. Це включає аналіз можливих наслідків травм та ймовірність їх виникнення. Розробка та впровадження заходів для зменшення ризику травм. Це може включати поліпшення ергономіки робочих місць, навчання розробників правильним методам роботи, використання захисного обладнання, створення режимів роботи та пауз для відпочинку.Проведення навчань та інструктажів з питань безпеки роботи, щоб розробники могли свідомо управляти ризиками та дотримуватись правил безпеки.

Постійний моніторинг та оцінка травмонебезпеки технологічного процесу, а також внесення вдосконалень в систему безпеки на основі отриманих даних та фідбеку.

Важливо розуміти, що травмонебезпека може варіюватися залежно від конкретних умов і процесів розробки програмного забезпечення. Тому, розглядаючи оцінку травмонебезпеки, рекомендується звернутися до специфічних нормативних актів та рекомендацій, а також консультуватися з фахівцями з охорони праці та безпеки, щоб забезпечити найвищий рівень безпеки для розробників програмного забезпечення.

Забезпечення безпеки та запобігання травмам у технологічному процесі розробки програмного забезпечення є надзвичайно важливим завданням. Ось кілька додаткових аспектів, які можуть бути враховані при оцінці травмонебезпеки. Застосування сучасних інструментів та програмних засобів, які допомагають знизити ризик виникнення помилок та проблем. Наприклад, використання систем керування версіями, автоматизованих тестувань та аналізу коду може допомогти уникнути програмних помилок, які можуть призвести до непередбачених наслідків.

Завершення оцінки травмонебезпеки технологічного процесу передбачає документування результатів і прийняття необхідних заходів забезпечення безпеки. Оцінка та управління травмонебезпекою повинні стати постійним процесом, оскільки технологічні процеси можуть змінюватися, виникати нові ризики та появлятися нові технології.

Крім оцінки травмонебезпеки, також слід враховувати інші аспекти безпеки та охорони здоров'я працівників, такі як ергономіка робочого місця, хімічна безпека, пожежна безпека та інші фактори, що можуть впливати на безпеку праці.

Додатково, важливо залучити працівників до процесу оцінки травмонебезпеки. Вони можуть мати цінні відомості про потенційні небезпеки, знати практичні аспекти робочого процесу та внести свої пропозиції щодо запобіжних заходів. Залучення працівників до формулювання та реалізації безпекових заходів сприяє покращенню свідомості про безпеку та забезпеченню більш ефективного впровадження заходів безпеки.

Необхідно також забезпечити постійне навчання та підвищення кваліфікації працівників щодо безпеки та охорони здоров'я на робочому місці. Це може включати навчання щодо використання захисного спорядження, процедур безпеки, першої допомоги та інших аспектів, необхідних для запобігання нещасним випадкам та мінімізації ризиків.

Загальною метою оцінки травмонебезпеки технологічного процесу є створення безпечного та здорового робочого середовища, де працівники можуть працювати

ефективно та безпечно. Оцінка травмонебезпеки допомагає ідентифікувати ризики та розробити план дій для запобігання нещасним випадкам та забезпечення безпеки працівників. Запобігання нещасним випадкам та забезпечення безпеки на робочому місці є постійним процесом, який вимагає системного підходу та участі всіх сторін: роботодавців, керівництва, спеціалістів з охорони праці та самими працівниками.

Правильне впровадження та дотримання вимог безпеки дозволяє знизити ризик нещасних випадків, травм та захворювань, покращити ефективність трудового процесу та загальну якість роботи. Організації повинні створити відповідні політики та процедури, а також забезпечити належне навчання та підтримку працівників у питаннях безпеки праці.

Крім того, важливо постійно вдосконалювати системи управління безпекою, враховуючи нові технології, законодавство та інновації. Постійний моніторинг стану безпеки та оцінка ризиків дозволяють вчасно виявляти потенційні небезпеки та вживати відповідних заходів для їх запобігання.

Безпека праці має велике значення як для працівників, так і для організацій. Забезпечення безпеки на робочому місці сприяє збереженню здоров'я, зниженню відсутності праці та підвищенню продуктивності. Крім того, це впливає на репутацію організації та сприяє залученню та збереженню кваліфікованих працівників.

Забезпечення безпеки в процесі розробки програмного забезпечення є постійним завданням, що потребує уваги до деталей, свідомого підходу та співпраці всіх учасників процесу.

ВИСНОВКИ

Підбиваючи підсуми даного дослідження, можна сміливо стверджувати, що парадигма реактивного програмування має неймовірно великий потенціал. В зв’язці з мікросервісною архітектурою воно дозволяє розробляти додатки на новому рівні. Самі проєкти стають більш гнучкими, а розробка та їх підтримка простішими.

Реактивне програмування — це підхід, який акцентує увагу на асинхронному програмуванні, керованому подіями, для обробки потоків даних і подій. Це дозволяє розробникам створювати більш чуйні, масштабовані та стійкі системи, використовуючи неблокуючий ввід-вивід, комунікацію на основі повідомлень та інші шаблони реагування.

Архітектура мікросервісу, з іншого боку, — це шаблон проектування програмного забезпечення, де додаток розділено на слабко пов’язані служби, які можна розгортати незалежно один від одного, які взаємодіють між собою через API. Мікросервіси забезпечують такі переваги, як масштабованість, ізоляція помилок і гнучкість у виборі технологій.

У цьому дослідженні ви, ймовірно, вивчатимете, як поєднати парадигми реактивного програмування з архітектурою мікросервісів для розробки високонавантажених програм. Високонавантажені програми зазвичай стосуються систем, які, як очікується, оброблятимуть великий обсяг одночасних запитів або обробки даних. Використовуючи реактивне програмування та мікросервіси, ви прагнете досягти кращої продуктивності, масштабованості та швидкості реагування.

Досліджуйте та розумійте різні фреймворки реактивного програмування, бібліотеки та шаблони, які можна використовувати для реалізації високонавантажених програм. Вивчіть принципи та найкращі практики проектування мікросервісів, включаючи декомпозицію сервісів, протоколи зв’язку, механізми відмовостійкості та стратегії розгортання.

Оцініть, як реактивне програмування та мікросервіси можуть покращити продуктивність і масштабованість високонавантажених програм, забезпечуючи ефективне використання системних ресурсів і ефективну обробку паралелізму.

Дослідіть, як реактивне програмування та мікросервіси можуть підвищити стійкість системи, відмовостійкість і можливості обробки помилок для забезпечення надійної роботи за високих навантажень. Дослідіть практичні аспекти впровадження реактивних програм в архітектурі мікросервісу, наприклад вибір відповідних технологій, визначення протоколів зв’язку, обробка узгодженості даних і керування складністю системи.

Літній проект, який ви описали, здається, передбачає глибоке занурення в перетин реактивного програмування та мікросервісів для вивчення потенційних переваг і проблем у розробці високонавантажених програм. Це може бути захоплюючим і складним дослідницьким завданням, яке потребує глибокого розуміння архітектури програмного забезпечення, парадигм програмування та методів оптимізації продуктивності.

ПЕРЕЛІК ПОСИЛАНЬ

1. Harris C. Microservices vs. monolithic architecture. *Atlassian*. URL: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith (дата звернення: 09.06.2023).

2. Fitzgibbons L. What are front end and back end? Definition from WhatIs.com. WhatIs.com. URL: https://www.techtarget.com/whatis/definition/front-end (дата звернення: 02.06.2023).

3. How to Choose a Programming Language For a Project? - GeeksforGeeks. GeeksforGeeks. URL: https://www.geeksforgeeks.org/how-to-choose-a-programming-language-for-a-project/ (дата звернення: 01.06.2023).

4. How to Choose the Right Technology Stack for Your Project? | SCAND. SCAND. URL: https://scand.com/company/blog/choosing-a-technology-stack/ (дата звернення: 02.06.2023).

5. Why Millions of Developers use JavaScript for Web Application Development?. Torque. URL: https://torquemag.io/2018/06/why-millions-of-developers-use-javascript-for-web-application-development/ (дата звернення: 03.06.2023).

6. What Is a Single Page Application? | Bloomreach. Bloomreach. URL: https://www.bloomreach.com/en/blog/2018/what-is-a-single-page-application (дата звернення: 04.05.2023).

7. Pros and Cons of Single-Page Applications. Spiceworks. URL: https://www.spiceworks.com/tech/devops/articles/what-is-single-page-application/ (дата звернення: 13.06.2023).

8. What is the Angular framework?. mDevelopers - Your Custom Software Development Company. URL: https://mdevelopers.com/blog/what-is-the-angular-framework (дата звернення: 19.05.2023).

9. What Is Java Used For?. Coursera. URL: https://www.coursera.org/articles/what-is-java-used-for (дата звернення: 12.05.2023).

10. Introduction of Object Oriented Programming - GeeksforGeeks. GeeksforGeeks. URL: https://www.geeksforgeeks.org/introduction-of-object-oriented-programming/ (дата звернення: 13.05.2023).

11. Functional Programming Paradigm - GeeksforGeeks. GeeksforGeeks. URL: https://www.geeksforgeeks.org/functional-programming-paradigm/ (дата звернення: 20.05.2023).

12. What Is Reactive Programming? | Baeldung on Computer Science. Baeldung on Computer Science. URL: https://www.baeldung.com/cs/reactive-programming (дата звернення: 11.05.2023).

13. What Is Reactive Programming? | Baeldung on Computer Science. Baeldung on Computer Science. URL: https://www.baeldung.com/cs/reactive-programming (дата звернення: 11.05.2023).

14. Introduction to Redis (What it is, what are the use cases etc) | Severalnines. Severalnines. URL: https://severalnines.com/blog/introduction-redis-what-it-what-are-use-cases-etc (дата звернення: 19.05.2023).

15. Cunha G. T. Reactive vs. Synchronous Performance Test - DZone. dzone.com. URL: https://dzone.com/articles/spring-boot-20-webflux-reactive-performance-test (дата звернення: 10.06.2023).

16. Яворовський О., Шевцова В., Зенкіна В. Безпека життєдіяльності, основи охорони праці: навчальний посібник. 2-ге вид. Всеукр. спеціаліз. вид-во «Медицина», 2018. 288 с.

17. Безпека життєдіяльності та основи охорони праці :: Державний університет телекомунікацій. Державний університет телекомунікацій. URL: https://www.dut.edu.ua/ua/lib/1/category/760/view/921 (дата звернення: 15.06.2023).

ДОДАТКИ

ДОДАТОК А СЕРВІС МАРШРУТИЗАЦІЇ

Лістинг 5.1:

```
plugins {
    id 'org.springframework.boot' version '2.7.4'
    id 'io.spring.dependency-management' version '1.0.14.RELEASE'
    id 'java'
}

group = 'com.foryouinnet'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

repositories {
    mavenCentral()
}

ext {
    set('springCloudVersion', "2021.0.4")
}

dependencies {
    implementation 'org.springframework.cloud:spring-cloud-starter-
config'
    implementation 'org.springframework.cloud:spring-cloud-starter-
gateway'
    implementation 'org.springframework.cloud:spring-cloud-starter-
netflix-eureka-client'
    implementation 'org.springframework.cloud:spring-cloud-starter-
bootstrap:3.1.3'

    implementation 'org.springframework.boot:spring-boot-starter-
security'


    implementation 'com.playtika.reactivefeign:feign-reactor-spring-
cloud-starter:3.2.6'
    implementation 'com.netflix.feign:feign-jackson:8.18.0'


    implementation 'org.springframework.boot:spring-boot-starter-
web:2.7.5'

    implementation 'org.hibernate:hibernate-validator:7.0.4.Final'

    compileOnly 'org.projectlombok:lombok:1.18.24'
    annotationProcessor 'org.projectlombok:lombok:1.18.24'
```

```
        developmentOnly 'org.springframework.boot:spring-boot-devtools'
        developmentOnly 'org.springframework.boot:spring-boot-starter-
actuator'

        testImplementation 'org.springframework.boot:spring-boot-starter-
test'
    }

    dependencyManagement {
        imports {
            mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${springCloudVersion}"
        }
    }

    tasks.named('test') {
        useJUnitPlatform()
    }
```

Лістинг 5.2:

```
    package com.foryouinnet.routingservice.configurations;

    import lombok.RequiredArgsConstructor;
    import org.springframework.cloud.gateway.route.RouteLocator;
    import
org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;
    import org.springframework.context.annotation.Bean;
    import org.springframework.context.annotation.Configuration;

    @Configuration
    @RequiredArgsConstructor
    public class RoutingConfiguration
    {
        @Bean
        public RouteLocator routeLocator(RouteLocatorBuilder
routeLocatorBuilder)
        {
            return routeLocatorBuilder.routes()
                    .route(route -> route
                            .path("/registration/register",
"/registration/confirm", "/registration/resend-code")
                            .filters(gatewayFilterSpec -> gatewayFilterSpec

.rewritePath("/registration/(?<remaining>.*)", "/registration-
service/registration/${remaining}"))
                            .uri("lb://REGISTRATION-SERVICE"))
                    .route(route -> route
                            .path("/authentication")
```

```
                                .filters(gatewayFilterSpec -> gatewayFilterSpec
.rewritePath("/authentication/(?<remaining>.*)", "/oauth2/${remaining}"))
                                .uri("lb://AUTHENTICATION-SERVICE")
                        )
                        .build();
            }
        }
```

ДОДАТОК Б СЕРВІС НАЛАШТУВАНЬ

Лістинг 5.5:

```
plugins {
id 'org.springframework.boot' version '2.7.4'
id 'io.spring.dependency-management' version '1.0.14.RELEASE'
id 'java'
}

group = 'com.foryouinnet'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

repositories {
mavenCentral()
}

ext {
set('springCloudVersion', "2021.0.4")
}

dependencies {
implementation 'org.springframework.cloud:spring-cloud-starter-
bootstrap:3.1.3'
implementation 'org.springframework.cloud:spring-cloud-config-server'
implementation 'org.springframework.cloud:spring-cloud-starter-netflix-
eureka-client'
implementation 'org.springframework.cloud:spring-cloud-starter'
developmentOnly 'org.springframework.boot:spring-boot-devtools'
testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

dependencyManagement {
imports {
     mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${springCloudVersion}"
}
}

tasks.named('test') {
useJUnitPlatform()
}
```

Лістинг 5.6:

```
eureka:
```

```
    client:
      service-url:
        defaultZone: http://localhost:8082/eureka
    instance:
      instance-id:
${spring.application.name}:${spring.application.instance_id:${random.value}}

  password:
    configuration:
      prefix: encoded

  authentication:
    service:
      client:
        name: client
        secret: secret

  application:
    urls:
      account-service:
        base-url: /account-service
        controllers:
          creation:
            controller-url: /creation
            endpoints:
              exist: /exist
              save: /save
          authentication:
            controller-url: /authentication
            endpoints:
              get: /get

      registration-service:
        base-url: /registration-service
        controllers:
          registrations:
            controller-url: /registration
            endpoints:
              register: /register
              confirm: /confirm
              resend-code: /resend-code

      authentication-service:

      authority-service:
        base-url: /authority-service
        controllers:
          account-registration:
            controller-url: /account
            endpoints:
              register: /init
              get: /get
```

```
logging:
  level:
    com:
      foryouinnet: ${project.logging.level:debug}
```

Лістинг 5.7:

```
package com.foryouinnet.configurationservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@EnableConfigServer
@EnableEurekaClient
@SpringBootApplication
public class ConfigurationServiceApplication {

public static void main(String[] args) {
      SpringApplication.run(ConfigurationServiceApplication.class,
args);
}

}
```

ДОДАТОК В СЕРВІС РЕЄСТРАЦІЇ

Лістинг 5.8:

```
package com.foryouinnet.registrationservice.controllers.impl;

import
com.foryouinnet.registrationservice.controllers.RegistrationController;
import com.foryouinnet.registrationservice.models.dto.AccountDto;
import
com.foryouinnet.registrationservice.models.dto.ConfirmationCodeDto;
import com.foryouinnet.registrationservice.services.RegistrationService;
import
com.foryouinnet.registrationservice.services.impl.RegistrationServiceImpl;
import lombok.RequiredArgsConstructor;
import lombok.extern.log4j.Log4j2;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.RestController;

import javax.validation.Validator;

@Log4j2
@RestController
@RequiredArgsConstructor
public class RegistrationControllerImpl implements
RegistrationController {

    private final RegistrationServiceImpl registrationService;

    @Override
    public ConfirmationCodeDto register(AccountDto accountDto) {

        log.info("Trying to create registration request for {}",
accountDto::getUsername);

        String id = registrationService.register(accountDto);

        log.info("Registration request {} was created.",
accountDto::getUsername);

        return ConfirmationCodeDto.builder()
                .hash(id)
                .build();
    }

    @Override
    public void confirm(ConfirmationCodeDto confirmationCodeDto) {
```

```
            log.info("Trying to confirm registration request with {} hash.",
confirmationCodeDto::getHash);

            registrationService.confirm(confirmationCodeDto);

            log.info("Registration request with hash {} was confirmed.",
confirmationCodeDto::getHash);
        }

        @Override
        public String resendCode(ConfirmationCodeDto confirmationCodeDto) {

            log.info("Trying to resend confirm code for hash {}.",
confirmationCodeDto::getHash);

            String id =
registrationService.resendCode(confirmationCodeDto.getHash());

            log.info("Code for hash {} was resent.",
confirmationCodeDto::getHash);

            return id;
        }
    }
```

Лістинг 5.9:

```
    package com.foryouinnet.registrationservice.services.impl;

    import com.foryouinnet.registrationservice.clients.AccountServiceClient;
    import com.foryouinnet.registrationservice.clients.MessageSenderClient;
    import
com.foryouinnet.registrationservice.exceptions.RegistrationServiceException;
    import com.foryouinnet.registrationservice.models.dto.AccountDto;
    import
com.foryouinnet.registrationservice.models.dto.ConfirmationCodeDto;
    import com.foryouinnet.registrationservice.services.AccountService;
    import
com.foryouinnet.registrationservice.services.ConfirmationCodeService;
    import com.foryouinnet.registrationservice.services.RegistrationService;
    import lombok.NonNull;
    import lombok.RequiredArgsConstructor;
    import lombok.extern.log4j.Log4j2;
    import org.springframework.stereotype.Service;
    import org.springframework.transaction.annotation.Transactional;
```

```
@Log4j2
@Service
@Transactional
@RequiredArgsConstructor
public class RegistrationServiceImpl implements RegistrationService {

    private final AccountServiceImpl accountService;
    private final ConfirmationCodeServiceImpl confirmationCodeService;
    private final MessageSenderClient messageSenderClient;
    private final AccountServiceClient accountServiceClient;

    @Override
    @Transactional
    public String register(@NonNull AccountDto accountDto) {

        if(accountServiceClient.cvxcvcvexists(accountDto.getEmail(),
accountDto.getUsername())) {
            log.debug("Account with email {} or username {} already
exists", accountDto::getEmail, accountDto::getUsername);
            throw new
RegistrationServiceException(RegistrationServiceException.EXISTS);
        }

        accountDto = accountService.register(accountDto);
        log.debug("Account with username {} was registered",
accountDto::getUsername);

        ConfirmationCodeDto confirmationCodeDto =
confirmationCodeService.create();
        log.debug("Confirmation code for account with username {} was
created", accountDto::getUsername);

        accountService.setConfirmationCode(accountDto,
confirmationCodeDto.getId());

        messageSenderClient.sendCode(accountDto.getEmail(),
confirmationCodeDto.getCode());
        log.debug("Confirm code was sent to {}",
accountDto::getUsername);

        return confirmationCodeDto.getHash();
    }

    @Override
    @Transactional
    public String resendCode(@NonNull String hash) {

        AccountDto accountDto =
accountService.getNotExpiredByConfirmationCodeHash(hash);
        log.debug("Account exists and not expired");

        ConfirmationCodeDto confirmationCodeDto =
confirmationCodeService.updateConfirmationCodeData(hash);
        log.debug("Confirm code for was recreated");
```

```
            messageSenderClient.sendCode(accountDto.getEmail(),
confirmationCodeDto.getCode());
            log.debug("Confirm code was sent to {}", accountDto.getEmail());

            return confirmationCodeDto.getHash();
        }

        @Override
        @Transactional
        public void confirm(@NonNull ConfirmationCodeDto
confirmationCodeDto) {

            confirmationCodeDto =
confirmationCodeService.extract(confirmationCodeDto);
            log.debug("Confirmation code was extracted");

            AccountDto accountDto =
accountService.extract(confirmationCodeDto.getId());
            log.debug("Account with {} username extracted",
accountDto::getUsername);

            accountServiceClient.save(accountDto);
            log.debug("Account with username {} was sent to account
service", accountDto::getUsername);
        }
    }
```

Лістинг 5.10:

```
    package com.foryouinnet.registrationservice.services.impl;

    import
com.foryouinnet.registrationservice.exceptions.ConfirmationCodeServiceExcepti
on;
    import
com.foryouinnet.registrationservice.mappers.ConfirmationCodeMapper;
    import
com.foryouinnet.registrationservice.models.dto.ConfirmationCodeDto;
    import
com.foryouinnet.registrationservice.models.entities.ConfirmationCode;
    import
com.foryouinnet.registrationservice.properties.RegistrationRequestProperty;
    import
com.foryouinnet.registrationservice.repositories.ConfirmationCodeRepository;
    import
com.foryouinnet.registrationservice.services.ConfirmationCodeService;
```

```
import com.foryouinnet.registrationservice.utils.DataGeneratorUtil;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
import lombok.extern.log4j.Log4j2;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.time.LocalDateTime;
import java.util.UUID;

@Log4j2
@Service
@RequiredArgsConstructor
public class ConfirmationCodeServiceImpl implements
ConfirmationCodeService {

    private final ConfirmationCodeRepository confirmationCodeRepository;
    private final ConfirmationCodeMapper confirmationCodeMapper;
    private final RegistrationRequestProperty
registrationRequestProperty;
    private final PasswordEncoder passwordEncoder;

    @Override
    @Transactional
    public ConfirmationCodeDto create() {

        String code =
DataGeneratorUtil.generateRandomNumber(registrationRequestProperty.getConfirm
CodeLength());
        ConfirmationCode confirmationCode =
generateConfirmationCode(code);

        confirmationCode =
confirmationCodeRepository.save(confirmationCode);

        ConfirmationCodeDto confirmationCodeDto =
confirmationCodeMapper.confirmationCodeToConfirmationCodeDto(confirmationCode
);
        confirmationCodeDto.setCode(code);

        return confirmationCodeDto;
    }

    @Override
    @Transactional
    public ConfirmationCodeDto updateConfirmationCodeData(@NonNull
String hash) {

        ConfirmationCode confirmationCode =
confirmationCodeRepository.findConfirmationCodeByHash(hash)
                .orElseThrow(() -> new
ConfirmationCodeServiceException(String.format(ConfirmationCodeServiceExcepti
on.NOT_EXIST)));
```

```
            if(!canCodeBeRecreate(confirmationCode)) {
                throw new
ConfirmationCodeServiceException(ConfirmationCodeServiceException.PAUSE);
            }

            String code =
DataGeneratorUtil.generateRandomNumber(registrationRequestProperty.getConfirm
CodeLength());
            confirmationCode =
confirmationCodeRepository.save(updateConfirmationCode(confirmationCode,
code));

            ConfirmationCodeDto confirmationCodeDto =
confirmationCodeMapper.confirmationCodeToConfirmationCodeDto(confirmationCode
);
            confirmationCodeDto.setCode(code);

            return confirmationCodeDto;
        }

        @Override
        @Transactional
        public ConfirmationCodeDto extract(@NonNull ConfirmationCodeDto
confirmationCodeDto) {

            ConfirmationCode confirmationCode =
get(confirmationCodeDto.getHash(), confirmationCodeDto.getCode());

            confirmationCodeRepository.delete(confirmationCode);

            return
confirmationCodeMapper.confirmationCodeToConfirmationCodeDto(confirmationCode
);
        }

        private ConfirmationCode get(@NonNull String hash, @NonNull String
code) {

            return
confirmationCodeRepository.findConfirmationCodeByHash(hash)
                    .filter(confirmationCode ->
passwordEncoder.matches(code, confirmationCode.getCode()))
                    .orElseThrow(() -> new
ConfirmationCodeServiceException(ConfirmationCodeServiceException.WRONG_REGIS
TRATION_ID_OR_CODE));
        }

        private ConfirmationCode generateConfirmationCode(@NonNull String
code) {
            return ConfirmationCode.builder()
                    .hash(UUID.randomUUID().toString())
                    .code(passwordEncoder.encode(code))
                    .codeSentAt(LocalDateTime.now())
```

```
                .build();
        }

        private ConfirmationCode updateConfirmationCode(@NonNull
ConfirmationCode confirmationCode, @NonNull String code) {

            confirmationCode.setHash(UUID.randomUUID().toString());
            confirmationCode.setCode(passwordEncoder.encode(code));
            confirmationCode.setCodeSentAt(LocalDateTime.now());

            return confirmationCode;
        }

        private boolean canCodeBeRecreate(@NonNull ConfirmationCode
confirmationCode) {

            return
LocalDateTime.now().isAfter(confirmationCode.getCodeSentAt().plusSeconds(regi
strationRequestProperty.getConfirmCodeResendingPauseDurationInMs().toSeconds(
)));
        }
    }
```

Лістинг 5.11:

```
    package com.foryouinnet.registrationservice.repositories;

    import
com.foryouinnet.registrationservice.models.entities.ConfirmationCode;
    import lombok.NonNull;
    import org.springframework.data.jpa.repository.JpaRepository;

    import java.util.Optional;

    public interface ConfirmationCodeRepository extends
JpaRepository<ConfirmationCode, String> {

        Optional<ConfirmationCode> findConfirmationCodeByHash(@NonNull
String hash);
    }
```

Лістинг 5.12:

```
package com.foryouinnet.registrationservice.models.entities;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.Generated;
import lombok.NoArgsConstructor;
import lombok.experimental.SuperBuilder;
import org.hibernate.annotations.CreationTimestamp;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;
import javax.persistence.Version;
import java.time.LocalDateTime;

@Data
@Table(name = "accounts")
@Entity
@SuperBuilder
@NoArgsConstructor
@EqualsAndHashCode(of = {"email", "username"})
@AllArgsConstructor
public class Account {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "email", unique = true, updatable = false, nullable =
false)
    private String email;

    @Column(name = "username", unique = true, updatable = false,
nullable = false)
    private String username;

    @Column(name = "password", updatable = false, nullable = false)
    private String password;

    @Column(name = "created_at", updatable = false, nullable = false)
    private LocalDateTime createdAt;

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
```

```
        @JoinColumn(name = "confirmation_code_id", referencedColumnName =
"id")
        private ConfirmationCode confirmationCode;

        @Version
        @Column(name = "version")
        private long version;
    }
```

Лістинг 5.13:

```
    package com.foryouinnet.registrationservice.models.entities;

    import lombok.AllArgsConstructor;
    import lombok.Data;
    import lombok.EqualsAndHashCode;
    import lombok.Generated;
    import lombok.NoArgsConstructor;
    import lombok.experimental.SuperBuilder;
    import org.hibernate.annotations.CreationTimestamp;

    import javax.persistence.Column;
    import javax.persistence.Entity;
    import javax.persistence.FetchType;
    import javax.persistence.GeneratedValue;
    import javax.persistence.GenerationType;
    import javax.persistence.Id;
    import javax.persistence.OneToOne;
    import javax.persistence.Table;
    import javax.persistence.Version;
    import java.time.LocalDateTime;

    @Data
    @Table(name = "confirmation_codes")
    @Entity
    @SuperBuilder
    @NoArgsConstructor
    @EqualsAndHashCode(of = {"hash", "code"})
    @AllArgsConstructor
    public class ConfirmationCode {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "id", unique = true, updatable = false, nullable =
false)
        private Long id;

        @Column(name = "hash", unique = true, nullable = false)
        private String hash;

        @Column(name = "code", nullable = false, updatable = false)
```

```
    private String code;

    @CreationTimestamp
    @Column(name = "code_sent_at", nullable = false, updatable = false)
    private LocalDateTime codeSentAt;

    @OneToOne(mappedBy = "confirmationCode", fetch = FetchType.LAZY)
    private Account account;

    @Version
    @Column(name = "version")
    private long version;
}
```

ДОДАТОК Г СЕРВІС КЕРУВАННЯ АККАУНТАМИ

Лістинг 5.14:

```
package com.foryouinnet.accountservice.controllers.impl;

import
com.foryouinnet.accountservice.controllers.AccountCreationController;
import
com.foryouinnet.accountservice.models.dtos.RegistrationRequestDto;

import com.foryouinnet.accountservice.services.AccountCreationService;
import
com.foryouinnet.accountservice.services.impl.AccountCreationServiceImpl;
import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequiredArgsConstructor
public class AccountCreationControllerImpl implements
AccountCreationController {

    private final AccountCreationServiceImpl accountCreationService;

    @Override
    public boolean isExist(String email, String username) {

        return accountCreationService.isExist(email, username);
    }

    @Override
    public void save(RegistrationRequestDto registrationRequestDTO) {

        accountCreationService.save(registrationRequestDTO);
    }
}
```

Лістинг 5.15:

```
package com.foryouinnet.accountservice.services.impl;

import com.foryouinnet.accountservice.models.dtos.AccountDto;
import com.foryouinnet.accountservice.models.dtos.AuthorityDto;
import
com.foryouinnet.accountservice.models.dtos.RegistrationRequestDto;
import com.foryouinnet.accountservice.models.entities.Account;
import com.foryouinnet.accountservice.services.AccountCreationService;
import com.foryouinnet.accountservice.services.AccountService;
```

```
import com.foryouinnet.accountservice.services.AuthorityService;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
import lombok.extern.log4j.Log4j2;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.Set;

@Log4j2
@Service
@RequiredArgsConstructor
public class AccountCreationServiceImpl implements
AccountCreationService {

    private final AccountServiceImpl accountService;
    private final AuthorityServiceImpl authorityService;

    @Override
    public boolean isExist(@NonNull String email, @NonNull String
username) {

        boolean isExist = accountService.isExist(email, username);

        log.debug("Account exists: {}", isExist);

        return isExist;
    }

    @Override
    @Transactional
    public void save(@NonNull RegistrationRequestDto
registrationRequestDTO) {

        AccountDto accountDto =
accountService.save(registrationRequestDTO);

        authorityService.init(accountDto.getUuid());
    }
}
```

Лістинг 5.16:

```
package com.foryouinnet.accountservice.services.impl;

import com.foryouinnet.accountservice.clients.AuthorityServiceClient;
import com.foryouinnet.accountservice.models.dtos.AuthorityDto;
import com.foryouinnet.accountservice.services.AuthorityService;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
import lombok.extern.log4j.Log4j2;
```

```
import org.springframework.stereotype.Service;

import java.util.Set;

@Log4j2
@Service
@RequiredArgsConstructor
public class AuthorityServiceImpl implements AuthorityService {

    private final AuthorityServiceClient authorityServiceClient;

    @Override
    public Set<AuthorityDto> init(@NonNull String accountUuid) {

        return authorityServiceClient.init(accountUuid);
    }
}
```

Лістинг 5.17:

```
package com.foryouinnet.accountservice.services.impl;

import
com.foryouinnet.accountservice.configuratins.PasswordConfiguration;
import
com.foryouinnet.accountservice.exceptions.AccountServiceException;
import com.foryouinnet.accountservice.mappers.AccountMapper;
import com.foryouinnet.accountservice.models.dtos.AccountDto;
import
com.foryouinnet.accountservice.models.dtos.RegistrationRequestDto;
import com.foryouinnet.accountservice.models.entities.Account;
import com.foryouinnet.accountservice.repositories.AccountRepository;
import com.foryouinnet.accountservice.services.AccountService;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.Set;
import java.util.UUID;
import java.util.stream.Stream;

@Service
@RequiredArgsConstructor
public class AccountServiceImpl implements AccountService {

    private final AccountRepository accountRepository;
    private final AccountMapper accountMapper;
    private final PasswordConfiguration passwordConfiguration;

    @Override
```

```
    public boolean isExist(@NonNull String email, @NonNull String
username) {

        return accountRepository.existsByEmailOrUsername(email,
username);
    }

    @Override
    public AccountDto save(@NonNull RegistrationRequestDto
registrationRequestDTO) {

        if(isExist(registrationRequestDTO.getEmail(),
registrationRequestDTO.getUsername())) {
            throw new
AccountServiceException(AccountServiceException.EXISTS_BY_EMAIL_OR_USERNAME);
        }

        Account account =
accountMapper.registrationRequestDTOtoAccount(registrationRequestDTO);

        account = accountRepository.save(prepareToSave(account));

        return accountMapper.accountToAccountDTO(account);
    }

    @Override
    public AccountDto get(@NonNull String email, @NonNull String
password) {

        return accountRepository.getByEmail(email)
                .filter(account ->
passwordConfiguration.getPasswordEncoder().matches(password,
account.getPassword()))
                .map(accountMapper::accountToAccountDTO)
                .orElseThrow(() -> new
AccountServiceException(AccountServiceException.WRONG_EMAIL_OR_PASSWORD));
    }

    @Override
    public AccountDto get(@NonNull String email) {

        return accountRepository.getByEmail(email)
                .map(accountMapper::accountToAccountDTO)
                .orElseThrow(() -> new
AccountServiceException(AccountServiceException.WRONG_EMAIL_OR_PASSWORD));
    }

    private Account prepareToSave(Account account) {

        account.setUuid(UUID.randomUUID().toString());
        account.setPassword(preparePassword(account.getPassword()));

        return account;
    }
```

```
        private String preparePassword(String password) {

            String encodedPrefix = passwordConfiguration.getEncodedPrefix()
+ ":";

            return password.startsWith(encodedPrefix) ?
password.replaceFirst(encodedPrefix, "") :
passwordConfiguration.getPasswordEncoder().encode(password);
        }
    }
```

Лістинг 5.18:

```
    package com.foryouinnet.accountservice.models.entities;

    import lombok.AllArgsConstructor;
    import lombok.Data;
    import lombok.NoArgsConstructor;
    import lombok.experimental.SuperBuilder;
    import org.hibernate.annotations.CreationTimestamp;

    import javax.persistence.Column;
    import javax.persistence.Entity;
    import javax.persistence.GeneratedValue;
    import javax.persistence.GenerationType;
    import javax.persistence.Id;
    import javax.persistence.Table;
    import javax.persistence.Version;
    import java.time.LocalDateTime;

    @Data
    @Table(name = "accounts")
    @Entity
    @SuperBuilder
    @NoArgsConstructor
    @AllArgsConstructor
    public class Account {

        @Id
        @Column(name = "id")
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long id;

        @Column(name = "uuid", unique = true, updatable = false, nullable =
false)
        private String uuid;

        @Column(name = "email", unique = true, nullable = false)
        private String email;

        @Column(name = "username", unique = true, nullable = false)
```

```
        private String username;

        @Column(name = "password", nullable = false)
        private String password;

        @CreationTimestamp
        @Column(name = "created_at", updatable = false, nullable = false)
        private LocalDateTime createdAt;

        @Version
        @Column(name = "version", nullable = false)
        private long version;
    }
```

Лістинг 5.19:

```
    package com.foryouinnet.accountservice.models.dtos;

    import lombok.AllArgsConstructor;
    import lombok.Data;
    import lombok.NoArgsConstructor;
    import lombok.experimental.SuperBuilder;

    import javax.validation.constraints.Email;
    import javax.validation.constraints.Pattern;
    import java.util.Set;

    @Data
    @AllArgsConstructor
    public class AccountDto {

        @Pattern(regexp = "^[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-
[0-9a-f]{12}$", message = "ID is incorrect")
        private String uuid;

        @Email(regexp = "(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-
.]+$)", message = "Email is incorrect", groups = {})
        private String email;

        @Pattern(regexp = "^(?=[a-zA-
Z._]{8,20}$)(?!.*[_.]{2})[^_.].*[^_.]$", message = "Username is incorrect",
groups = {})
        private String username;

        @Pattern(regexp = "^(?=.*[a-z])(?=.*[A-
Z])(?=.*\\d)(?=.*[@$!%*?&])[A-Za-z\\d@$!%*?&]{8,10}$", message = "Password is
incorrect", groups = {})
        private String password;

        private Set<AuthorityDto> authorities;
    }
```

ДОДАТОК Д СЕРВІС ОБРОБКИ МЕДІА КОНТЕНТУ

Лістинг 5.20:

```
package com.foritinnet.moviestreamingservice.controller.impl;

import com.foritinnet.moviestreamingservice.controller.MovieController;
import com.foritinnet.moviestreamingservice.service.MovieService;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
import lombok.extern.log4j.Log4j2;
import org.springframework.core.io.buffer.DataBuffer;
import org.springframework.http.server.reactive.ServerHttpRequest;
import org.springframework.http.server.reactive.ServerHttpResponse;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;

import java.util.concurrent.atomic.AtomicInteger;

@Log4j2
@RestController
@RequiredArgsConstructor
public class MovieControllerImpl implements MovieController {

    private final MovieService movieService;

    @Override
    public Flux<DataBuffer> getVideo(@NonNull Integer videoId, @NonNull
String language, @NonNull ServerHttpRequest request, @NonNull
ServerHttpResponse response) {

        log.info("Request video {}", videoId);

        AtomicInteger count = new AtomicInteger(0);

        return movieService.getVideo(videoId, language, request,
response)
                .doOnNext(dataBuffer ->
count.addAndGet(dataBuffer.readableByteCount()))
                .doOnComplete(() -> System.out.println("Completed with "
+ count.get()));
    }
}
```

Лістинг 5.21:

```
package com.foritinnet.moviestreamingservice.service.impl;

import com.foritinnet.moviestreamingservice.model.TimeRange;
import com.foritinnet.moviestreamingservice.service.AudioService;
import com.foritinnet.moviestreamingservice.service.MovieService;
import com.foritinnet.moviestreamingservice.service.VideoService;
import com.foritinnet.moviestreamingservice.util.RangeUtil;
```

```
import io.netty.buffer.ByteBufAllocator;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
import lombok.SneakyThrows;
import org.bytedeco.ffmpeg.global.avcodec;
import org.bytedeco.ffmpeg.global.avutil;
import org.bytedeco.javacv.FFmpegFrameGrabber;
import org.bytedeco.javacv.FFmpegFrameRecorder;
import org.bytedeco.javacv.FFmpegLogCallback;
import org.bytedeco.javacv.Frame;
import org.bytedeco.javacv.Java2DFrameConverter;
import org.bytedeco.javacv.SeekableByteArrayOutputStream;
import org.springframework.core.io.buffer.DataBuffer;
import org.springframework.core.io.buffer.DataBufferFactory;
import org.springframework.core.io.buffer.DataBufferUtils;
import org.springframework.core.io.buffer.NettyDataBufferFactory;
import org.springframework.http.server.reactive.ServerHttpRequest;
import org.springframework.http.server.reactive.ServerHttpResponse;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Flux;

import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.BufferedInputStream;
import java.io.ByteArrayInputStream;
import java.io.InputStream;

@Service
@RequiredArgsConstructor
public class MovieServiceImpl implements MovieService {

    private final VideoService videoService;
    private final AudioService audioService;

    @Override
    public Flux<DataBuffer> getVideo(@NonNull Integer videoId, @NonNull
String language, @NonNull ServerHttpRequest request, @NonNull
ServerHttpResponse response) {

        TimeRange timeRange =
RangeUtil.extractRange(request.getHeaders());

        return audioService.getAudio(videoId, language, timeRange)
                .zipWith(videoService.getVideo(videoId, timeRange))
                .flatMapMany(tuple ->
concatenateVideoWithAudio(tuple.getT2(), tuple.getT1()));
    }

    @SneakyThrows
    private Flux<DataBuffer> concatenateVideoWithAudio(@NonNull
InputStream videoStream, @NonNull InputStream audioStream) {

        FFmpegLogCallback.set();
```

```
            FFmpegFrameGrabber videoGrabber = new
FFmpegFrameGrabber(videoStream);
            videoGrabber.setFormat("mp4");
            videoGrabber.start();

            FFmpegFrameGrabber audioGrabber = new
FFmpegFrameGrabber(audioStream);
            audioGrabber.setFormat("mp3");


audioGrabber.setAudioFrameNumber(videoGrabber.getLengthInVideoFrames());

            audioGrabber.start();

            SeekableByteArrayOutputStream seekableByteArrayOutputStream =
new SeekableByteArrayOutputStream();
            FFmpegFrameRecorder recorder = new
FFmpegFrameRecorder(seekableByteArrayOutputStream,
videoGrabber.getImageWidth(), videoGrabber.getImageHeight(),
audioGrabber.getAudioChannels());
            recorder.setVideoCodec(avcodec.AV_CODEC_ID_H264);
            recorder.setAudioCodec(avcodec.AV_CODEC_ID_MP3);
            recorder.setFormat("mp4");
            recorder.setFrameRate(videoGrabber.getFrameRate());

            recorder.setVideoBitrate(videoGrabber.getVideoBitrate());
            recorder.setAudioBitrate(audioGrabber.getAudioBitrate());
            recorder.start();

            int videoWithNull = 0;
            int audioWithNull = 0;
            int videoFrames = 0;
            int audioFrames = 0;
            Frame videoFrame, audioFrame;
            while (true) {

                videoFrame = videoGrabber.grab();
                audioFrame = audioGrabber.grab();

                if(videoFrame == null) {
                    videoWithNull++;
                }
                if(audioFrame == null) {
                    audioWithNull++;
                }

                if(videoFrame == null && audioFrame == null) {
                    break;
                }

                if(videoFrame != null) {
                    videoFrames++;
                    recorder.record(videoFrame);
                }
```

```
                if(audioFrame != null) {
                    audioFrames++;
                    recorder.recordSamples(audioFrame.samples);
                }
            }

            System.out.println("Video with null: " + videoWithNull);
            System.out.println("Audio with null: " + audioWithNull);

            System.out.println("Total video frames: " + videoFrames);
            System.out.println("Total audio frames: " + audioFrames);

            recorder.stop();
            recorder.close();
            recorder.release();

            videoGrabber.stop();
            videoGrabber.close();
            videoGrabber.release();

            audioGrabber.stop();
            audioGrabber.close();
            audioGrabber.release();

            return DataBufferUtils.readInputStream(() -> new
ByteArrayInputStream(seekableByteArrayOutputStream.toByteArray()), new
NettyDataBufferFactory(ByteBufAllocator.DEFAULT), 4096);
        }
    }
```

Лістинг 5.22:

```
package com.foritinnet.moviestreamingservice.provider.impl;

import com.foritinnet.moviestreamingservice.model.ByteRange;
import com.foritinnet.moviestreamingservice.model.TimeRange;
import com.foritinnet.moviestreamingservice.model.entity.VideoMetadata;
import com.foritinnet.moviestreamingservice.provider.VideoProvider;
import io.netty.buffer.ByteBufAllocator;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
import org.springframework.core.io.buffer.DataBufferUtils;
import org.springframework.core.io.buffer.NettyDataBufferFactory;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import software.amazon.awssdk.core.ResponseBytes;
import software.amazon.awssdk.core.async.AsyncResponseTransformer;
import software.amazon.awssdk.services.s3.S3AsyncClient;
import software.amazon.awssdk.services.s3.model.GetObjectRequest;

import java.io.InputStream;
```

```
@Service
@RequiredArgsConstructor
public class AwsVideoProvider implements VideoProvider {

    private final S3AsyncClient s3AsyncClient;

    private final String BUCKET_NAME = "language-with-me-films-bucket";
    private final String MOVIE_FOLDER = "video";

    @Override
    public Mono<InputStream> getVideo(@NonNull VideoMetadata
videoMetadata, @NonNull ByteRange byteRange) {

        return
Mono.fromFuture(s3AsyncClient.getObject(generateGetObjectRequest(videoMetadat
a, byteRange), AsyncResponseTransformer.toBytes()))
                .map(ResponseBytes::asInputStream);
    }

    private GetObjectRequest generateGetObjectRequest(@NonNull
VideoMetadata videoMetadata, @NonNull ByteRange byteRange) {

        return GetObjectRequest.builder()
                .bucket(BUCKET_NAME)
                .range(byteRange.toString())
                .key(String.format("%s/%s.%s", MOVIE_FOLDER,
videoMetadata.getId(), videoMetadata.getFormat()))
                .build();
    }
}
```

Лістинг 5.23:

```
package com.foritinnet.moviestreamingservice.provider.impl;

import com.foritinnet.moviestreamingservice.model.ByteRange;
import com.foritinnet.moviestreamingservice.model.entity.AudioMetadata;
import com.foritinnet.moviestreamingservice.model.entity.VideoMetadata;
import com.foritinnet.moviestreamingservice.provider.AudioProvider;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
import software.amazon.awssdk.core.ResponseBytes;
import software.amazon.awssdk.core.async.AsyncResponseTransformer;
import software.amazon.awssdk.services.s3.S3AsyncClient;
import software.amazon.awssdk.services.s3.model.GetObjectRequest;

import java.io.InputStream;

@Service
```

```
    @RequiredArgsConstructor
    public class AwsAudioProvider implements AudioProvider {

        private final S3AsyncClient s3AsyncClient;

        private final String BUCKET_NAME = "language-with-me-films-bucket";
        private final String MOVIE_FOLDER = "audio";

        @Override
        public Mono<InputStream> getAudio(@NonNull AudioMetadata
audioMetadata, @NonNull ByteRange byteRange) {

            return
Mono.fromFuture(s3AsyncClient.getObject(generateGetObjectRequest(audioMetadat
a, byteRange), AsyncResponseTransformer.toBytes()))
                    .map(ResponseBytes::asInputStream);
        }

        private GetObjectRequest generateGetObjectRequest(@NonNull
AudioMetadata audioMetadata, @NonNull ByteRange byteRange) {

            return GetObjectRequest.builder()
                    .bucket(BUCKET_NAME)
                    .range(byteRange.toString())
                    .key(String.format("%s/%s.%s", MOVIE_FOLDER,
audioMetadata.getId(), audioMetadata.getFormat()))
                    .build();
        }
    }
```