Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Факультет комп'ютерних наук
Кафедра теоретичної та прикладної системотехніки

«Затверджую»
Зав. кафедри теоретичної та
прикладної системотехніки
_____ д.т.н., проф. С. І. Шматков
«___» _____ 2024 р

Пояснювальна записка

до кваліфікаційної роботи
бакалавра

на тему: **«Система автоматизації розробки програмного забезпечення на основі використання архітектурних підходів»**

Захищено на засіданні
Атестаційної комісії № 44
протокол № __ від __.06.2024 р.
Оцінка _____ / _____
Голова Атестаційної комісії
_____ Скоб Ю. О.
(підпис) (прізвище та ініціали)

Виконав:
студент 4 курсу, групи КУ– 41
Галузь знань: 15 – Автоматизація та приладобудування
Спеціальність: 151 – «Автоматизація та комп'ютерно-інтегровані технології»
Чиняков Данило Сергійович_____
(прізвище, ім'я та по батькові) (підпис)

Керівник:
Доцент ЗВО, PhD, ст. викл. кафедри теоретичної та прикладної системотехніки
Мороз Ольга Юріївна _____
(прізвище, ім'я та по батькові) (підпис)

Рецензент:
Доцент ЗВО, зав.каф. електроніки та управляючих систем, к.ф.-м.наук, доцент
Хруслов Максим Михайлович _____
(підпис)

Харків – 2024

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи бакалавра складається зі вступу, 3 розділів, висновків, списку використаних джерел і 5 додатків. Загальний обсяг роботи складає 72 сторінки, із яких 45 сторінок основної частини з 33 рисунками, 12 найменуваннями списку використаних джерел та 5 додатками.

Метою кваліфікаційної роботи є забезпечення обґрунтованого вибору архітектурного підходу розробки програмного забезпечення.

Об'єкт дослідження – процес розробки програмного забезпечення на основі використання архітектурних підходів.

Предмет дослідження – система автоматизації розробки програмного забезпечення на основі використання монолітної, мікросервісної та Cloud-Native архітектур.

Проблема, що вирішується в кваліфікаційній роботі, полягає у застосуванні сучасних підходів та інструментів для вибору архітектурного підходу в розробці програмного забезпечення.

Основне завдання кваліфікаційної роботи полягає в оцінці швидкості розробки, продуктивності, використання пам'яті та зручності розробки для трьох різних архітектур: монолітної, на основі фреймворку Spring Boot, мікросервісної, на основі фреймворку Quarkus, та Cloud Native архітектури , з допомогою Quarkus native compilation та контейнеризації задля вибору підходу для конкретної системи. Ключові аспекти, що досліджуються включають швидкість розробки, продуктивність та ресурсовитрати, зручність розробки, а також об'єм знань по напрямку.

Ключові слова: *автоматизація, монолітна архітектура, мікросервісна архітектура, нативна компіляція, Java, Quarkus, Spring Boot.*

ABSTRACT

The explanatory note to the bachelor's thesis consists of an introduction, 3 chapters, conclusions, a list of references and 4 appendices. The total volume of the work is 72 pages, including 45 pages of the main part with 33 figures, 12 references and 5 appendices.

The purpose of the qualification work is to provide a reasonable choice of an architectural approach to software development.

The object of research is the process of software development based on the use of architectural approaches.

Subject of research – software development automation systems based on the use of monolithic, microservice and Cloud-Native architectures.

The problem solved in the qualification work is to apply modern approaches and tools for choosing an architectural approach to software development.

The main task is to evaluate the development speed, performance, memory usage and usability of three different architectures: monolithic, based on the Spring Boot framework, microservice, based on the Quarkus framework, and Cloud Native architecture, using Quarkus native compilation and containerization to select the approach for a particular system. The key aspects under study include development speed, performance and resource consumption, ease of development, and the amount of knowledge in the field.

Keywords: ***automation, monolithic architecture, microservice architecture, native compilation, Java, Quarkus, Spring Boot.***

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ І УМОВНИХ ПОЗНАЧЕНЬ

HTTP	–	Hypertext Transfer Protocol.
Фреймворк (Framework)	–	Платформа або основа для розробки програмного забезпечення, яка надає структуру, готові компоненти та інструменти для спрощення процесу розробки. Він забезпечує набір загальних функцій, які можна використовувати для розробки програм у межах певної області або для вирішення конкретних завдань.
Java	–	Об'єктно-орієнтована, мультиплатформна мова програмування, яка була розроблена компанією Sun Microsystems у 1995 році.
Native (Нативний)	–	Процес компіляції програмного коду безпосередньо до машинного коду, який розуміє конкретний апаратний пристрій або операційну систему
API (Application Programming Interface)	–	Набір правил та протоколів, які визначають спосіб, яким програми або компоненти програмного забезпечення можуть взаємодіяти між собою. API визначає набір функцій, методів, структур даних та конвенцій, які використовуються для спілкування між різними програмами або сервісами.
CI (Continuous integration)	–	Практика розробки програмного забезпечення, яка полягає в автоматизованій інтеграції коду розробників у спільний репозиторій частіше, ніж традиційні методи розробки. Основна ідея CI – це інтегрувати зміни в коді регулярно (зазвичай кілька разів на день), після чого вони автоматично тестуються за допомогою автоматизованих тестів.
CD (Continuous delivery)	–	Практика розробки програмного забезпечення, яка передбачає автоматизовану та повторювану поставку виправлень, нового функціоналу та інших змін у продукт з метою максимізації швидкості та надійності цього процесу.

ВСТУП

В сучасному світі інформаційних технологій ефективність розробки, тестування та супроводження програмного забезпечення має критичне значення для успіху будь-якої компанії. З розвитком технологій, архітектурні підходи до побудови програмних систем також зазнали значних змін. На сьогоднішній день три основні архітектурні підходи, які найчастіше використовуються, це монолітна архітектура, мікросервісна архітектура та Cloud-Native архітектура.

Монолітна архітектура була домінуючим підходом впродовж багатьох років. Вона передбачає створення єдиного цілісного додатку, де всі компоненти інтегровані в одну кодову базу. Цей підхід спрощує розробку та розгортання, але має свої недоліки, такі як складність масштабування та підтримки.

Мікросервісна архітектура з'явилася як відповідь на обмеження монолітного підходу. Вона базується на розділенні додатку на незалежні сервіси, кожен з яких виконує свою окрему функцію і може розгортатися незалежно від інших. Це дозволяє досягти більшої гнучкості, масштабованості та спрощує підтримку, проте додає складності в управлінні та оркестрації сервісів.

Cloud-Native архітектура використовує переваги хмарних технологій, що дозволяє створювати системи, які легко масштабуються, є стійкими до відмов і можуть швидко адаптуватися до змін. Використання хмарних платформ та контейнеризації, таких як Docker та Kubernetes, дозволяє розробникам створювати і розгортати додатки з високою ефективністю.

Одним з ключових аспектів сучасної розробки є нативна компіляція, яка дозволяє значно підвищити продуктивність додатків за рахунок оптимізації їх виконання на конкретних апаратних платформах. Це особливо актуально для високонавантажених систем, де кожен цикл процесора має значення.

Актуальність роботи полягає в використанні технологій, таких як монолітна архітектура, мікросервісна архітектура та Cloud-Native архітектура з використанням нативної компіляції що робить обґрунтованим процес розробки, тестування, веріфікації та супроводження програмного забезпечення.

Кваліфікаційна робота спрямована на глибокий аналіз трьох основних архітектурних підходів: монолітної, мікросервісної та Cloud Native. Зростання обсягів даних, потреба у швидкій реакції на зміни та необхідність масштабованості вимагають не тільки вибору відповідного варіанту архітектури, але й ретельного розуміння його впливу на усі аспекти розробки та експлуатації програмних рішень. Акцент на вивченні їхнього впливу на швидкість розробки, продуктивність, ресурсовитрати та зручність розробки має велике значення для тих, хто приймає стратегічні рішення в галузі автоматизації. Актуальність даної проблематики виходить далеко за межі лише технічних розглядів, оскільки вона напряму впливає на ефективність бізнес-процесів, масштабованість, стабільність та інші ключові аспекти розробки програмного забезпечення.

З врахуванням стрімкого розвитку технологій та підвищення конкуренції на ринку програмного забезпечення, визначення підходящого архітектурного підходу до розробки програмного забезпечення стає завданням критичного значення для компаній та розробників. У роботі детально розглянуто сутності кожної з обраних архітектур, а також їхні переваги та обмеження, що забезпечує підстави для обґрунтованого вибору в конкретних умовах розробки.

Метою дослідження роботи є забезпечення обґрунтованого вибору архітектурного підходу розробки програмного забезпечення.

Об'єкт дослідження – процес розробки програмного забезпечення на основі використання архітектурних підходів.

Методи дослідження – аналіз літературнхи джерел щодо монолітної архітектури, мікросервісної архітектури та Cloud-Native архітектури,

зіставлення результатів експерементів з відомими практиками впроваджень програмних модулів та проводення дослідження їх параметрів та методи системного аналізу.

Предмет дослідження – система автоматизації розробки програмного забезпечення на основі використання монолітної, мікросервісної та Cloud-Native архітектур.

Завдання дослідження

1. Виконати аналіз проблеми вибору архітектурного підходу до розробки програмного забезпечення, проаналізувавши популярні існуючі архітектурні підходи.
2. Розглянути способи реалізації даних методик, та визначення характеристик для порівняння.
3. Розробити та реалізувати модулі для автоматизації внутрішнього обліку персоналу та обробки онлайн-замовлень використовуючи підходи монолітної архітектури, мікросервісної архітектури та Cloud-Native архітектури.
4. Виконати тестування кожного з підходів та проаналізувати отримані результати.
5. Створити модель системи автоматизації розробки програмного забезпечення на основі використання архітектурних підходів, зробити аналіз.

РОЗДІЛ 1
АНАЛІЗ АРХІТЕКТУР РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Архітектура программного забезпечення

Архітектура програмного забезпечення – це інфраструктура програмного забезпечення, у межах якої можна вказати, розгорнути та виконати компоненти програми, що забезпечують функціональність користувача [1].

Вона визначає ключові аспекти системи, такі як структура, модульність, розподіленість, архітектурні стилі та принципи.

Основна мета архітектури програмного забезпечення – забезпечити розробку додатку, який буде легким у розумінні, модифікації та підтримці, а також ефективно вирішувати поставлені завдання. Це включає в себе вибір правильних технічних рішень, створення структури програми заради забезпечення гнучкості для майбутніх змін.

Архітектура не тільки визначає загальну стратегію розробки, а й сприяє зручності управління проєктом та забезпечує стабільність та продуктивність програмного продукту на протязі його життєвого циклу.

Автоматизація є актуальним завданням у сучасному світі, проте вибір правильної архітектури для реалізації автоматизаційних рішень вимагає обстеження і вирішення ряду складних проблем.

У роботі буде детально розглянуто ключові проблеми. Першою є складність бізнес-процесів та бізнес-логіки у різних предметних областях. Однією з основних складнощів при автоматизації є різноманітність процесів та логіки, що виникають у різних предметних сферах. У різних галузях може існувати велика кількість взаємодіючих елементів, таких як замовлення, облік товарів, управління персоналом, і ці елементи можуть варіюватися від невеликих до масштабних підприємств. Поєднання такої різноманітності бізнес-процесів уніфікованою системою може стати викликом, оскільки

необхідно забезпечити гнучкість та адаптованість до конкретних вимог кожної з них.

З іншого боку, велика кількість процесів може призвести до переплутаних логік та важкодоступних структур, що ускладнює підтримку та розвиток системи. Вибір архітектури, яка може ефективно взаємодіяти з різними аспектами систем, стає важливою задачею для подолання цієї складності.

Ефективна масштабованість та висока продуктивність є важливими характеристиками для будь-якої системи автоматизації. Здатність системи ефективно пристосовуватися до зростання обсягів операцій та навантаження є ключовою для забезпечення плавної роботи у великих мережах або під час пікових навантажень, таких як святкові заходи чи акції.

Розглядаючи масштабування, виділяють два типи: вертикальне масштабування та горизонтальне масштабування.

Рисунок 1.1 – Вертикальне масштабування.

Вертикальне масштабування включає збільшення ресурсів чи потужності існуючого обладнання для одного сервера або компонента.

Зазвичай здійснюється покращенням апаратного забезпечення, таким як заміна процесора чи додавання пам'яті, він відносно простий у реалізації,

зокрема, коли система вже працює, але може бути обмеженим можливостями апаратного забезпечення та може призводити до підвищення вартості масштабування.

Рисунок 1.2 – Горизонтальне масштабування.

Мікросервісна архітектура може виявитися більш витратною на етапі розробки через необхідність створення та підтримки набору невеликих, але автономних сервісів. Однак цей підхід може принести вигоди у плані масштабованості та гнучкості, що знижує загальну вартість витрат на підтримку у випадку зміни вимог або розширення бізнесу.

У випадку Cloud-Native архітектури, вартість може включати витрати на використання хмарних обчислювальних ресурсів, а також на підтримку та моніторинг інфраструктури. Тим не менш, вигоди від автоматизації та можливості миттєво масштабувати ресурси можуть виявитися важливими для ефективності та надійності системи.

Загалом, важливо враховувати повну вартість власності, включаючи як витрати на впровадження, так і подальші витрати на підтримку та адаптацію системи до змін у бізнес-середовищі. Крім того, вибір архітектурного підходу повинен враховувати не лише фінансові аспекти, але й стратегічні цілі та потреби конкретного підприємства. Вибір архітектури повинен бути економічно обгрунтованим. Розробка та підтримка автоматизованої системи повинні бути доступними для бізнесу, а витрати повинні бути в межах бюджету.

З точки зору розробників один із ключових аспектів є зручність розробки та подальшого супроводу системи.

Монолітна архітектура, завдяки її єдиноцільності та компактності, може спрощувати процес розробки, оскільки усі компоненти знаходяться в одному кодовому базисі. Це полегшує відлагодження та внесення змін, але може стати складним у випадку великих, складних систем.

Мікросервісна архітектура, навпаки, надає більшу гнучкість та можливість розробки окремих сервісів незалежно один від одного. Це дозволяє розробникам працювати над окремими частинами системи, полегшуючи спрощення коду та швидше внесення змін. Однак, необхідно враховувати, що це може призвести до складнішого управління та вимагати високого рівня оркестрації для взаємодії між сервісами.

Архітектурний підхід Cloud-Native заснований на мікросервісах та використанні хмарних технологій. Він дозволяє розробникам використовувати віртуалізацію та контейнеризацію для швидкої розгортання та масштабування сервісів. В такому підході зручність розробки поєднується з високою гнучкістю масштабування та управління інфраструктурою, але може вимагати додаткового навчання для впровадження.

Однак, вибір архітектурного підходу повинен враховувати не лише технічні аспекти, але і наявність кваліфікованого персоналу та його здатність до розробки та супроводу обраної архітектури.

Важливо забезпечити, щоб обрана система відповідала не лише потребам бізнесу, але й ресурсам та компетенціям розробницького колективу.

1.2 Різновиди архітектурних підходів при розробці програмного забезпечення

Архітектурна еволюція програмних додатків подолала великий шлях, рухаючись від масивних монолітів до гнучких мікросервісів та іноваційної Cloud-Native архітектури. Кожен архітектурний підхід несе в собі унікальні переваги та виклики, вимагаючи обдуманого вибору, щоб ваш додаток відповідав конкретним вимогам та досягав поставлених цілей.

Монолітна архітектура (рис. 1.3).

До початку 2000-х років програмні додатки розроблялися як єдина масивна конструкція коду, де всі складові володіли ідентичними ресурсами та простором пам'яті.

Рис. 1.3 – Монолітна архітектура.

Цей формат архітектури відомий як монолітна, і його основними перевагами є простота та легкість у розробці. Зате, слід визнати, що вона також виявилася вразливою до недоліків, зокрема, у сфері масштабованості та підтримки. Незважаючи на реалізацію багатьох частин всього програмного забезпечення, додаток розгортається як одна єдина автономна програма [2].

Плюси:

– Простота і зрозумілість: Монолітна архітектура – це простий та легко зрозумілий підхід до створення додатків.

– Легкість у розробці і обслуговуванні: Код знаходиться в єдиному рішенні, що полегшує пошук та обслуговування.

Мінуси:

– Складність у підтримці: Підтримка може стати проблемою при зростанні розміру та складності додатку.

– Менша гнучкість: Менша адаптивність до змін порівняно з іншими архітектурами.

– Низький рівень безпеки: Потребує уваги до заходів безпеки для забезпечення адекватного захисту.

Загалом монолітна архітектура є відмінним вибором для простих додатків, які вимагають високої продуктивності.

Проте, зі зростанням розміру та складності, розгляд альтернативних архітектур, таких як мікросервіси, може бути важливим для подолання можливих обмежень монолітної структури.

Сервісно-орієнтована архітектура (рис. 1.4).

SOA (Service-Oriented Architecture) виникла на початку 2000-х років як інноваційний метод побудови розподілених систем, використовуючи веб-сервіси.

Основна ідея цієї архітектури полягає в розгляді додатку як набору менших, самостійних та перевикористовуємих сервісів, які можна ефективно комбінувати для створення складних програмних продуктів.

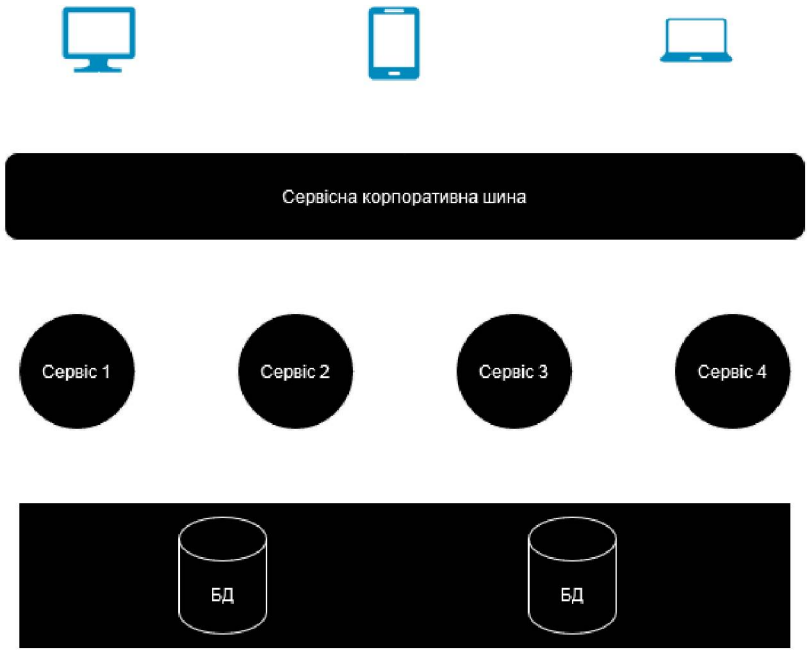


Рисунок 1.4 – Сервісно-орієнтована архітектура.

SOA відзначається покращеною масштабованістю, гнучкістю та можливістю повторного використання, але в той же час вона стикається із викликами, такими як складність та управління.

Плюси:

– Багаторазове Використання Сервісів: Система SOA дозволяє ефективно використовувати сервіси в різних контекстах та додатках.

– Інтероперабельність на Різних Платформах і Мовах: SOA забезпечує можливість взаємодії на різних платформах та мовах програмування, що розширює його застосування.

– Гнучкість та Адаптивність: Архітектура SOA дозволяє системі гнучко адаптуватися до змін у вимогах та оточенні.

Спрощене Виявлення та Споживання Сервісів: SOA спрощує процес виявлення та використання сервісів, полегшуючи їхню інтеграцію.

Мінуси:

– Проблеми Інтеграції: Виникають труднощі при інтеграції сервісів з різних джерел.

– Проблеми Безпеки: Потребує додаткових заходів для забезпечення безпеки, щоб уникнути несанкціонованого доступу.

– Накладні Витрати на швидкодію: Може призводити до збільшених витрат на обмін даними між сервісами, впливаючи на швидкодію.

– Підвищена Складність: Інтеграція SOA може стати складнішою завданням, особливо при великому обсязі сервісів.

Загалом, SOA представляє потужний підхід до розробки програмного забезпечення, проте вимагає уважного вирішення можливих викликів.

Мікросервісна архітектура.

Мікросервісна архітектура – це методологія розробки програмного забезпечення, яка розбиває додаток на невеликі та автономні сервіси, що взаємодіють між собою через API (рис. 1.5).

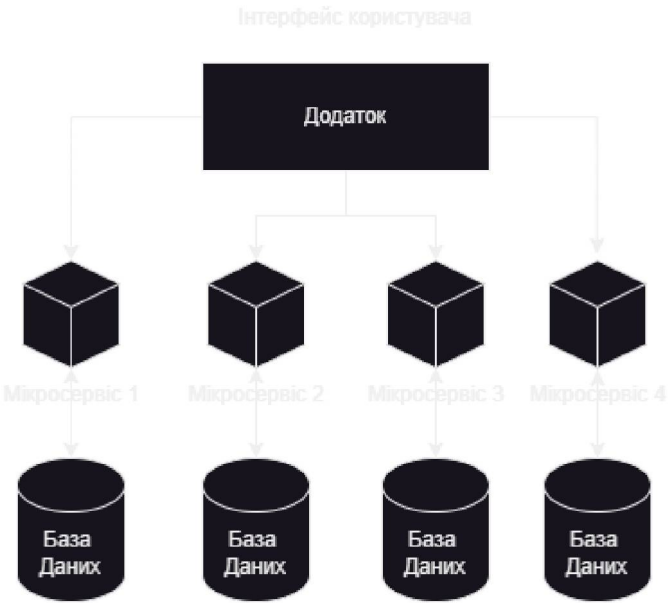


Рисунок 1.5 – Мікросервісна архітектура.

Цей архітектурний підхід виник на початку 2010-х як відповідь на виклики монолітних архітектур, спрощуючи розгортання та масштабування.

Плюси:

– Гнучкість та швидкість розгортання: Можливість незалежного розгортання та масштабування кожного сервісу.

– Легке управління: Кожен сервіс може керуватися окремо, спрощуючи управління великими системами.

– Відновлення та незалежність: Автономія сервісів дозволяє легко виявляти та виправляти помилки без впливу на інші частини системи.

Мінуси:

– Складність інтеграції: Ускладненість забезпечення взаємодії між різними сервісами.

– Велика кількість точок входу: Збільшення кількості компонентів може призвести до більшого обсягу точок входу та виходу для адміністрування та моніторингу.

Специфічні виклики масштабування: Деякі аспекти, такі як координація транзакцій та управління консистентністю, можуть стати складнішими в мікросервісних системах.

Cloud-Native архітектура – це підхід до розробки та впровадження програмного забезпечення, орієнтований на використання хмарних сервісів та ресурсів (рис. 1.6). Cloud native – це підхід до створення додатків як мікросервісів і запуску їх на контейнерних і динамічно організованих платформах, який повністю використовує переваги моделі хмарних обчислень. Хмарні технології – це про те, як, а не де додатки створюються та розгортаються [2].

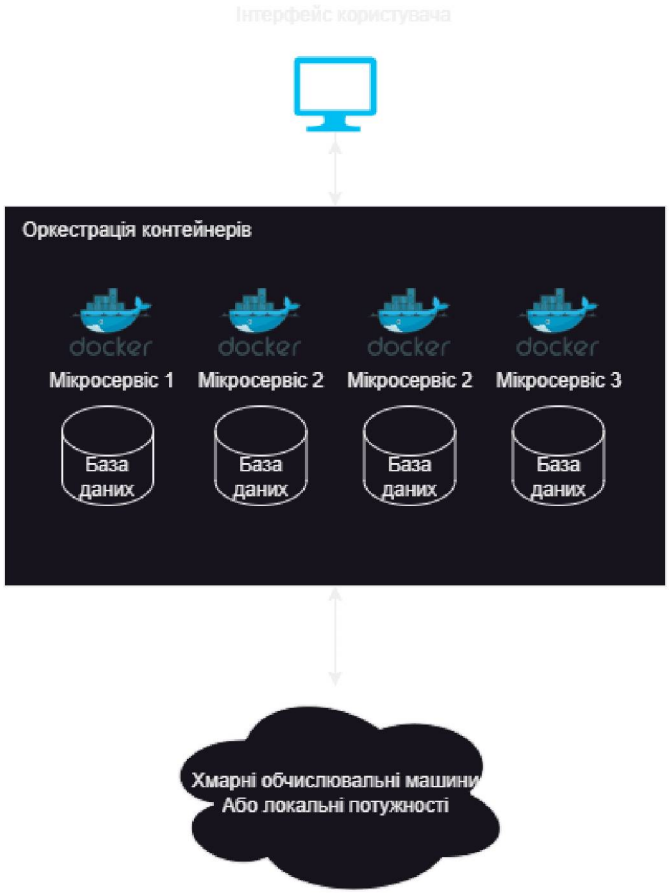


Рисунок 1.6 – Cloud-Native архітектура.

Цей підхід виник у відповідь на зростання популярності хмарних технологій та спрямований на підвищення швидкості розробки, масштабованості та ефективності використання інфраструктури.

Плюси:

Еластичність: Можливість швидко масштабувати і зменшувати інфраструктуру відповідно до потреб додатку.

Автоматизація: Застосування автоматизованих інструментів для керування та розгортанням додатків.

Швидкість розробки: Спрощена розробка та впровадження завдяки хмарним сервісам та інфраструктурі як коду.

Мінуси:

Навчання та адаптація: Вимагає нових знань та підготовки персоналу для роботи з хмарними технологіями.

Безпека: Збільшення ризиків стосовно безпеки даних та дотримання стандартів безпеки.

Залежність від хмарних провайдерів: Обмеженість у виборі хмарного провайдера та можливість слідкувати за змінами у їхніх послугах.

1.3 Методики впровадження обраних архітектурних підходів

Для наочної демеонстрації було обрано три модулі які будуть розроблені за допомогою трьох архітектурних підходів:

1. Модуль клієнтів
2. Модуль інвентаризації
3. Модуль замовлень

Модуль клієнтів буде відповідати за збереження, обробку, та представлення даних про клієнтів.

Модуль інвентаризації буде відповідати за збереження, обробку, та представлення даних про наявні товари та послуги.

Модуль онлайн замовлень буде комунікувати з модулем клієнтів та інвентаризації для отримання даних, він буде напряму звертатись до у пам’яті, при монолітній архітектурі, або ж за допомогою API запитів, при мікросервісному та Cloud-Native випадку.

Для монолітної архітектури (рис. 1.7) всі три модуля будуть знаходитись в одній кодовій базі та використовувати одну спільну базу даних, в нашому випадку PostgreSQL.

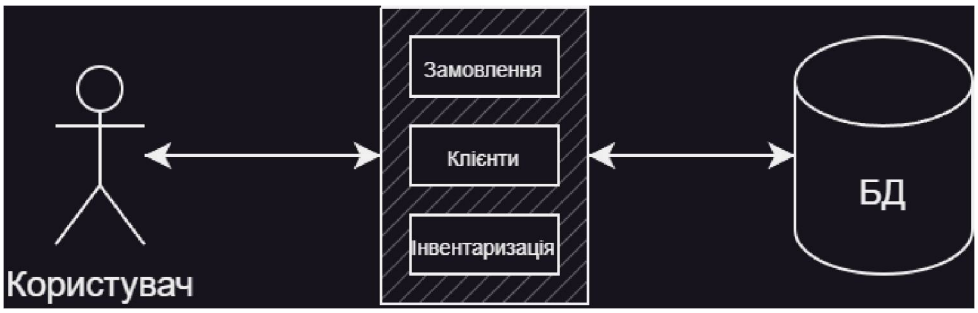


Рисунок 1.7 – Схема додатку використовуючи монолітну архітектуру.

Для мікросервісної архітектури (рис. 1.8) всі три модуля будуть знаходитись кожна в своїй кодовій базі та використовувати три окремих бази даних, в нашому випадку PostgreSQL.

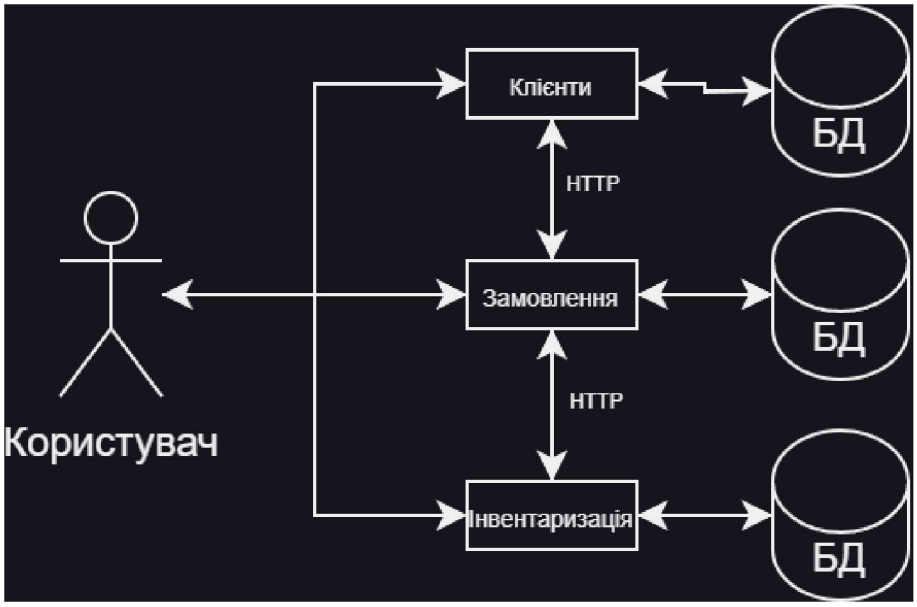


Рисунок 1.8 – Схема додатку використовуючи мікросервісну архітектуру.

Для Cloud-Native архітектури (рис. 1.9) всі три модуля будуть знаходитись кожна в своїй кодовій базі, великим плюсом є те, що можна перевикористати вже написані модулі мікросервісної архітектури, внесши лише деякі зміни. Використовувати будемо одну спільну базу даних, в нашому випадку PostgreSQL. Основна модифікація для цього архітектурного підходу: впровадження Docker-контейнерів та

Kubernetes для оркестрації контейнерів, балансування навантаження, та впровадження реплікацій за потреби.

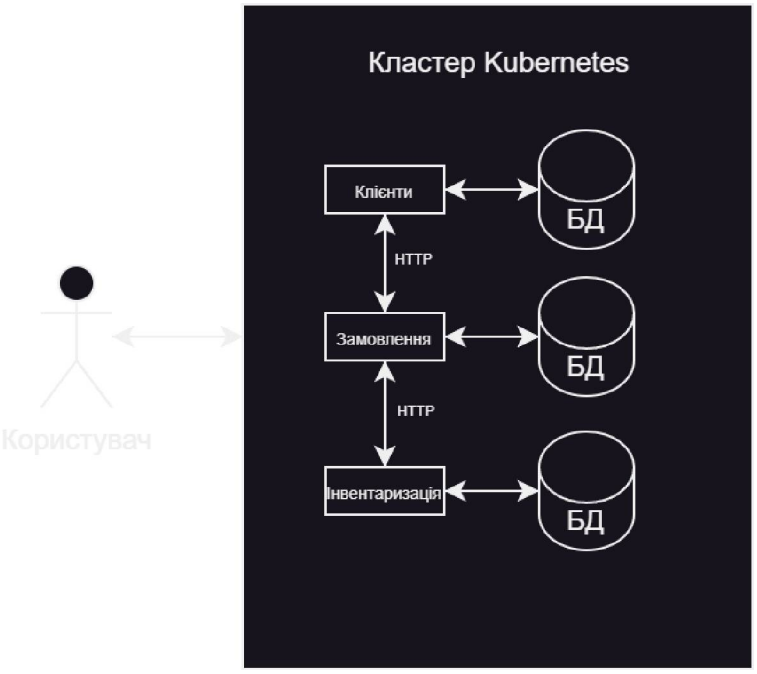


Рисунок 1.9 – Схема додатку використовуючи Cloud-Native архітектуру

1.4 Основні задачі дослідження

Для досягнення мети дослідження було обрано систему яка буде використоваувати архітектурні підходи для автоматизації.

Задачі включають:

1. Зробити аналіз проблеми вибору архітектурного підходу до розробки программного забезпечення:

2. Розглянути способи реалізації даних методик, та визначення характеристик для порівняння.

3. Розробити модулі для автоматизації обробки інвентаризації, покупців та обробки онлайн-замовлень використовуючи підходи монолітної архітектури, мікросервісної та Cloud-Native архітектури.

4. Проведення тестування та аналіз отриманих результатів.

5. Створення системи автоматизації на основі отриманих результатів.

Для монолітного додатку було обрано фреймворк Spring Boot, так як він став класичним для розробки на мові програмування Java.

Для мікросервісного додатку, який буде створений з трьох мікросервісів, був обраний Quarkus, який є більш новим фреймворком, та розроблений спеціально під мікросервісну архітектуру.

Для Cloud-Native архітектури було обрано можливість нативної компіляції Quarkus, і використання кластеру Kubernetes, що по-перше дозволить перевикористати вже написану кодову базу для мікросервісної архітектури, і по-друге зберігти найкращі практики Cloud-Native архітектури, де контейнерізація йде на першому місці.

Після розробки цих модулів буде проведено тестування за різними параметрами, які будуть включати:

- Час запуску
- Обсяг зайнятої оперативної пам’яті
- Кількість запитів які додаток може обробити
- Час на відповідь від серверу

На основі цих даних буде розроблена модель системи автоматизації розробки програмного забезпечення на основі використання архітектурних підходів, за параметрами які були отримані в розділі тестування.

Висновки за розділом 1

У цьому розділі були розглянуті ключові проблеми автоматизації підприємств, різновиди існуючих автоматизаційних систем та проведений поверхневий аналіз наявних технологічних рішень.

Було вирішено обрати три модулі для розробки, кожен з яких має свої особливості щодо навантаження та стабільності. Різниця між монолітною, мікросервісною та Cloud-Native архітектурою знаходить своє відображення в способах впровадження цих модулів. Монолітний підхід передбачає їх об'єднання в одну кодову базу та спільну базу даних, що підходить для стабільних навантажень. Мікросервісна архітектура передбачає розміщення

кожного модулю у власній кодовій базі, і кожний модуль буде мати свою базу даних. У Cloud-Native архітектурі кожен модуль також має власну кодову базу, як і окрему базу даних для кожного модулю, але використовується Docker-контейнеризація та Kubernetes для оркестрації контейнерів, що дозволяє ефективно масштабувати та керувати навантаженням.

Були визначені основні завдання дослідження, які включають аналіз проблеми вибору архітектурного підходу, розробку модулів для автоматизації, тестування та розробку системи автоматизації.

Для монолітного додатку був обраний Spring Boot, для мікросервісної – Quarkus, а для Cloud-Native – Quarkus з нативною компіляцією та Kubernetes.

Тестування буде проведено з огляду на різні параметри, такі як час запуску, обсяг зайнятої пам'яті, продуктивність та швидкість відповіді сервера. На основі результатів тестування буде розроблена система автоматизації, яка використовуватиме найкращі практики відповідно до обраного архітектурного підходу.

Ці кроки дозволять не лише реалізувати автоматизовану систему, а й обрати найбільш відповідний архітектурний підхід для конкретних потреб і навантажень підприємства.

РОЗДІЛ 2
РОЗРОБКА МОДУЛІВ НА ОСНОВІ ЯКИХ БУДЕ РОБИТИСЬ СИСТЕМА АВТОМАТИЗАЦІЇ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Монолітна архітектура передбачає створення єдиного цілісного додатку, де всі компоненти інтегровані в одну кодову базу. Цей підхід є простим у розробці та розгортанні, але має свої недоліки, такі як складність масштабування та підтримки. Використання Spring Framework значно спрощує розробку монолітних додатків завдяки своїм потужним інструментам та бібліотекам.

2.1 Розробка монолітного модуля з використанням Spring framework

Spring Boot – це фреймворк для розробки Java-додатків, який базується на фреймворку Spring. Він спрощує процес створення веб-додатків, мікросервісів та інших застосунків на основі Java. Одним із ключових принципів Spring Boot є конвенція над конфігурацією, що дозволяє розробникам швидше починати проєкт та уникнути багатої рутинної налаштування.

Spring Boot був випущений в 2014 році як ініціатива для полегшення розробки з використанням Spring Framework. Він виник як відповідь на потребу в простоті та ефективності при розробці Java-додатків, зокрема веб-додатків та мікросервісів.

Spring Boot набув величезної популярності в розробницькій спільноті завдяки своїй простоті, швидкості старту проєкту і низці вбудованих функціональностей. Велика кількість підтримки, постійні оновлення та активна спільнота розробників роблять Spring Boot одним із найвибраніших фреймворків для створення Java-додатків.

Інформації про Spring Boot доступно величезна кількість. Офіційна документація Spring Boot, яка включає в себе різні гіди та приклади, є відмінним ресурсом для новачків та досвідчених розробників. Крім того,

існують книги, онлайн-курси та блоги від спільноти, які допомагають вивчити та розширити знання про Spring Boot. Велика кількість питань та відповідей на них на форумах, таких як Stack Overflow, також робить можливим отримання допомоги вирішення конкретних проблем.

2.1.1 Створення проєкту

Створення проєкту Spring Boot з допомогою start.spring.io

Для початку роботи над монолітним проєктом Spring Boot потрібно створити початковий проєкт. Один з найшвидших та найзручніших способів це зробити – скористатися веб-інтерфейсом https://start.spring.io/.

Цей інструмент дозволяє створити каркас проєкту з необхідними налаштуваннями та залежностями всього за кілька кроків.

1. Відкрито веб-браузер на сторінці https://start.spring.io/.
2. Налаштування проєкту:
   - Вказано потрібні налаштування для проєкту Spring Boot:
     Обрано останню доступну версію Spring Boot.
     Встановлено мову програмування Java.
     Додано артефактне ім'я проєкту (наприклад, "restaurant-management").
3. Додано залежності:
   - В розділі "Dependencies" додано наступні залежності:
     1) Spring Web: Ця залежність дозволить створювати веб-додатки з використанням Spring MVC.
     2) Lombok: Lombok – це бібліотека, яка автоматизує створення методів доступу та інших шаблонних методів, спрощуючи розробку.
     3) Spring Data JPA: Ця залежність дозволить легко взаємодіяти з базою даних через JPA.
     4) PostgreSQL Driver: Ця залежність додає драйвер JDBC для взаємодії з базою даних PostgreSQL.

4. Генерація проєкту:

Після додавання залежностей натиснуто кнопку "Generate" для створення та завантаження архіву проєкту Spring Boot.

5. Розпакування проєкту:

Після завантаження архіву розпаковано його на комп'ютері.

6. Імпортовано проєкт у IDE:

Імпортовано розпакований проєкт у інтегроване середовище розробки IntelliJ IDEA.

Ці залежності дозволять швидко створити веб-додаток з використанням Spring Boot, спростять роботу з базою даних та зроблять наш код більш чистим та ефективним завдяки автоматичній генерації коду Lombok.

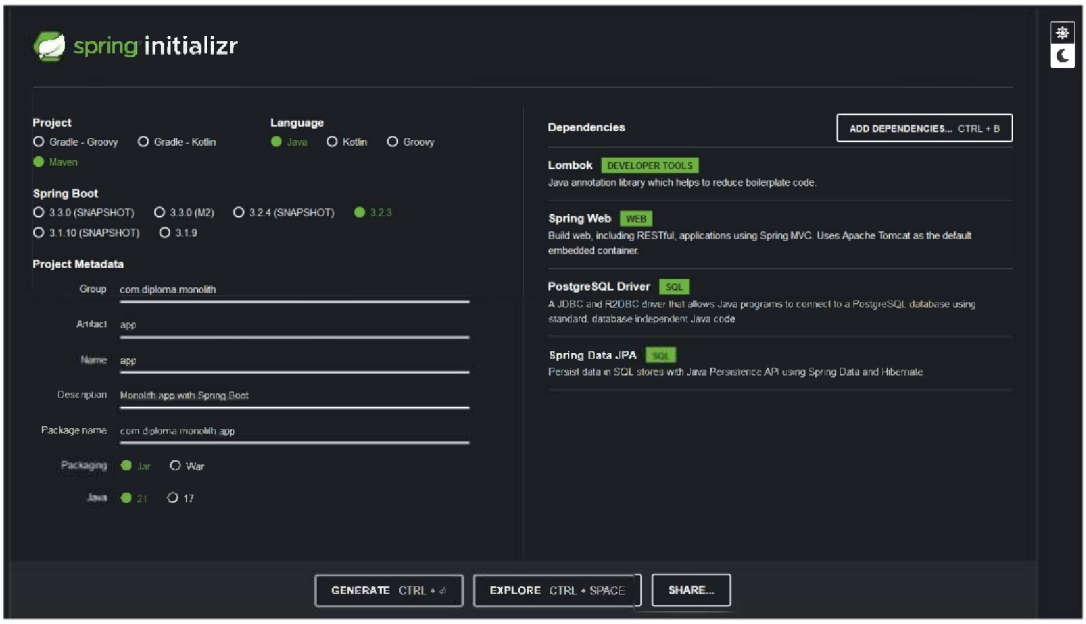


Рисунок 2.1 – Приклад створення проєкту використовуючи Spring initializr

2.1.2 Особливості впровадження монолітної архітектури

Так як Spring Boot було обрано для впровадження монолітної архітектури, то при написанні додатку будуть певні особливості які притаманні монолітній архітектури.

Єдина кодова база, це водночас великий плюс, і мінус. Це рішення дозволяє зберігати всю логіку в одному місці, полегшує пошук, і робить

розгортання додатку більш легким. В той ж час це веде до більшого часу запуску, неможливістю зміни окремого модуля без перекомпіляції усієї кодової бази, і це не дозволяє ефективно горизонтально масштабуватись.

Було вирішено розділити монолітний додаток на пакети, які будуть логічно відокремлювати класи, за їх призначенням. Також три модуля мають свій відповідний префікс – Customer, Item, Order, для замовника, предметів, та замовлень відповідно, це можна побачити на рис. 2.2

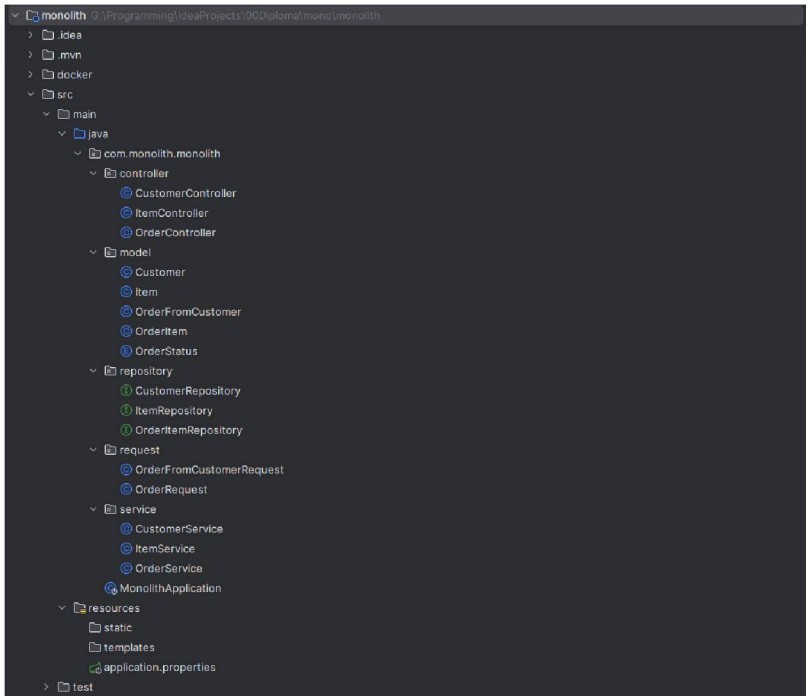


Рисунок 2.2 – Кодова база монолітного додатку на основі Spring Boot.

```
# Назва застосунку Spring
spring.application.name=monolith

# Вибірковий порт сервера, який може бути заданий для зміни порту за замовчуванням (8080).
server.port=8080

# Конфігурація URL-адреси джерела даних. Це адреса, по якій застосунок з'єднується з базою даних.
spring.datasource.url=jdbc:postgresql://localhost:5432/monolith

# Вказує Hibernate діалект для PostgreSQL
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

# Ім'я користувача для підключення до бази даних.
spring.datasource.username=user

# Пароль для підключення до бази даних.
spring.datasource.password=password

# Властивість Hibernate, яка контролює стратегію автоматичного створення, оновлення,
# перевірки або видалення схеми бази даних при старті додатку.
# Значення "create" автоматично створює нову схему бази даних, видаляючи попередні дані,
spring.jpa.hibernate.ddl-auto=create
```

Рисунок 2.3 – Файл конфігурації application.properties для Spring Boot.

Spring Boot використовує application.properties файл, в якому налаштовуються конфігурація додатку. Так як це монолітний додаток, то було використано одну Базу Даних, де зберігаються всі сутності, це видно з строки spring.datasource.url, ця конфігурація відповідає за JDBC посилання на базу даних. Усю конфігурацію та коментарі до кожного рядка можно побачити на рисунку 2.3

Для збереження інформації додаток використовує базу даних PostgreSQL, яка запускається у Docker-контейнері. Для того щоб досягнути цього треба налаштувати середовище виконання, та виконати файл docker-compose.yml (рис 2.4), написавши в термінал docker-compose up.

Для монолітного додатоку використовується одна база даних де зберігаються всі дані.

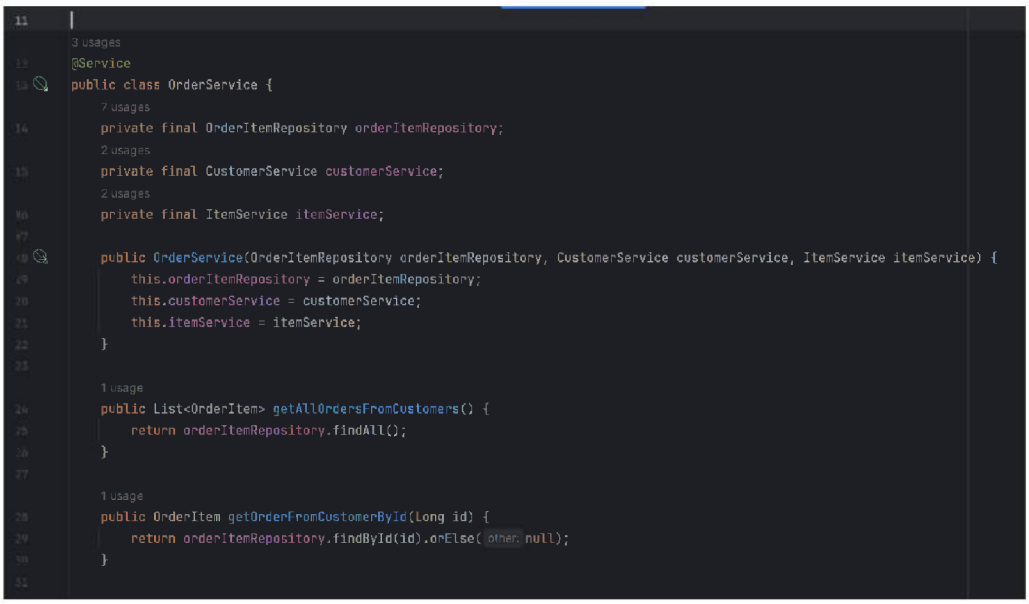
```yaml
version: "3.9"
services:
  postgres:
    image: postgres:16
    environment:
      POSTGRES_DB: "monolith"
      POSTGRES_USER: "user"
      POSTGRES_PASSWORD: "password"
    ports:
      - "5432:5432"
```

Рисунок 2.4 – Docker-compose файл для бази даних монолітного додатку.

Одна з особливостей монолітної архітектури – єдина кодова база.

Із-за цього спілкування та обмін даними відбувається в пам’яті, без використання мережі. Що швидше, і не потребує написання зайвих контроллерів, або класів для запитів.

Order модуль повинен отримати інформацію від Customer та Item модуля, для валідації інформації, що легко було досягнуто ін’єкцією відповіднитх сервісів (рисунок 2.5).



```
3 usages
@Service
public class OrderService {
    7 usages
    private final OrderItemRepository orderItemRepository;
    2 usages
    private final CustomerService customerService;
    2 usages
    private final ItemService itemService;

    public OrderService(OrderItemRepository orderItemRepository, CustomerService customerService, ItemService itemService) {
        this.orderItemRepository = orderItemRepository;
        this.customerService = customerService;
        this.itemService = itemService;
    }

    1 usage
    public List<OrderItem> getAllOrdersFromCustomers() {
        return orderItemRepository.findAll();
    }

    1 usage
    public OrderItem getOrderFromCustomerById(long id) {
        return orderItemRepository.findById(id).orElse( other: null);
    }
```

Рисунок 2.5 – Order сервіс монолітного додатку.

Що дозволяє напряму звертатись до функціоналу Customer та Item модулів, без необхідності дублювання коду або логіки.

2.2 Розробка мікросервісеного модуля з використанням Quarkus

Quarkus – це фреймворк для розробки Java-додатків, спеціально призначений для створення оптимізованих мікросервісів та Cloud-Native застосунків. Його особливість полягає в наданні низького часу старту та оптимізації використання ресурсів, що робить Quarkus ідеальним вибором для високопродуктивних ініціатив, таких як serverless та контейнеризація.

Quarkus був випущений у 2019 році компанією Red Hat. Його розробка була зорієнтована на вирішення завдань, пов'язаних із створенням ефективних та швидких Java-додатків для сучасних хмарних інфраструктур.

Quarkus набирає популярність, особливо серед розробників, які працюють з мікросервісною архітектурою та хмарними рішеннями. Його оптимізована природа, підтримка гарячого перезавантаження та інші характеристики роблять його привабливим для проєктів, де важливі такі аспекти, як швидкість та низький витрати ресурсів.

За кількістю доступної інформації Quarkus програє Spring та Spring Boot, так як це новий фреймворк. Не достає не тільки інформації по використанню,

а й інтеграції з іншими сервісами, модулями. Тому можуть бути ситуації коли самому потрібно буде писати рішення для підключення до сторонніх сервісів. Не зважаючи на це спільнота Quarkus активно ділиться своїм досвідом через блоги, вебінари та інші ресурси. За допомогою форумів та соціальних мереж також можна знайти відповіді на різні питання та отримати підтримку від спільноти.

2.2.1 Створення проєкту Quarkus

Створення проєкту Quarkus з використанням code.quarkus.io

Для початку роботи над мікросервісним додатков Quarkus потрібно створити початковий проєкт, це буде повторено три рази, з однаковими extension, але використовуючи різні назви пакетів та артефактне ім’я.

Один з найшвидших та найзручніших способів це зробити – скористатися веб-інтерфейсом https://code.quarkus.io/.

Цей інструмент дозволяє створити каркас проєкту з необхідними налаштуваннями та розширеннями всього за кілька кроків.

1. Відкрито веб-браузер на сторінці https://code.quarkus.io/: Відкрийте ваш веб-браузер і перейдіть на веб-сайт code.quarkus.io.
2. Налаштування проєкту:

Оберіть необхідні налаштування для проєкту Quarkus:

Виберіть останню доступну версію Quarkus.

Встановіть мову програмування Java.

Введіть артефактне ім'я проєкту.

3. Додано розширення:

У розділі "Extensions" додайте наступні розширення:

- REST Client Jackson
- REST Client
- Hibernate Reactive
- Reactive PostgreSQL client
- Container Image Jib
- Kubernetes

4. Генерація проєкту:

Після додавання розширень натисніть кнопку "Generate your application" для створення та завантаження архіву проєкту Quarkus.

5. Розпакування проєкту:

Після завантаження архіву розпакуйте його на вашому комп'ютері.

6. Імпорт проєкту у IDE:

Імпортуйте розпакований проєкт у ваше інтегроване середовище розробки, наприклад, IntelliJ IDEA.

Ці розширення додадуть необхідну функціональність для проєкту Quarkus, дозволяючи створити веб-додаток з підтримкою REST, реактивного доступу до бази даних та готовності для розгортання на Kubernetes.

Приклад Orders мікросервіса можна побачити на рисунку 2.6. Простий інтерфейс та простота налаштування дозволяють легко створити базовий проєкт та почати роботу над ним.

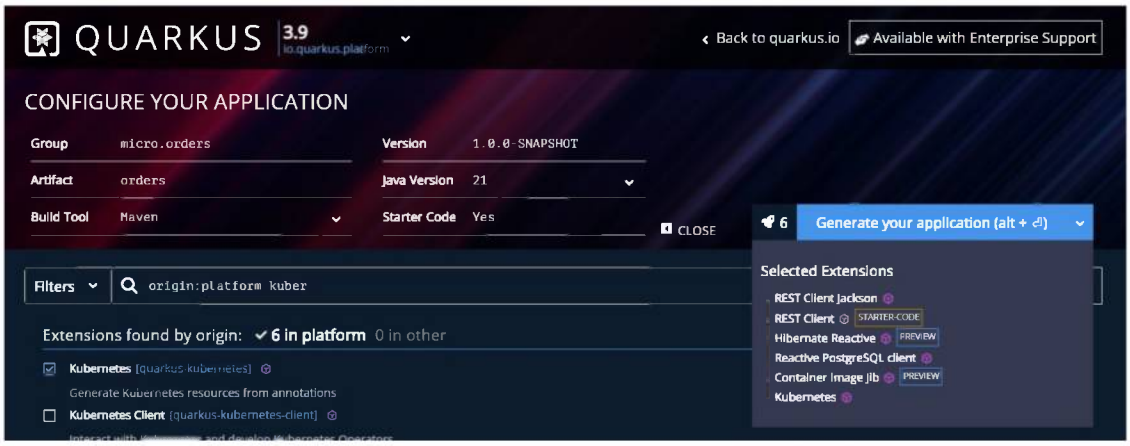


Рисунок 2.6 – Приклад створення проєкту використовуючи code.quarku.io

2.2.2 Особливості впровадження мікросервісної архітектури

Було вирішено розділити мікросервісний додаток на три мікросервіси, які будуть логічно відокремлюватись, за їх призначенням.

Ці три мікросервіса мають відповідну назву, до модуля якого вони відносяться – Customer, Item, Order, для замовника, предметів, та замовлень відповідно, це можна побачити на рисунку 2.7

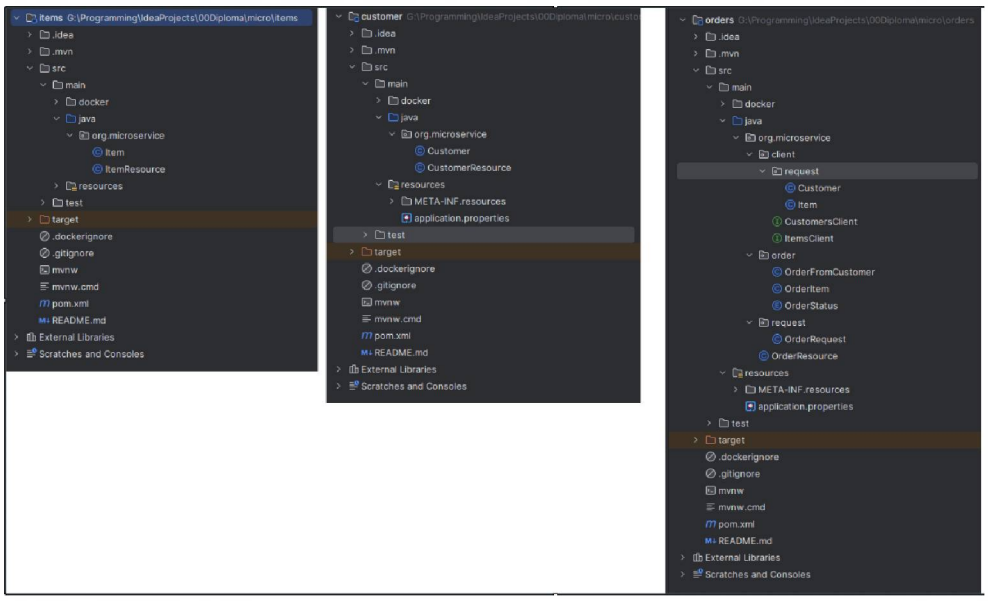


Рисунок 2.7 – Кодова база трьох мікросервісів.

З цих трьох мікросервісів найбільшим виявився orders мікросервіс, так як йому потрібно робити виклики до мікросервісів customer та items. Тому для виконання цих запитів йому потрібні відповідні класи для виконнання запитів:

Customer та Item класи для запитів, CustomersClient та ItemsClient це класи для спілкування Order мікросервісу з Items та Customer мікросервісами за допомогою HTTP запитів.

На прикладі CustomerClient можна побачити декларативній підхід до написання клієнта, де доводиться писати мінімум коду, треба зазначити шлях, тип приймаємого значення, та параметри (рисунок 2.8)

```
1   package org.microservice.client;
2
3   import ...
8
9   @Path("/customers")
10  @RegisterRestClient
11  public interface CustomersClient {
12      @GET
13      @Path("/{id}")
14      @Consumes(MediaType.APPLICATION_JSON)
15      @Produces(MediaType.APPLICATION_JSON)
16      Uni<Customer> getCustomer(@PathParam("id") Long id);
17
18  }
```

Рисунок 2.8 – Клієнт для спілкування з Customer мікросервісом.

Що сильно спрощує написання коду. Залишається додати конфігурацію в application.properties файл, з адресою знаходження Customer мікросервіса (рис. 2.9)

```
quarkus.devservices.enabled=false

quarkus.datasource.db-kind=postgresql
quarkus.datasource.reactive.url=postgresql://192.168.1.111:5435/orders
quarkus.datasource.reactive.max-size=20
quarkus.datasource.username=user
quarkus.datasource.password=password
quarkus.hibernate-orm.database.generation=drop-and-create

quarkus.rest-client."org.microservice.client.CustomersClient".url=http://192.168.1.111:8081
quarkus.rest-client."org.microservice.client.ItemsClient".url=http://192.168.1.111:8082

```

Рисунок 2.9 – Файл конфігурації для Orders мікросервіса.

Для мікросервісної архітектури було вирішено зробити три окремих бази даних. Так як при такому підході не буде сповільнення роботи, якщо три мікросервіси будуть звертатись до однієї бази даних. І це є поширенною практикою виділяти свою базу даних під кожний мікросервіс, або групу мікросервісів.

```yaml
version: "3.9"
services:
  postgres-items:
    image: postgres:16
    environment:
      POSTGRES_DB: "items"
      POSTGRES_USER: "user"
      POSTGRES_PASSWORD: "password"
    ports:
      - "5433:5432"
  postgres-customers:
    image: postgres:16
    environment:
      POSTGRES_DB: "customers"
      POSTGRES_USER: "user"
      POSTGRES_PASSWORD: "password"
    ports:
      - "5434:5432"
  postgres-orders:
    image: postgres:16
    environment:
      POSTGRES_DB: "orders"
      POSTGRES_USER: "user"
      POSTGRES_PASSWORD: "password"
    ports:
      - "5435:5432"
```

Рисунок 2.10 – docker-compose файл для трьох баз даних мікросервісного додатку.

Завдяки цим новим введеням docker-compose файл (рис. 2.10) було розширено, для того щоб за допомогою команди можна було запустити три бази даних – одна для кожного мікросервісу. Відповідно для кожного мікросервіса була налаштована своя база даних, це можна побачити на прикладі строчок конфігурацій quarkus.datasource. reactive.url (див. рис. 2.9) для мікросервіса Orders.

2.3 Розробка Cloud-Native модуля з використанням Quarkus native build

Quarkus Native Build (нативна компіляція) – це функціонал Quarkus, спрямований на створення нативних виконуваних файлів для Java-додатків. Нативні виконувані файли дозволяють додаткам стартувати швидше та займати менше обсягу пам'яті, що є критичними аспектами для хмарних та мікросервісних рішень.

Підтримка нативної компіляції була введена після випуску основної версії Quarkus, яка вийшла в 2019 році. Оптимізація для створення нативних виконуваних файлів стала важливою для Quarkus, оскільки це відповідало потребам сучасних Cloud-Native архітектур.

Quarkus Native Build завоював популярність через свою здатність покращити продуктивність та ресурсозбереження додатків. Оптимізація для нативних виконуваних файлів робить Quarkus привабливим вибором для проєктів, де важливо мінімізувати витрати ресурсів та часу старту.

Інформацію про Quarkus Native Build можна знайти в офіційній документації Quarkus, де наведені приклади як все правильно налаштувати.

Варто зазначити те, що частіше всього Quarkus Native Build працює одразу, на основі вже написаного коду за допомогою Quarkus фреймворку, але іноді треба витратити час на конфігурацію, або вкласти можливість нативної компіляції на стадії розробки.

2.2.1 Створення проєкту

Так як Quarkus Native Build працює на основі вже написаного коду для мікросервісів, було внесено мінімальну кількість правок.

Проте, використання нативної компіляції може внести певні обмеження, особливо стосовно рефлексії, динамічної загрузки класів та інших JVM-особливостей.

Анотація @RegisterForReflection (рис 2.11) в Quarkus дозволяє вказати, що певний клас повинен бути доступним для рефлексії під час виконання, коли додаток скомпільовано в нативний бінарний файл за допомогою GraalVM. Це важливо, оскільки за замовчуванням, при нативній компіляції, багато класів не доступні для рефлексії, щоб зберегти час запуску та зменшити розмір виконуваного файлу.

На прикладі мікросервісу Order цю анотацію було використано в класах які використовуються для запитів до мікросервісів Item та Customer за допомогою HTTP. Із-за того що бібліотека Jackson використовує рефлексію для перетворення класів на JSON об’єктів для пересилання по мережі.

```
1  package org.microservice.client.request;
2
3  import io.quarkus.runtime.annotations.RegisterForReflection;
4
   5 usages
5  @RegisterForReflection
6  public class Customer {
       2 usages
7      public Long id;
       1 usage
8      public String mail;
9
       1 usage
10     public String username;
11
       no usages
12     public Customer(Long id, String mail, String username) {
13         this.id = id;
14         this.mail = mail;
15         this.username = username;
16     }
17
       no usages
18     public Customer() {
19     }
20 }
21
```

Рисунок 2.11 – Використання анотації @RegisterForReflection для класу запита до Customer мікросервісу.

Це особливо корисно для класів, які використовуються сторонніми бібліотеками, які можуть здійснювати рефлексивні виклики, наприклад, бібліотеки серіалізації/десеріалізації JSON, такі як Jackson або Gson.

2.2.2 Особливості впровадження Cloud-Native архітектури

Так як для Cloud-Native архітектури було обрано нативну компіляцію, потрібно виконати передумови для того щоб була можливість зробити нативну компіляцію.

Детальніший опис на рахунок налаштування середовища можна подивитись за детальним посібником від розробників Quarkus розташованим за посиланням: https://quarkus.io/guides/building-native-image.

Для початку треба встановити GraalVM CE (community edition), і встановити відповідні значення змінних оточення – GRAALVM_HOME та JAVA_HOME. Перевірити це можна написавши команду java -version в термінал (рис 2.12) і побачивши відповідний надпис про використання потрібної версії.

```
C:\Users\floppa>java -version
java version "21.0.2" 2024-01-16 LTS
Java(TM) SE Runtime Environment Oracle GraalVM 21.0.2+13.1 (build 21.0.2+13-LTS-jvmci-23.1-b30)
Java HotSpot(TM) 64-Bit Server VM Oracle GraalVM 21.0.2+13.1 (build 21.0.2+13-LTS-jvmci-23.1-b30, mixed mode, sharing)

C:\Users\floppa>
```

Рисунок 2.11 – Результат виконання команди java -version при правильно встановленій версії GraalVM CE.

Після цього потрібно встановити середовище розробника для C, бо ми створюємо нативний додаток для Windows. Це можна зробити встановивши Visual Studio 2022 Visual C++ Build Tools.

На цьому підготовка до створення нативного додатку під Windows завершені. Тепер треба виконати команду mvnw install -Dnative в кореневій теці мікросервісу, для того щоб почати створення нативного виконуваного файлу. Після двух хвилин очікування і повідомлення про вдале створення

(рисунок 2.12) в /target теці можна побачити виконуваний файл items-1.0.0-SNAPSHOT-runner.exe – це і є нативний додаток.

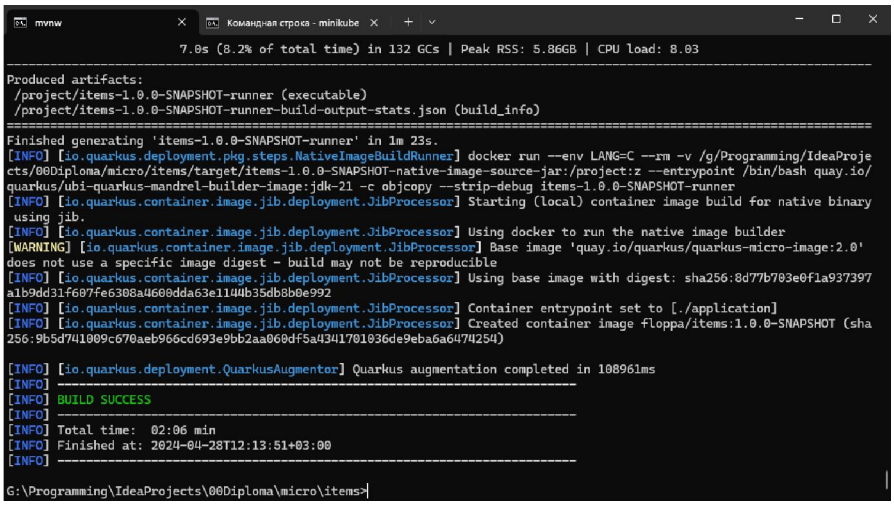


Рисунок 2.11 – Повідомлення про успішно створення нативного додатку.

Після успішного створення нативних додатків для кожного з мікросервісів, в Docker з’являються image відповідних мікросервісів (рис. 2.12). Вони будуть використані для того щоб запустити три мікросервіси в кластері Kubernetes, а саме в Minikube, бо він ідеально підходить для локального тестування та простіший в освоєнні.

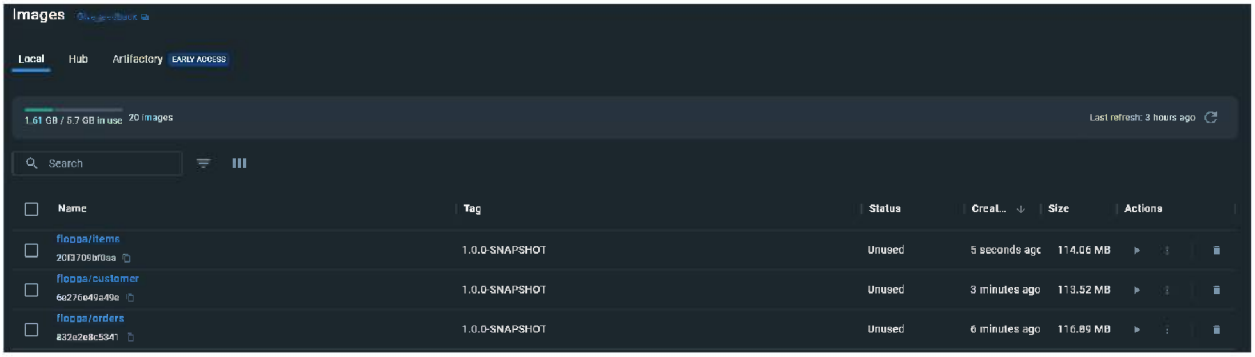


Рисунок 2.12 – Наявність Docker image мікросервісів в локальному репозиторії.

Для розгортання мікросервісів в кластері Kubernetes знадобиться встановлений Minikube. Запустивши його за допомогою команди minikube start, було введено ще команду minikube dashboard. Це веб-додаток який

дозволяє візуально спостерігати за мікросервісами, та тим як вони запустились та працюють.

Для того щоб запустити мікросервіси в кластері треба внести відповідні зміни в properties файли, і написати файли сервісів та розгортання. В ситуаціїї з Cloud-Native архітектурою було використано три файли (повний вміст файлів в додатку Г):

1) db-creds.yml

Файл в якому зберігається конфігурація імені, паролів, назви для баз даних.

2) db-deploy.yml

Файл в якому описано три бази даних, які будуть використовувати параметри налаштувань використовуючи інформацію з db-creds.yml

3) microservices.yml

Файл в якому описано розгортання та параметри мікросервісів Orders, Items, Customers, з відповідними конфігруаціями баз даних.

Тепер по черзі треба виконати команду kubectl apply -f Назва_Файлу, для кожного з них, спочатку db-creds, потім db-deploy, і останнім microservices.

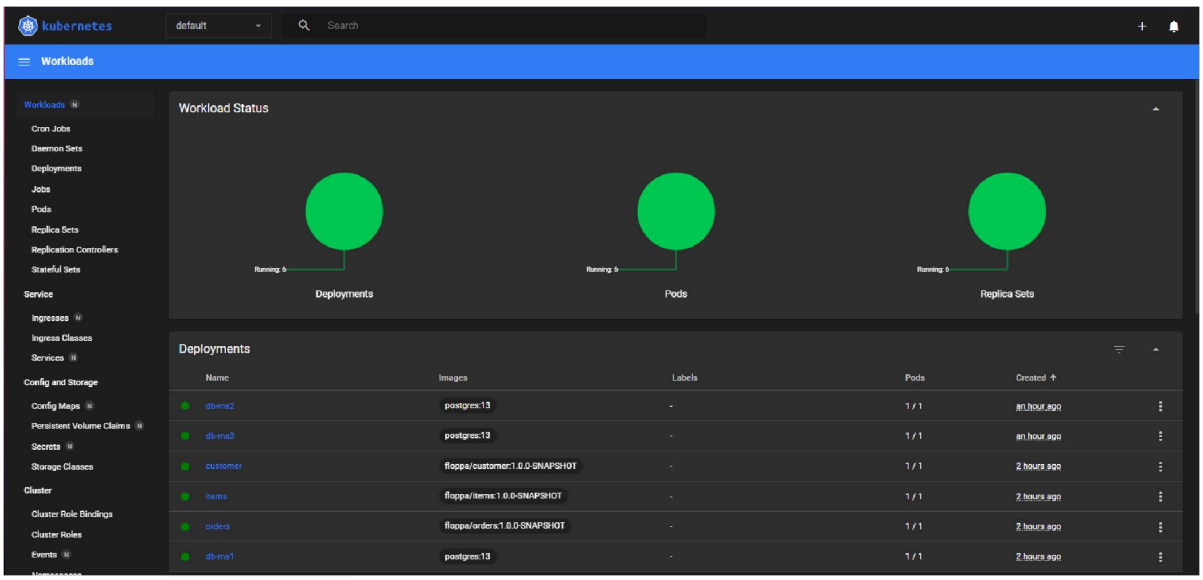


Рисунок 2.13 – Результат розгортання мікросервісів та баз даних у кластері Kubernetes.

Якщо все пройшло успішно, то на веб-сторінці Kubernetes dashboard, який було відкрито раніше, можна побачити всі додатки які ми розгорнули (рис. 2.13), і відповідні зелені відмітки що вони повідомили про успішний запуск.

На цьому впровадження Cloud-Native архітектури можна вважати завершеним. Так як сучасні хмарні обчислювальні середовища розраховані на те щоб працювати з Kubernetes, і ми задовільнили цю потребу.

Висновки за розділом 2

В цьому розділі було розглянуто впровадження архітектурних підходів при розробці додатків, які будуть протестовано.

Для кожного з типів архітектури було описано особливості впровадження архітектурного підходу. Для монолітного додатку – єдина кодова база з однією базою даних. Для мікросервісного – три додатки з трьома базами даних які спілкуються між собою по HTTP. І для Cloud-Native – три додатки яки було нативно скопильовано і запущено в кластері Kubernetes.

Для монолітної архітектури на основі Spring Boot було використано веб-інтерфейс швидкого старту проєкту, що дозволяє створити проєкт з необхідними налаштуваннями та залежностями. За допомогою цього інструменту можна швидко і легко налаштувати проєкт, обравши потрібні залежності, такі як Spring Web, Lombok, Spring Data JPA та PostgreSQL Driver. Ці залежності дозволяють спростити розробку веб-додатків та взаємодію з базою даних. Після генерації проєкту та його імпортування в інтегроване середовище розробки, можна швидко розпочати роботу над монолітним додатком, зберігаючи всю логіку в одному місці. Одна з особливостей монолітної архітектури – це єдина кодова база, яка полегшує пошук та розгортання додатку, проте може призводити до більшого часу запуску та складнощів у зміні окремих модулів без перекомпіляції усієї кодової бази.

Для мікросервісної архітектури було розглянуто використання Quarkus, фреймворку, спеціально розробленому для створення оптимізованих мікросервісів та Cloud-Native застосунків. Цей фреймворк надає низький час

старту та оптимізацію ресурсів, що робить його ідеальним для високопродуктивних ініціатив, таких як serverless та контейнеризація. Крім того, було описано процес створення проєкту Quarkus за допомогою інтерфейсу code.quarkus.io, що дозволяє швидко налаштувати необхідні розширення та налаштування. Також описано особливості впровадження мікросервісної архітектури з використанням Quarkus, зокрема, розділення баз даних для окремих мікросервісів та налаштування спілкування між ними через HTTP запити.

Для Cloud-Native архітектури було викладено докладний опис процесу налаштування середовища для нативної компіляції додатків, використання відповідних інструментів та технологій, таких як GraalVM CE та Visual Studio 2022 Visual C++ Build Tools. Проведено докладний аналіз процесу створення нативних виконуваних файлів та їх розгортання в середовищі Docker та Kubernetes, використовуючи Minikube для локального тестування. Описані необхідні кроки і файлова структура для розгортання мікросервісів та баз даних в кластері Kubernetes, а також процес перевірки успішності запуску за допомогою веб-сторінки Kubernetes dashboard. Впровадження Cloud-Native архітектури є завершеним, надаючи можливість працювати з сучасними хмарними обчислювальними середовищами та використовувати переваги Kubernetes для кращого керування та масштабування додатків.

РОЗДІЛ 3
ТЕСТУВАННЯ ТА АНАЛІЗ СИСТЕМ, СТВОРЕННЯ АВТОМАТИЗАЦІЇ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Методика тестування, та характеристики які будуть порівнюватись

У процесі аналізу швидкодії та ефективності програмних додатків ключовим аспектом є методика тестування, яка включає визначення критеріїв для порівняння різних характеристик.

Однією з важливих характеристик є час запуску додатка. Час запуску відіграє важливу роль, оскільки він безпосередньо впливає .

Довгий час запуску може відштовхнути користувачів, особливо у середовищах, де швидке надання послуг є критично важливим. Таким чином, оптимізація часу запуску може значно покращити загальне сприйняття додатку кінцевими користувачами.

Ще одним важливим параметром для порівняння є обсяг пам'яті, занятого на жорсткому диску. Ефективне використання дискового простору є критичним для додатків, які встановлюються на пристрої з обмеженим дисковим простором, так як в хмарному середовищі це дозволяє знайти поле для економії, та обрати варіант хмарної обчислювальної машини яка буду коштувати дешевше.

Використання оперативної пам'яті є третьою критичною характеристикою, яка аналізується під час тестування додатків. Високе використання оперативної пам'яті може призводити до загального зниження продуктивності системи, особливо якщо вона вимушена використовувати swap, що значно сповільнює роботу. Для додатків, що працюють у фоновому режимі або додатків, які мають постійно активні процеси, оптимізація використання оперативної пам'яті може значно підвищити ефективність роботи декількох програм одночасно.

JMeter являється потужним інструментом для тестування, він має візуально зрозумілий інтерфейс (рис 3.1) та потужні можливості щодо навантаження додатків.

Було створено тестовий сценарій, однаковий для всих трьох систем, так як вони працюють ідентично. Спочатку за допомогою setup Thread Group робляться по 10 записів с випадковими значеннями у Items та Customer бази даних за допомогою відповідних модулів, ці дані потрібні в майбутньому і вони не використовуються в тестуванні.

Далі основна група Main test Group робить налаштовану кількість запитів протягом 5 хвилин. Це було зроблено для того щоб можно було підрахувати пропускну здатність, час відгуку, та кількість запитів при якому з'являються помилки, або відкидуються запити за неможливістю їх обробки.

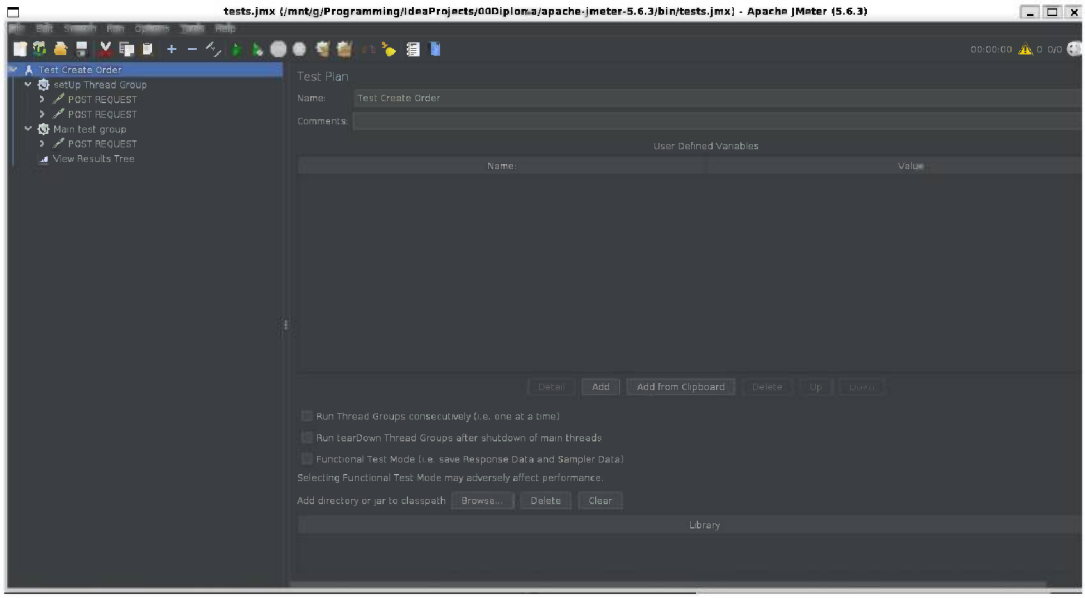


Рисунок 3.1 – Тестовий сценарій Jmeter.

Під час тестування використовуються звіти JMeter, які надають детальну інформацію про різні аспекти процесу виконання додатків.

Основні метрики яки будуть отримані за допомогою Jmeter – затримка, та пропускна здатність додатків.

Затримка означає час, необхідний для обробки однієї операції або запиту.

Вона охоплює різні компоненти, такі як час обробки, час мережевого транзиту та затримки в черзі. Низька затримка є бажаною, оскільки це означає швидкий час відгуку і більш чуйний користувальницький досвід.

Пропускна здатність, з іншого боку, вимірює обсяг операцій, які система може обробити за певний проміжок часу.

Вона відображає здатність системи обробляти безліч запитів одночасно. Висока пропускна здатність свідчить про здатність системи ефективно керувати і масштабуватися зі збільшенням попиту [4].

Аналізуючи ці дані, можна отримати уявлення про стабільність та масштабованість додатка, що є важливим для розробки високопродуктивних та надійних систем.

Загалом, вибір методики тестування та характеристик, які будуть порівнюватися, є вирішальним для забезпечення ефективності та швидкодії програмних рішень, і ці характеристики грають важливу роль при виборі архітектурного підходу при розробці автоматизаційного рішення. Ретельно підібрані параметри дозволяють не тільки виявляти потенційні проблеми перед впровадженням продукту в експлуатацію, але й оптимізувати продукт для кращого користувацького досвіду.

3.2 Проведення тестування, та висвітлення результатів

Після проведення заміру часу запуску були отримані наступні результати (рис. 3.2).

Рисунок 3.2 – Графік часу запуску додатків.

Над стовпцями можна побачити час в секундах, і візуально легко побачити різницю між часом запуску. Це дуже важливий параметр, так як при оновленні додатку, буде мінімальний час коли він буде недоступний. Також, чим менше час, тим простіше масштабуватись, додаючи нові реплікі додатку.

Швидкість запуску Cloud-Native додатків на два порядки швидше, ніж монолітного та мікросервісного. Що дозволяє миттєво масштабуватись за потреби, і так швидко прибирати репліки коли вони не потрібні.

Далі було проведення тестування оперативної пам’яті (рис. 3.3), так як це важливий параметр при виборі машини для хмарного обчислення, також цей параметр важливий для монолітного додатку, коли треба обирати обчислювальну машину, з вільним простором на майбутнє.

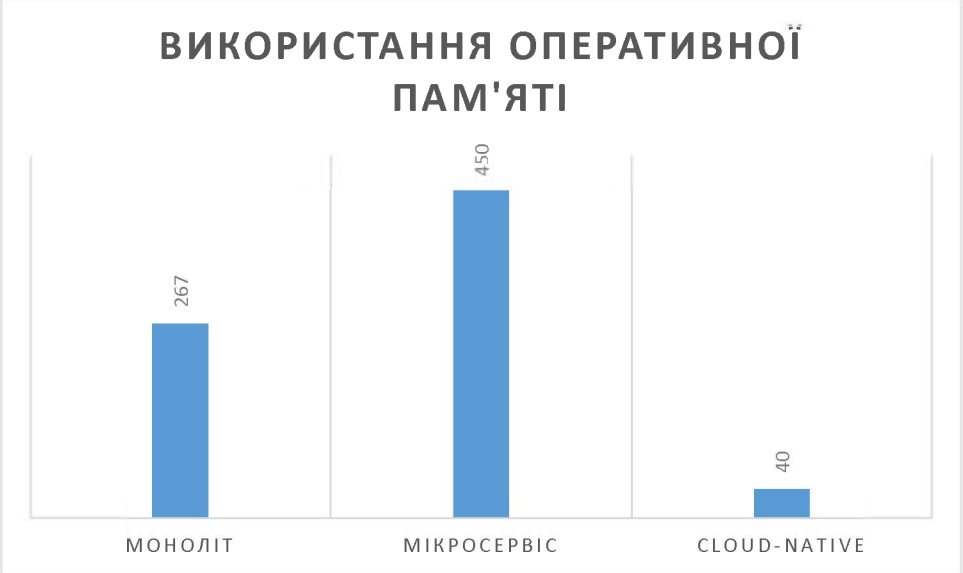


Рисунок 3.3 – Графік використання оперативної пам’яті додатків.

Треба зазначити те що в монолітному додатку треба було зафіксувати лише один показник, в той час як для мікросервісної та Cloud-Native архітектур, результатом є усереднена сума усих запущених додатків. Цим і можна пояснити такий великий розмір для мікросервісної архітектури, але слід і відмітити знову дуже компактний розмір Cloud-Native додатків.

Слідучим показником який було протестовано став обсяг зайнятої пам’яті на накопичувач (рис 3.4). І хоча цей параметр не відіграє суттєвої ролі

при виборі архітектурного підходу для розробки програмного забезпечення, результати отримані в ході цього тестування можуть виявитись корисними.

Рисунок 3.4 – Графік використання пам’яті на накопичувачі.

Як і в попередньому тесті, в монолітному додатку треба було зафіксувати лише один показник, в той час як для мікросервісної та Cloud-Native архітектур, результатом є усереднена сума усих відкомпільованих додатків.

Тому такий розмір легко пояснити для мікросервісних додатків, бо кожен з них приблизно займавст 40 Мб, що нижче ніж у монолітному додатку, але так як було розроблено три мікросервіси, потрібно враховувати і зробити суму всіх трьох.

На рахунок Cloud-Native додатків то у випадку компіляції виконуваних файлів для нативних платформ, включення всіх необхідних бібліотек та ресурсів безпеки у вихідний файл призводе до збільшення розміру програми. Коли всі ці компоненти компілюються до нативного виконуваного файлу, це призводить до збільшення його розміру.

Далі було використано інструмент Apache JMeter для стресс тестування розроблених додатків.

В першу чергу подивимось на результати пропускної здатності (рис. 3.5), так як вона є критично важливою характеристикою для веб-додатків, оскільки вона визначає здатність системи обробляти велику кількість запитів від користувачів за одиницю часу без втрати продуктивності та ефективності.

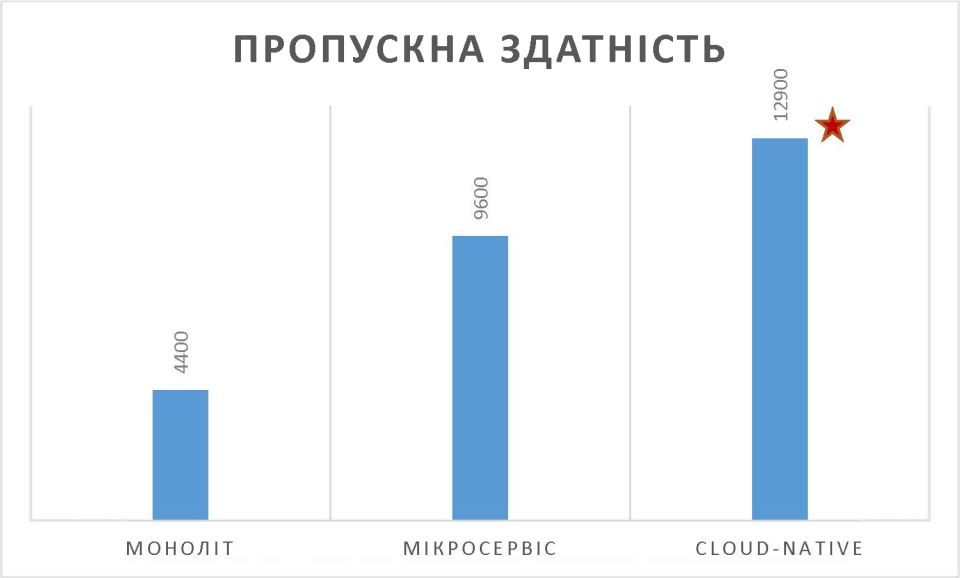


Рисунок 3.5 – Пропускна здатність (запитів в хвилину).

Одразу треба зробити зауваження, що в ситуації Cloud-Native додатку пропускна здатність може виходити за межі 12000 запитів в секунду. Але із-за обмежень наявних комплектуючих це вийшло максимальним результатом який я зміг протестувати.

Монолітна архітектура демонструє вражаючий результат в 4500 оброблених запитів в хвилину, і якщо очікуване навантаження дорівнює цьому значенню або кратно менше, це буде ідеальним варіантом по причині простоти розробки та впровадження.

Мікросервісна архітектура в свою чергу показала результат в 9600 запитів в хвилину, що більше ніж в два рази більше ніж додаток створений за принципами монолітної архітектури.

Cloud-Native додаток показав результат в 12900 оброблених запитів в хвилину, що є максимальним з трьох.

Наступним було зібрано метрики на рахунок затримки кожного з додатків. Спочатку подивимось на максимальну затримку (рис 3.6).

Аналіз даних затримок між різними архітектурними підходами демонструє значні відмінності у продуктивності систем. Монолітна архітектура має найвищу максимальну затримку, яка складає 2120 мс, що може бути обумовлено централізованою природою обробки даних і великим навантаженням на єдиний сервер або базу даних. Мікросервісна архітектура показує покращення з максимальною затримкою 1575 мс, що відображає переваги децентралізації: кожен сервіс обробляє лише частину загального навантаження, що дозволяє зменшити час відгуку. Найкращі показники затримки має Cloud-Native архітектура з показником у 730 мс, що вказує на високу ефективність контейнеризації та оркестрації сервісів у Kubernetes, яка забезпечує розподілення ресурсів і плюси нативної компіляції.

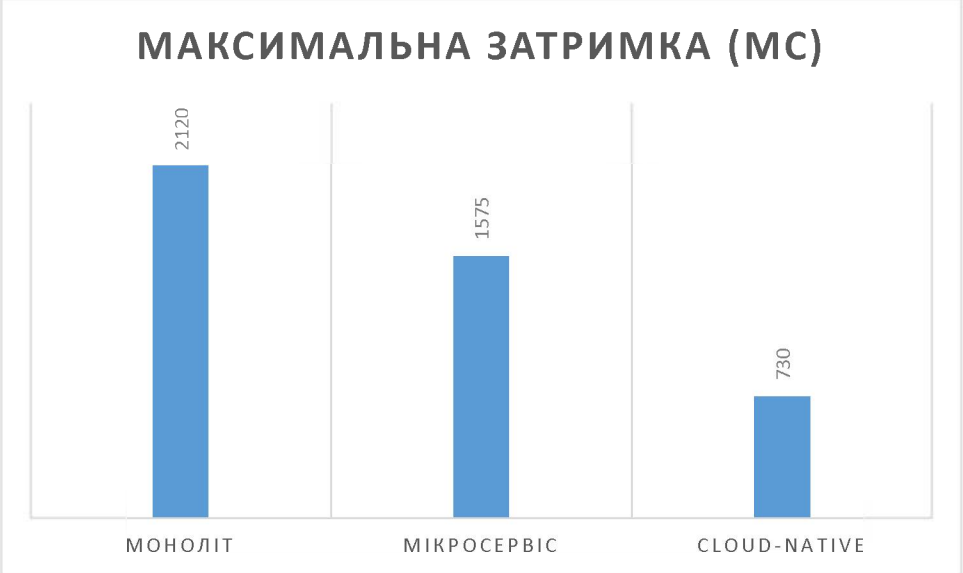


Рисунок 3.6 – Максимальна затримка (мс)

Apache JMeter також може додати метрики медіани, 90, 95, 99 перцентилі затримки, що допоможе глибше зрозуміти затримки (рис 3.7).

Аналіз даних затримок для різних архітектур веб-додатків підкреслює важливість вибору правильного архітектурного підходу у забезпеченні продуктивності системи. Монолітна архітектура, незважаючи на значну максимальну затримку в 2120 мс, показує вражаюче низькі медіану та 90-й перцентиль затримок, що становлять 10 мс. Це може вказувати на високу ефективність моноліту при обробці стандартних запитів, але також на значне

погіршення продуктивності при пікових навантаженнях, що відображено у значеннях 99 перцентиля в 198 мс.

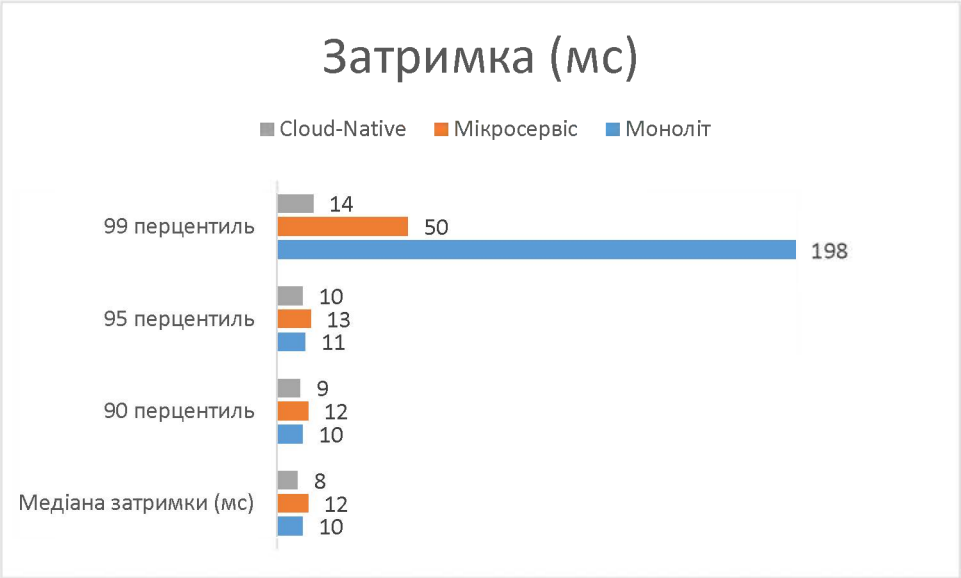


Рисунок 3.7 – Статистика затримок (мс)

Архітектура на основі мікросервісів показує більш стабільні результати з максимальною затримкою 1575 мс та медіаною в 12 мс. Всі перцентильні значення лишаються відносно низькими, з невеликим зростанням до 50 мс у 99 перцентилі. Це вказує на більшу здатність архітектури до стабільності та передбачуваності в обробці запитів, навіть при підвищених навантаженнях.

Cloud-Native архітектура демонструє найкращі показники по всіх метриках: найменша максимальна затримка в 730 мс, низька медіана в 8 мс, та дуже малі значення перцентилів 9, 10 та 14 мс відповідно. Ці дані свідчать про високу ефективність та швидкість обробки запитів, а також про здатність архітектури ефективно масштабуватись та обробляти пікові навантаження без значних затримок.

3.3 Створення системи автоматизації на основі використання архітектурних підходів використовуючи отримані дані

Проведені тестування показали доцільнцість використання архітектурних підходів при певних сукупностей характеристик, в умовах вимог та обмежень представлених в таблиці.

Основуючись на отриманих результатах було розроблено рекомендаційну модель обрання архітектурного підходу за параметрами (рис. 3.8).

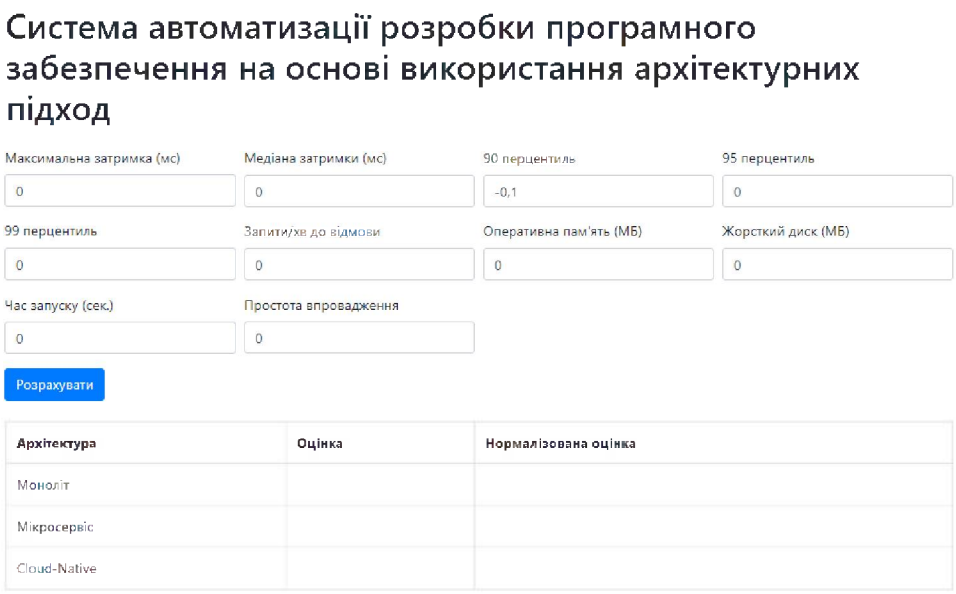


Рисунок 3.8 – Зовнішний вигляд моделі автоматизації.

Даний додаток має в собі всі дані які були зібрані в розділі тестування та візуально інформує про найбільш підходящий архітектурний стиль залежно від потреб проєкту. Нормалізація даних дозволила кількісно порівняти різні архітектурні підходи за параметрами, такими як затримка, обробка запитів, використання ресурсів та час запуску.

Так, наприклад, архітектура Cloud-Native показала найменші затримки і вражаючу ефективність при високих навантаженнях (до 12,900 запитів в хвилину), що робить її ідеальною для систем з великою кількістю одночасних користувачів та високими вимогами до швидкості обробки. Водночас, монолітна архітектура, незважаючи на вищу максимальну затримку, має велику перевагу в мінімальному використанні оперативної пам’яті і низькі вимоги до місця на жорсткому диску, що може бути критично для додатків з обмеженими ресурсами або для розгортання на старому обладнанні.

Мікросервісна архітектура знаходиться на золотій середині, пропонуючи баланс між витратами на ресурси та здатністю обслуговувати значну кількість запитів. Цей підхід є оптимальним для проєктів, які вимагають модульності та еластичності без надмірного навантаження на апаратні ресурси.

Розроблену систему було протестовано введенням вагових коефіцієнтів для кожного з параметрів та натиснення кнопки “Розрахувати“ (рис. 3.8).

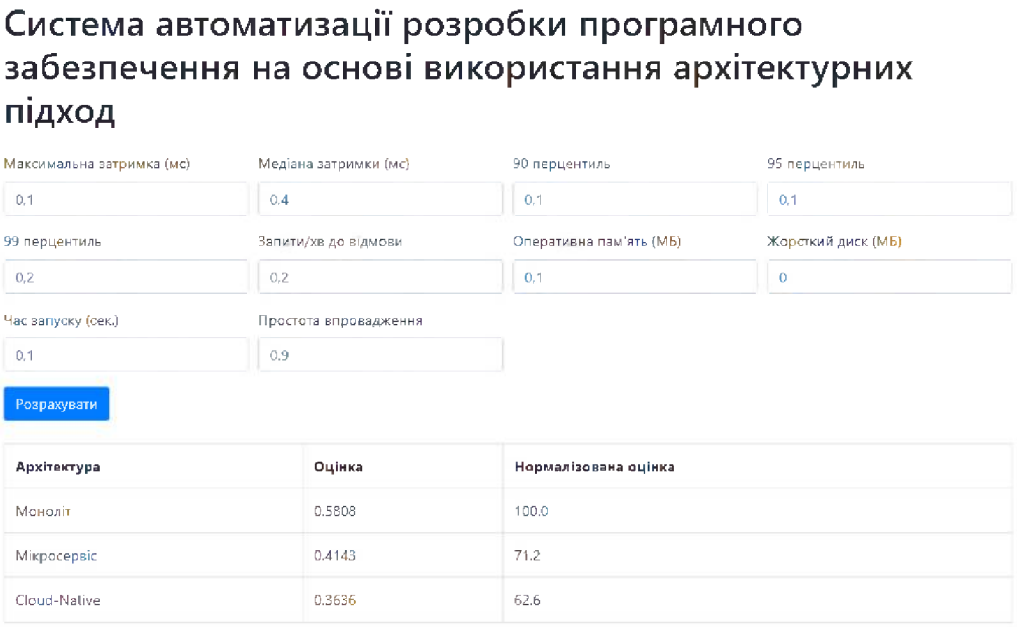


Рисунок 3.8 – Перевірка роботи моделі автоматизації.

Якщо обрати за параметрами найбільш вагомими параметрами простоту впровадження та медіану затримки, то монолітна архітектура видає результат в 100 балів, що логіно, бо вона демонструє гарні результати як і по медіанній затримці, так і найкращий результат за простотою впровадження.

Остаточний вибір архітектури має базуватися не тільки на технічних параметрах, але й враховувати довгострокову стратегію розвитку проєкту, можливості масштабування та вимоги до підтримки. Ця модель рекомендацій надає зрозумілі критерії для вибору, які можуть бути адаптовані до специфіки будь-якого проєкту, забезпечуючи високу адаптивність і гнучкість в управлінні програмними рішеннями.

Висновки за розділом 3

Після проведення тестування швидкодії та ефективності програмних додатків, виявлено, що час запуску програми є ключовим аспектом, особливо в середовищах, де швидке надання послуг є критично важливим. Оптимізація часу запуску може покращити загальне сприйняття додатку користувачами. Також важливим параметром для порівняння є обсяг пам'яті, зайнятого на жорсткому диску. Ефективне використання дискового простору є критичним для додатків, що встановлюються на пристроях з обмеженим дисковим простором.

Тестування використання оперативної пам'яті також виявилося важливим, оскільки високе використання оперативної пам'яті може призводити до загального зниження продуктивності системи. Аналіз даних звітів JMeter надає детальну інформацію про різні аспекти виконання додатків, такі як час відгуку, кількість помилок та пропускна здатність, що є важливим для оцінки якості роботи додатків.

Після отримання результатів тестування було створено рекомендаційну модель, що може рекомендувати відповідний архітектурний стиль відповідно до визначених параметрів. Модель інформує про найкращі архітектурні рішення для конкретних вимог проєкту, базуючись на нормалізованих даних з розділу тестування. Це дозволяє оцінити і порівняти різні підходи, забезпечуючи зрозумілість і простоту вибору архітектурних рішень для ефективного управління програмними проєктами.

ВИСНОВКИ

В роботі було проведено аналіз та порівняння трьох різних архітектурних підходів у розробці програмного забезпечення: монолітної, мікросервісної та Cloud-Native архітектур. У роботі було досліджено різновиди існуючих автоматизаційних систем та проведений аналіз наявних популярних технологічних рішень в області архітектури програмного забезпечення.

Було розроблено три програмних модулі з використанням цих архітектурних підходів, для подальшого тестування та збору параметрів.

Було зібрано дані щодо різних характеристик цих підходів, таких як максимальна затримка, кількість запитів до відмови, використання оперативної пам'яті та обсяг пам'яті на жорсткому диску.

На основі отриманих даних була розроблена рекомендаційна модель для вибору архітектурного підходу в залежності від конкретних вимог та обмежень проєкту. Виявлено, що кожен з розглянутих підходів має свої переваги та недоліки, і вибір оптимального підходу залежить від конкретних вимог проєкту та його контексту.

Для практичного застосування рекомендаційної моделі важливо враховувати специфіку проєкту, його потреби та обмеження. Такий підхід дозволить забезпечити логічний розподіл ресурсів та досягнення поставлених цілей проєкту з аргументованими витратами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fritz Solms. What is Software Architecture? Conference: SAICSIT '12 Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference Volume:ACM P.363-373 DOI: 10.1145/2389836.2389879
2. The Comparison of Microservice and Monolithic Architecture Conference: 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH) DOI:10.1109/MEMSTECH49584.2020.9109514
3. Cloud Native Application Architecture [Електронний ресурс], URL:https://medium.com/walmartglobaltech/cloud-native-application-architecture-a84ddf378f82 (Дата звернення: 03.04.2024).
4. Latency vs Throughput in Spring Boot Applications [Електронний ресурс], URL: https://levelup.gitconnected.com/latency-vs-throughput-in-spring-boot-applications-85bc68308752 (Дата звернення: 01.04.2024).
5. From Monolithic Systems to Microservices: A Comparative Study of Performance [Електронний ресурс], URL:https://www.mdpi.com/2076-3417/10/17/5797 (Дата звернення: 20.04.2024).
6. Nada Salaheddin ELGHERIANI, Nuredin Ali Salem AHMED. MICROSERVICES VS. MONOLITHIC ARCHITECTURES [THE DIFFERENTIAL STRUCTURE BETWEEN TWO ARCHITECTURES]. International Journal of Applied Sciences and Technology ISSN: 2717-8234 Article type: Research Article
7. Andrzej Barczak, Michał Barczak. Performance comparison of monolith and microservices based applications. [Електронний ресурс], URL: https://www.iiis.org/CDs2021/CD2021Summer/papers/SA354XK.pdf (Дата звернення: 23.04.2024).
8. Monolithic vs. Microservice Architecture: A Comprehensive Comparison [Електронний ресурс], URL: https://www.linkedin.com/pulse/monolithic-

vs-microservice-architecture-comprehensive-girish-vas (Дата звернення: 24.04.2024).

9. Getting started with API Load Testing (Stress, Spike, Load, Soak) [Електронний ресурс], URL: https://www.youtube.com/watch?v=r-Jte8Y8zag (Дата звернення: 24.04.2024).
10. Quarkus Official Documentation, Latest Guides. [Електронний ресурс], URL: https://quarkus.io/guides/ (Дата звернення: 27.04.2024).
11. Spring Boot Official Documentation. [Електронний ресурс], URL: https://docs.spring.io/spring-boot/index.html (Дата звернення: 27.04.2024).
12. (The Art of) (Java) Benchmarking.[Електронний ресурс], URL: https://shipilev.net/talks/j1-Oct2011-21682-benchmarking.pdf (Дата звернення: 01.05.2024).

Додаток А

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Факультет комп'ютерних наук
Кафедра теоретичної та прикладної системотехніки
Рівень вищої освіти (освітньо-кваліфікаційний рівень) **бакалавр**
галузь знань:**15 – Автоматизація та приладобудування**
спеціальність:**151 – Автоматизація та комп'ютерно-інтегровані технології**

ЗАТВЕРДЖУЮ
Завідувач кафедри теоретичної
та прикладної системотехніки
_____д.т.н., проф. Шматков С. І.
«21» грудня 2024 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Чинякова Данила Сергійовича
(прізвище, ім'я, по батькові студента)

1. Тема роботи «**Інформаційна технологія розробки програмного забезпечення на основі використання архітектурних підходів**»

керівник роботи Мороз Ольга Юріївна, PhD, старший викладач ЗВО
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «03» травня 2024року № 4101-5/909

2.Строк подання студентом роботи 31 травня 2024року

3. Перелік питань, які потрібно розробити
1. Аналіз проблеми вибору архітектурного підходу до розробки программного забезпечення, аналіз популярних існуючих архітектурнів підходів.
2. Визначення основних характеристик для порівняння архітектурних підходів.
3. Розробка та реалізація модулів для автоматизації внутрішнього обліку персоналу та обробки онлайн-замовлень за допомогою монолітної архітектури, мікросервісної архітектури та Cloud-Native архітектури.
4. Проведення тестування модулів
5. Аналіз та порівняння даних розроблених моделей.

4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1	Аналіз академічної та наукової літератури за темою роботи та аналіз актуальності вибору архітектурного підходу до розробки программного забезпечення.	31.10.2023 – 15.01.2024
2	Аналіз методів провадження сучасних архітектурних підходів.	19.12.2023 – 02.01.2024
3	Розробка програмних модулів на основі використання монолітної, мікросервісної та Cloud-Native архітектур.	02.01.2024 – 02.02.2024
4	Тестування та збір характеристик розроблених моделей.	02.02.2024 – 19.03.2024
5	Аналіз та порівняння даних розроблених моделей.	20.03.2024 – 30.04.2024
6	Опис інформаційної технології за результатами отриманих при тестувані.	30.03.2024 – 30.04.2024
7	Підготовка статті на тему кваліфікаційної роботи.	31.03.2024 – 31.05.2024
8	Оформлення звіту за результатами переддипломної практики.	15.05.2024 – 31.05.2024
9	Представлення кваліфікаційної роботи керівнику та рецензенту.	31.05.2024
10	Оформлення пояснювальної записки та підготовка презентації.	31.03.2024 – 31.05.2024

5. Дата видачі завдання 21.12.2023

Студент Д. С. Чиняков
ініціали, прізвище
підпис

Керівник роботи О. Ю. Мороз
ініціали, прізвище
підпис

Додаток Б

Технічне завдання на розробку програмного вироби «Система автоматизації розробки програмного забезпечення на основі використання архітектурних підходів»

Назва розділу	Назва і зміст підрозділу
1. Введення	1.1. Назва програмного виробу – Система автоматизації розробки програмного забезпечення на основі використання архітектурних підходів. 1.2. Галузь застосування – 15 Автоматизація та приладобудування.
2. Підстава для розробки	2.1. Навчальний план за спеціальністю 151 – Автоматизація та комп'ютерно-інтегровані технології. 2.2. Завдання на кваліфікаційну роботу бакалавра, затверджено наказом ХНУ імені В. Н. Каразіна № 4101-5/909 від 03.05.2024 р. (представити як Додаток А до пояснювальної записки до кваліфікаційної роботи).
3. Призначення розробки	3.1. Мета розробки програмного вироби – допомога при виборі архітектурного підходу при створенні IT-проєкту за рахунок використання зібраних параметрів при тестуванні. 3.2. Призначення програмного вироби для автоматизації вибору архітектурного підходу за параметрами. 3.3. Початкові дані для розробки: лист задач, їх характеристики.
4. Технічні вимоги до програмного виробу	4.1. Вимоги до функціональних характеристик: 1) представляти з себе програмну реалізацію 2) додати три додатки які використовують три архітектурних підходи для демонстрації особливостей впровадження, і надання можливості особистого тестування 3) надавати зібрані дані при тестуванні 4) надавати додаток з візуальним інтерфейсом який буде зберігати в собі всі дані та містити формули для підрахунку підходящого архітектурного підходу за наданими параметрами та умовами. 4.2. Вимоги до надійності: Можна обрати архітектурний підхід за ваговими коефіцієнтами бажаних характеристик. Розроблені моделі мають однакові відповіді при однакових запитах. 4.3. Вимоги до умов експлуатації офісні приміщення. 4.4. Вимоги до складу і параметрів технічних засобів Персональний комп'ютер у повній комплектації (ноутбук) 4.5. Вимоги до інформаційної та програмної сумісності забезпечити сумісність з усіма обчислювальними засобами. 4.6. Вимоги до маркування та упаковки відсутні. 4.7. Вимоги до транспортування і зберігання відсутні.

	4.8. Спеціальні вимоги не пред'являються.		
5. Вимоги до програмної документації.	Програмною документацією до виробу «Система автоматизації розробки програмного забезпечення на основі використання архітектурних підходів» вважати: 1) Справжнє Технічне завдання на розробку програмного вироби (представити у вигляді Додатку Б до пояснювальної записки до кваліфікаційної роботи). 2) Програму і методику випробувань розробленого програмного вироби (представити у вигляді Додатку В до пояснювальної записки до кваліфікаційної роботи). 3) Опис програмного виробу (представити в розділі 3 пояснювальної записки до кваліфікаційної роботи). 4) Текст програми (представити в Додатку Г до пояснювальної записки до кваліфікаційної роботи).		
6. Техніко-економічні показники	Вимоги до розрахунку техніко-економічних показників не потрібні.		
7. Стадії та етапи розробки	1 етап	Аналіз академічної та наукової літератури за темою роботи та аналіз актуальності вибору архітектурного підходу до розробки программного забезпечення.	31.10.2023 – 15.01.2024
	2 етап	Аналіз методів провадження сучасних архітектурних підходів.	19.12.2023 – 02.01.2024
	3 етап	Розробка програмних модулів на основі використання монолітної, мікросервісної та Cloud-Native архітектур.	02.01.2024 – 02.02.2024
	4 етап	Тестування та збір характеристик розроблених моделей.	02.02.2024 – 19.03.2024
	5 етап	Аналіз та порівняння даних розроблених моделей.	20.03.2024 – 30.04.2024
	6 етап	Опис інформаційної технології за результатами отриманих при тестувані.	30.03.2024 – 30.04.2024
	7 етап	Підготовка статті на тему кваліфікаційної роботи.	31.03.2024 – 31.05.2024
	8 етап	Оформлення звіту за результатами переддипломної практики.	15.05.2024 – 31.05.2024
	9 етап	Представлення кваліфікаційної роботи керівнику та рецензенту.	31.05.2024
	10 етап	Оформлення пояснювальної записки та підготовка презентації.	31.03.2024 – 31.05.2024

8. Порядок контролю і приймання	1) Перевірку ходу розробки програмного виробу керівнику робіт виконувати раз в 2 тижні. 2) Випробування програмного продукту провести відповідно до програми та методики випробувань на базі комп'ютерного класу. 3) Захист розробленої моделі провести на засіданні Атестаційної комісії. 4) Пояснювальну записку подати на паперових носіях в 1 примірнику і в електронному вигляді в 1 примірнику.

Виконавець

студент групи КУ-41

Чиняков Д.С

Замовник

PhD, старший викладач ЗВО

О. Ю. Мороз

ДОДАТОК В

Програма і методика випробувань програмного вироби «Система автоматизації розробки програмного забезпечення на основі використання архітектурних підходів»

1 Об'єкт випробувань

1.1 Найменування випробуваного програмного вироби «Система автоматизації розробки програмного забезпечення на основі використання архітектурних підходів».

1.2 Область його застосування управління проєктами, IT-індустрія і розробка ПЗ.

2. Мета випробувань

Метою кваліфікаційної роботи є дослідження та порівняння різних архітектурних підходів написання программного забезпечення.

3. Загальні положення

3.1 Підстави для проведення випробувань

Підставою для проведення випробувань є наказ про призначення атестаційної комісії.

3.2 Місце і тривалість випробувань

Приймальні (приймально-здавальні) випробування проводяться на базі комп'ютерного класу кафедри в період роботи атестаційної комісії.

3.3 Обсяг випробувань

Приймальні випробування програмного вироби проводяться в обсязі відповідному цієї Програми і методики випробувань.

3.4 Організації, які беруть участь у випробуваннях

Приймальні випробування проводяться атестаційною комісією напередодні засідання (або в процесі засідання) за участю Замовника, Виконавця та інших осіб, присутніх на засіданні.

4. Вимоги до програми або програмного виробу

4.1. Вимоги до функціональних характеристик:

1) представляти з себе програмну реалізацію

2) додати три додатки які використовують три архітектурних підходи для демонстрації особливостей впровадження, і надання можливості особистого тестування

3) надавати зібрані дані при тестуванні

4) надавати додаток з візуальним інтерфейсом який буде зберігати в собі всі дані та містити формули для підрахунку балів архітектурного підходу за наданими параметрами та умовами.

4.2. Вимоги до надійності:

Можна обрати архітектурний підхід за ваговими коефіцієнтами бажаних характеристик. Розроблені моделі мають однакові відповіді при однакових запитах.

4.3. Вимоги до умов експлуатації офісні приміщення.

4.4. Вимоги до складу і параметрів технічних засобів Персональний комп'ютер у повній комплектації (ноутбук)

4.5. Вимоги до інформаційної та програмної сумісності забезпечити сумісність з усіма обчислювальними засобами.

4.6. Вимоги до маркування та упаковки відсутні.

4.7. Вимоги до транспортування і зберігання відсутні.

4.8. Спеціальні вимоги відсутні.

5. Вимоги до програмної документації

Склад програмної документації, що подається на випробування, включає:

1) Технічне завдання на розробку програмного вироби (представлено в Додатку Б до пояснювальної записки до кваліфікаційної роботи).

2) Ця Програма і методика випробувань розробленого програмного вироби (представлена в Додатку В до пояснювальної записки до кваліфікаційної роботи).

3) Опис програмного вироби (представлено в розділі 3 пояснювальної записки до кваліфікаційної роботи).

6. Засоби і порядок випробувань

6.1 Засоби випробувань

Випробування проводяться на технічних засобах, яких персональний комп'ютер, ноутбук.

Випробування проводяться з використанням програмних засобів, яких EXCEL, командний рядок, Kubernetes, Docker, Postman.

6.2 Порядок проведення випробувань

6.2.1. Перевірка програмної документації.

Перевірка комплектності складу програмної документації здійснюється за критерієм наявності зазначеної в технічному завданні документації.

6.2.2. Перевірка якості програмної документації.

Перевірку здійснювати за критерієм відповідності вимогам єдиної системи програмної документації (ЄСПД).

6.2.3. Перевірка виконання програми.

Тест 1: Перевірка відповідей розроблених моделей при однакових запитах. По вмісту запиту та відповіді робиться висновок про правильно працездатність моделей (рисунки В.1, В.2, В.3).

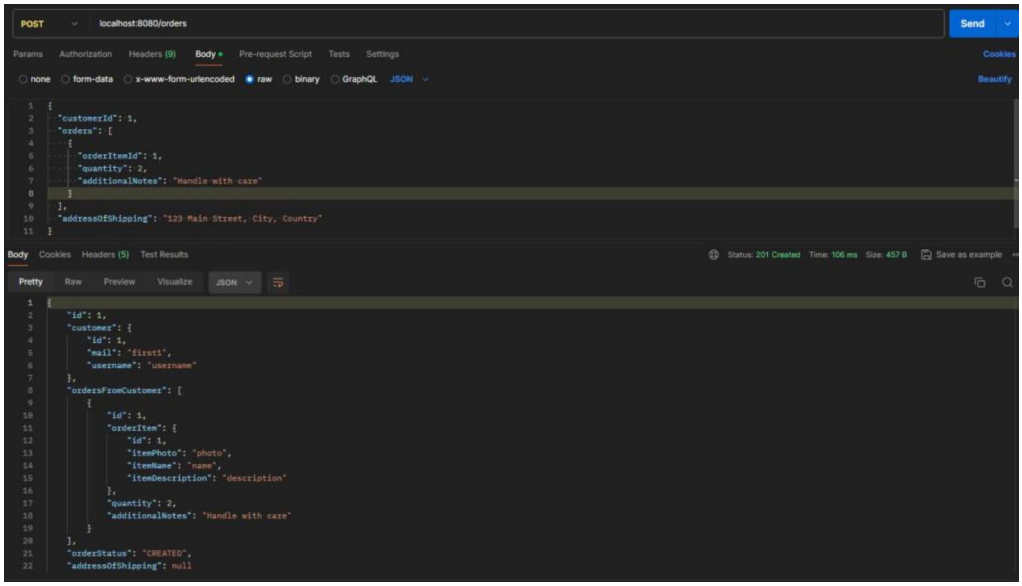


Рисунок В.1 – Перевірка створення замовлення для моделі виконаної за монолітною архітектурою

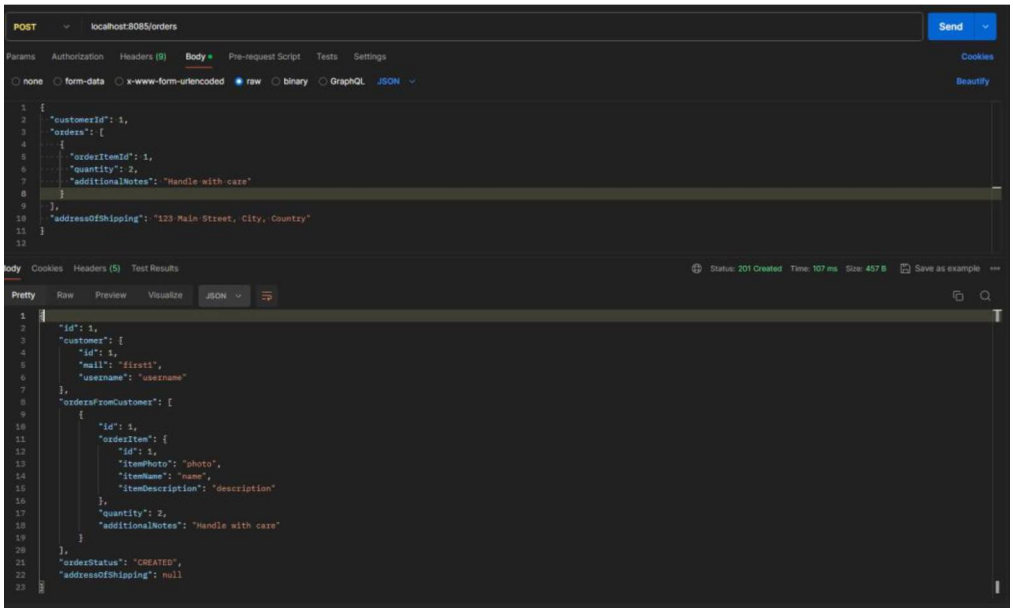


Рисунок В.2 – Перевірка створення замовлення для моделі виконаної за мікросервісною архітектурою

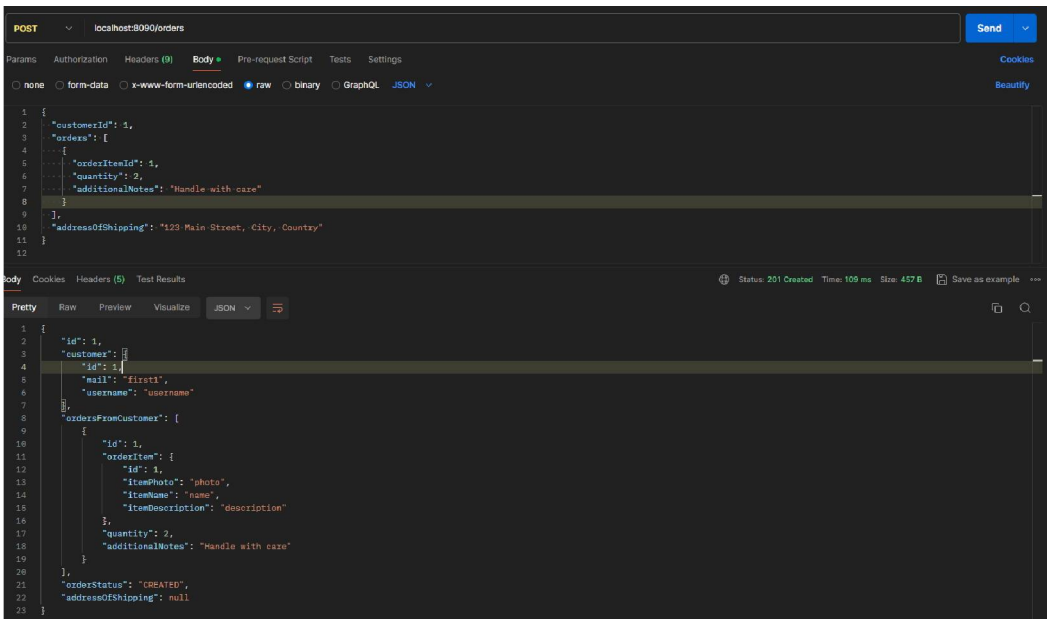


Рисунок В.3 – Перевірка створення замовлення для моделі виконаної за Cloud-Native архітектурою

Висновок: Перевірка відповідей розроблених моделей при однакових запитах виконано успішно, всі відповіді однакові, при однакових запитах.

Тест 2: Перевірка обрання архітектурнрнр підходу за ваговими коефіцієнтами, з максимальними коефіцієнтами простоти впровадження та медіани затримки. По правильності розрахунків робиться висновок про працездатність моделі (рисунок В.4).

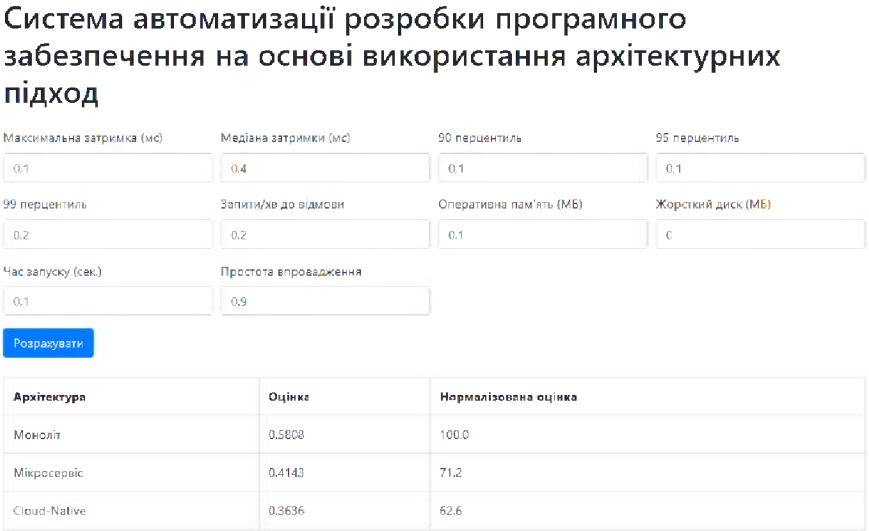


Рисунок В.4 – Перевірка моделі з максимальними коефіцієнтами простоти впровадження та медіани затримки

Висновок: Перевірка обрання архітектурного підходу пройшла успішно, так як .

Тест 3: Перевірка обрання архітектурнрнр підходу за ваговими коефіцієнтами, з максимальними коефіцієнтами простоти впровадження та медіани затримки. По правильності розрахунків робиться висновок про працездатність моделі (рисунок В.5).

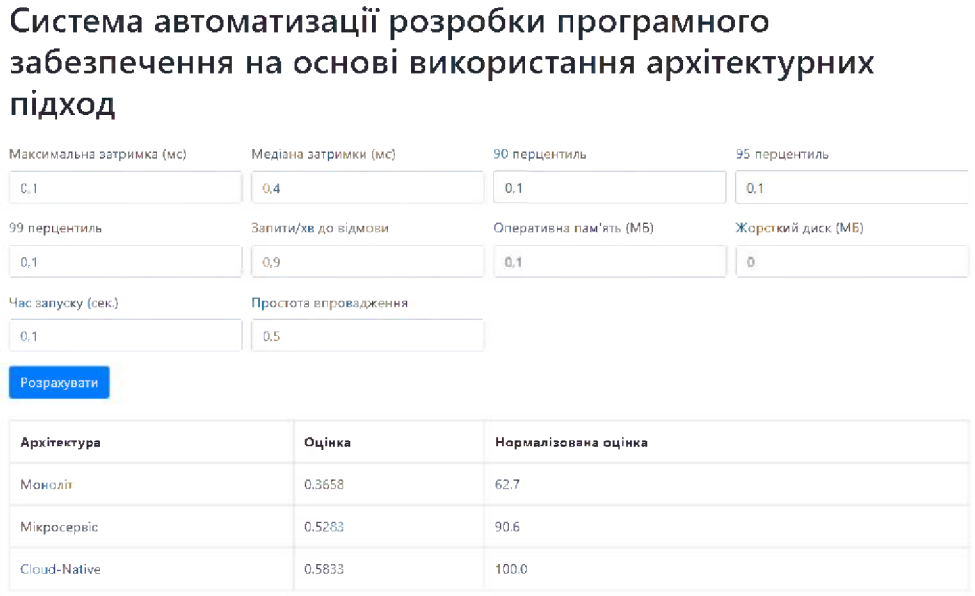


Рисунок В.5 – Перевірка моделі з максимальними коефіцієнтами запити/хв до відмови, та середньою простотою впровадження.

Висновок: Перевірка обрання архітектурного підходу пройшла успішно.

Висновки: при вдалому виконанні всіх 3 тестів випробування розробленого додатку вважаються успішними.

Виконавець
студент групи КУ-41
Чиняков Д.С.

Додаток Г

ФАЙЛИ КОНФІГУРАЦІЇ ДЛЯ РОЗГОРТАННЯ МІКРОСЕРВІСІВ У КЛАСТЕРІ KUBERNETES.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
 name: db-config-ms1
data:
 DB_HOST: db-ms1
 DB_USER: user
 DB_PASSWORD: password
 DB_NAME: database1

---
apiVersion: v1
kind: ConfigMap
metadata:
 name: db-config-ms2
data:
 DB_HOST: db-ms2
 DB_USER: user
 DB_PASSWORD: password
 DB_NAME: database2

---
apiVersion: v1
kind: ConfigMap
metadata:
 name: db-config-ms3
data:
 DB_HOST: db-ms3
 DB_USER: user
 DB_PASSWORD: password
 DB_NAME: database3
```

Рисунок Г.1 – db.creds.yml – файл для збереження конфігурація імені, паролів, назви для баз даних.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: db-ms1
spec:
 replicas: 1
 selector:
  matchLabels:
   app: db-ms1
 template:
  metadata:
   labels:
    app: db-ms1
  spec:
   containers:
   - name: postgres
    image: postgres:13
    env:
    - name: POSTGRES_DB
     valueFrom:
      configMapKeyRef:
       name: db-config-ms1
       key: DB_NAME
    - name: POSTGRES_USER
     valueFrom:
      configMapKeyRef:
       name: db-config-ms1
       key: DB_USER
    - name: POSTGRES_PASSWORD
     valueFrom:
      configMapKeyRef:
       name: db-config-ms1
       key: DB_PASSWORD
    ports:
    - containerPort: 5432
    volumeMounts:
    - mountPath: /var/lib/postgresql/data
     name: db-data
   volumes:
   - name: db-data
    emptyDir: {}
apiVersion: v1
kind: Service
metadata:
 name: db-ms1
spec:
 ports:
 - port: 5432
 selector:
  app: db-ms1
 type: ClusterIP
```

Рисунок Г.2 – db-deploy.yml – файл в якому описано три бази даних, які будуть використовувати параметри налаштувань використовуючи інформацію з db-creds.yml.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: orders
spec:
 replicas: 1
 selector:
  matchLabels:
   app: orders
 template:
  metadata:
   labels:
    app: orders
  spec:
   containers:
   - name: orders
    image: floppa/orders:1.0.0-SNAPSHOT
    imagePullPolicy: Never
    env:
    - name: DB_HOST
     valueFrom:
      configMapKeyRef:
       name: db-config-ms1
       key: DB_HOST
    - name: DB_USER
     valueFrom:
      configMapKeyRef:
       name: db-config-ms1
       key: DB_USER
    - name: DB_PASSWORD
     valueFrom:
      configMapKeyRef:
       name: db-config-ms1
       key: DB_PASSWORD
    - name: DB_NAME
     valueFrom:
      configMapKeyRef:
       name: db-config-ms1
       key: DB_NAME
    ports:
    - containerPort: 8080

apiVersion: v1
kind: Service
metadata:
 name: orders
spec:
 ports:
 - port: 8080
 selector:
  app: orders
 type: ClusterIP
```

Рисунок Г.3 – microservices.yml – файл в якому описано розгортання та параметри мікросервісів Orders, Items, Customers, з відповідними конфігруаціями баз даних.

Додаток Д

АПРОБАЦІЯ РЕЗУЛЬТАТІВ РОБОТИ.

Роботу було представлено на конференції VI Міжнародна студентська наукова конференція «НАУКА СЬОГОДЕННЯ: ВІД ДОСЛІДЖЕНЬ ДО СТРАТЕГІЧНИХ РІШЕНЬ» 10.05.2024 м. Чернігів, Україна

СЕКЦІЯ 14.
АВТОМАТИЗАЦІЯ ТА ПРИЛАДОБУДУВАННЯ

Чиняков Данило Сергійович, здобувач вищої освіти
факультету комп'ютерних наук
Харківський національний університет імені В.Н. Каразіна, Україна

Науковий керівник: Мороз Ольга Юріївна, PhD, старший викладач
ЗВО факультету комп'ютерних наук
Харківський національний університет імені В.Н. Каразіна, Україна

АВТОМАТИЗАЦІЯ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ ВИКОРИСТАННЯ АРХІТЕКТУРНИХ ПІДХОДІВ

В сучасному світі, де швидкість та ефективність виробництва програмного забезпечення набувають все більшої важливості, вибір правильної архітектури стає вирішальним фактором для успішної розробки та експлуатації програмних продуктів. Актуальність даної проблематики виходить за межі лише технічних розглядів, оскільки вона напряму впливає на ефективність бізнес-процесів, масштабованість, стабільність та інші ключові аспекти розробки програмного забезпечення.

Основна мета архітектури програмного забезпечення – забезпечити розробку додатку, який буде легким у розумінні, модифікації та підтримці, а також ефективно вирішувати поставлені завдання. Це включає в себе вибір правильних технічних рішень, створення оптимальної структури програми та забезпечення гнучкості для майбутніх змін.

Автоматизувати розробку програмного забезпечення на основі використання архітектурних підходів стало можливим систематизуя три модулі, що розроблені за допомогою трьох архітектурних підходів: модуль клієнтів, модуль інвентаризації та модуль замовлень. Після написання цих моделей було проведено тестування, і збір даних, які представлено в фінальній системи автоматизації, де можна обрати найбільш відповідний архітектурний підхід визначивши вагові коефіцієнти параметрів.

Однією з найвідоміших програмних архітектур є монолітна архітектура [1]. Для монолітної архітектури (рис. 1) усі три модулі будуть знаходитись в одній кодовій базі та використовувати одну спільну базу даних, в нашому випадку PostgreSQL.

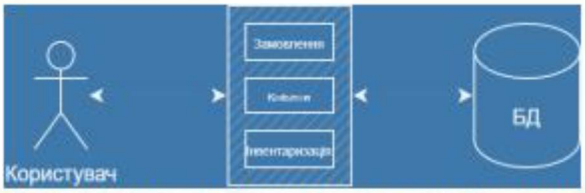


Рис. 1. **Схема додатку з використанням монолітної архітектури**

Мікросервіси – це автономні послуги, що розгортаються незалежно, з єдиною і чітко визначеною метою [2]. Тому для мікросервісної архітектури всі три модулі будуть знаходитись кожна в своїй кодовій базі, розгортатись незалежно, та використовувати три окремих бази даних, в нашому випадку PostgreSQL (рис. 2).

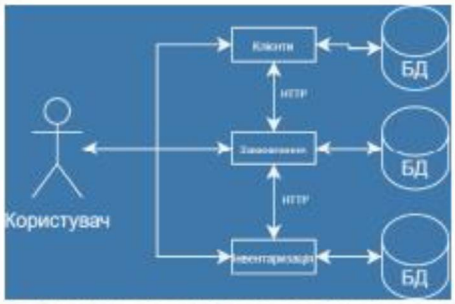


Рис. 2. **Схема додатку з використанням мікросервісної архітектури**

Для Cloud-Native архітектури (рис. 3) всі три модуля будуть знаходитись кожна в своїй кодовій базі, великим плюсом є те, що можна скористатися вже написаними модулями мікросервісної архітектури, внесши лише деякі зміни, використовуючи одну спільну базу даних, в нашому випадку PostgreSQL. Основна модифікація для цього архітектурного підходу: впровадження Docker-контейнерів та Kubernetes для оркестрації контейнерів, балансування навантаження, та впровадження реплікацій за потреби.

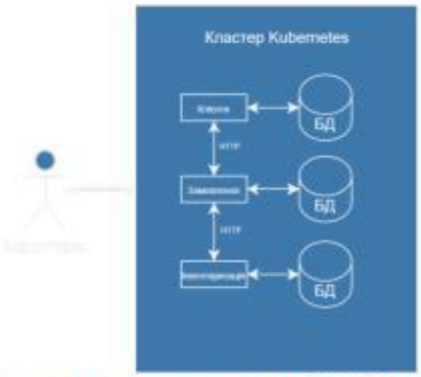


Рис. 3. **Схема додатку з використанням Cloud-Native архітектури**

Приступаючи до збору даних, враховуємо основні метрики, що будуть отримані за допомогою Jmeter – затримка (рис. 4), та пропускна здатність (рис. 5) додатків.

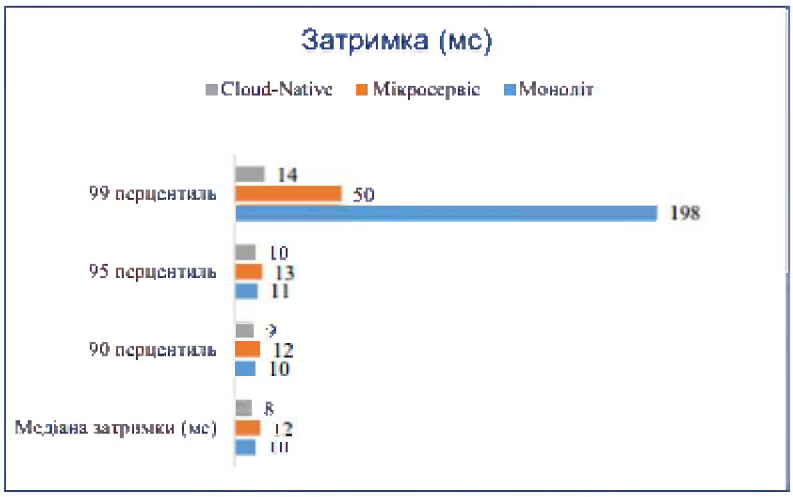


Рис. 4. **Статистика затримок (мс)**

Затримка означає час, необхідний для обробки однієї операції або запиту. Пропускна здатність, з іншого боку, вимірює обсяг операцій, які система може обробити за певний проміжок часу. Вона відображає здатність системи обробляти безліч запитів одночасно. Висока пропускна здатність свідчить про здатність системи ефективно керувати і масштабуватися зі збільшенням попиту [3].

Рис. 5. **Пропускна здатність (запитів в хвилину)**

За отриманими даними Cloud-Native архітектура перемогає за всіма параметрами як монолітну архітектуру, так і Cloud-Native. Але якщо прогнозовані навантаження на додаток не будуть перевищувати показники отримані в тестувані, то розгляд цих архітектур буде гарним рішенням, так як впровадження Cloud-Native архітектури, і всієї інфраструктури потребує багато часових та фінансових витрат.

Монолітна архітектура в той самий час потребує мінімальних витрат, і не вибаглива к інфраструктурі.

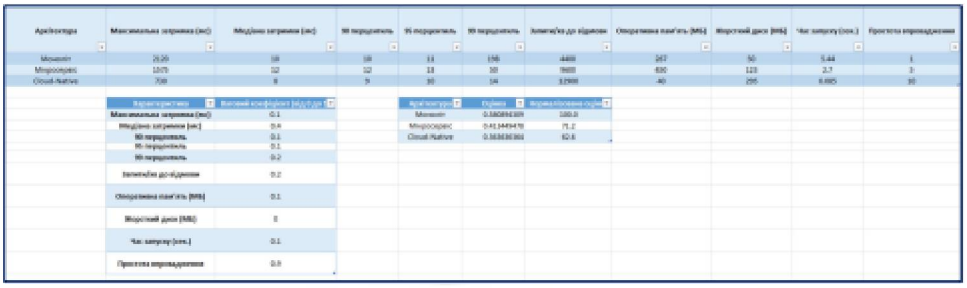


Рис. 6. **Зовнішній вигляд роботи моделі**

Спираючись на отримані параметри, було розроблено систему автоматизації розробки програмного забезпечення на основі використання архітектурних підходів (рис. 6).

Якщо обрати за найбільш вагомими парамстрами простоту впровадження та медіану затримки, тоді монолітна архітектура видає результат в 100 балів, що логічно, бо вона демонструє гарні результати – як і по медіанній затримці, так і найкращий результат за простотою впровадження.

Остаточний вибір архітектури має базуватися не тільки на технічних параметрах, але й враховувати довгострокову стратегію розвитку проєкту, можливості масштабування та вимоги до підтримки. Ця модель рекомендацій надає зрозумілі критерії для вибору, що може бути адаптованим до специфіки будь-якого проєкту, забезпечуючи високу адаптивність і гнучкість в управлінні програмними рішеннями.

Список використаних джерел:

1. Nada Salaheddin Elgheriani, Nuredin Ali Salem Ahmed. Microservices vs. Monolithic architectures. [The differential structure between two architectures]. MINAR International Journal of Applied Sciences and Technology. 2022. Vol.4, № 3, P.500-514. URL: https://www.minarjournal.com/dergi/microservices-vs-monolithic-architectures-the-differential-structure-between-two-architectures20221202031410.pdf (Last accessed: 02.01.2024).
2. Microservices a definition of this new architectural term. URL: www.martinfowler.com/articles/microservices.html (Last accessed: 25.12.2023).
3. Latency vs Throughput in Spring Boot Applications. Ionut Anghel. Dec 20, 2023. URL: https://levelup.gitconnected.com/latency-vs-throughput-in-spring-boot-applications-85bc68308752 (Last accessed: 25.02.2024).