**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ**

**Факультет Інформаційних технологій**

**УДК 004.8:004.3:004.738**

| **ПОГОДЖЕНО** | **ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ** |
|---|---|
| **Декан факультету** | **Завідувач кафедри** |
| Інформаційних технологій | Комп'ютерних систем, мереж та кібербезпеки |
| _____ Болбот І.М., д.т.н, проф. | _____ Касаткін Д.Ю., к. пед.н, доц. |
| підпис ПІБ, вчене звання і ступінь | підпис ПІБ, вчене звання і ступінь |
| «___»_____2024 р. | «___»_____2024 р. |

# МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

**На тему**: «Дослідження комп'ютерної системи управління бізнес процесами на основі інструментарію Spring»

Спеціальність: 123 «Комп’ютерна інженерія»

Освітня програма: Комп’ютерні системи і мережі

Орієнтація освітньої програми: Освітньо-професійна

**Гарант освітньої програми**

д.т.н., доцент _____ _ Шкарупило В. В.

(науковий ступінь та вчене звання) (підпис) (ПІБ)

**Керівник дипломного проекту**

д.т.н., доцент _____ _ Шкарупило В. В.

(науковий ступінь та вчене звання) (підпис) (ПІБ)

**Виконав** _____ _ Ткаченко В.В.

(підпис) (ПІБ студента)

**КИЇВ – 2024**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**
**БІОРЕСУРСІВ І**
**ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ**
**ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ**

**«ЗАТВЕРДЖУЮ»**
**завідувач кафедри**
комп'ютерних систем і мереж
_____ / Касаткін Д.Ю., к.п.н., доц./
підпис ПІБ, вчене звання і ступінь
«__»_____20___р.

# З А В Д А Н Н Я

## ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ РОБОТИ СТУДЕНТУ

Ткаченку Владиславу Віталійовичу
(прізвище, ім'я, по батькові)

Спеціальність (напрям підготовки) комп'ютерна інженерія
Освітня програма
Орієнтація освітньої програми

Тема магістерської роботи Дослідження засобів автоматизації обчислювальних процесів розподілених комп'ютерних систем

затверджена наказом ректора НУБіП України від "1" 11 2023 р. № 1999 С
Термін подання завершеної роботи на кафедру
Вихідні дані до магістерської роботи

Перелік питань, що підлягають дослідженню:

1.
2.
3.

Перелік графічного матеріалу (за потреби)

**Дата видачі завдання "_____"_____2023 р.**

**Керівник магістерської роботи**_____ Шкарупило В. В., д.т.н., доц.
( підпис ) (прізвище та ініціали)

**Завдання прийняв до виконання** Ткаченко В.В.
(підпис ) (прізвище та ініціали студента)

# КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів дипломного проєкту (роботи) | Строк виконання етапів проєкту ( роботи ) | Примітка |
|---|---|---|---|
| 1 | Аналіз предметної області | | Виконано |
| 2 | Проєктування системи | | Виконано |
| 3 | Реалізація системи | | Виконано |
| 4 | Тестування системи | | Виконано |
| 5 | Оформлення пояснювальної записки | | Виконано |

**Студент** _____ Ткаченко В.В
( підпис ) (ініціали та прізвище)

**Керівник проєкт (роботи)** _____ В.В. Шкарупило
( підпис ) (ініціали та прізвище)

# РЕФЕРАТ

Пояснювальна записка: 98 сторінок, містить 6 рисунків, 5 таблиць, 29 лістингів, використано 19 джерел інформації.

Об’єктом дослідження є процес розробки веб-додатків з використанням різноманітних архітектур та стилів для ефективної та цілеспрямованої розробки програмного продукту для управління бізнес процесів.

Метою роботи полягає у дослідженні різноманітних архітектур та стилів для ефективної та цілеспрямованої розробки програмного продукту для управління бізнес процесів.

Наукова новизна даної роботи полягає у тому, що вона спрямована на інтеграцію та аналіз різних архітектурних підходів у розробці програмного забезпечення. Особлива увага приділяється вивченню не лише технічних аспектів кожного підходу, а й їх впливу на гнучкість, масштабованість та простоту розробки та тестування систем.

Практична цінність цієї роботи полягає в тому, що вона надає конкретні рекомендації щодо вибору оптимального архітектурного підходу для реалізації програмного забезпечення залежно від його характеристик, вимог бізнесу та можливостей команди розробників.

**ЗМІСТ**

# ВСТУП

Сучасний розвиток інформаційних технологій сприяє глобальній цифровій трансформації бізнесу, що зумовлює зростання попиту на автоматизацію управління бізнес-процесами. Ефективне управління бізнес-процесами стає ключовою конкурентною перевагою для підприємств, оскільки дозволяє оптимізувати ресурси, підвищувати продуктивність та знижувати витрати. У зв'язку з цим все більше компаній інтегрують спеціалізовані комп'ютерні системи, що дозволяють автоматизувати щоденні операції та забезпечують гнучкість бізнес-процесів.

Серед різноманітних технологічних платформ для розробки таких систем одним із провідних рішень є інструментарій Spring. Spring забезпечує широкий спектр інструментів для розробки надійних, масштабованих та ефективних веб-додатків, що задовольняють сучасні вимоги до продуктивності та безпеки. Завдяки своїй модульній архітектурі та підтримці різних архітектурних підходів, таких як мікросервіси, монолітні та подієві архітектури, Spring дозволяє розробникам створювати адаптивні системи, що легко інтегруються у будь-яке середовище.

Актуальність даного дослідження полягає у необхідності впровадження гнучких та продуктивних систем управління бізнес-процесами, які б відповідали потребам швидкозмінного ринкового середовища. Обраний інструментарій Spring надає не тільки гнучкість у розробці системи, але й можливість її адаптації до нових вимог завдяки підтримці різних архітектурних підходів.

Метою даної роботи є дослідження можливостей використання інструментарію Spring для створення комп'ютерної системи управління бізнес-

процесами з акцентом на вибір оптимальних методів розробки веб-додатків та архітектурних підходів.

Завдання дослідження:

1. Розглянути методи розробки веб-додатків та визначити їх особливості при побудові систем управління бізнес-процесами.
2. Дослідити архітектурні підходи та вибрати оптимальні рішення для створення адаптивної та масштабованої системи.
3. Розробити прототип комп’ютерної системи управління бізнес-процесами з використанням Spring, інтегрованої з базою даних MySQL.
4. Провести тестування розробленої системи, визначити її продуктивність та надати рекомендації щодо оптимізації.

Об'єктом дослідження є комп'ютерні системи управління бізнес-процесами, а предметом – методи та архітектури, що використовуються для їх розробки з використанням Spring.

У результаті виконання даної роботи передбачається отримати розроблений прототип системи управління бізнес-процесами, що буде базуватися на інструментах Spring, та визначити перспективи його вдосконалення і масштабування для розв'язання практичних завдань підприємств.

# 1 ОПИС МЕТОДІВ РОЗРОБКИ ВЕБ ДОДАТКІВ

У 1988 році Інтернет було оновлено, щоб забезпечити швидкий процес і різноманітні інтегровані функції [1], починаючи з ери настільних програм і продовжуючи до створення веб-додатків і мобільних додатків. Насправді, веб-додаток – це програма, до якої можна отримати доступ із браузера з комп'ютера чи мобільного пристрою будь-де та будь-коли [2] [3]. Швидке зростання програмної інженерії забезпечує величезні переваги для користувачів, а також для корпорації, особливо для зменшення витрат часу та витрат на розробку в поточному виробництві. Багато видів бізнесу роблять цю технологію основною вимогою для побудови системи для підтримки бізнес-функцій і процесів, що працюють належним чином і працюють краще порівняно з попередньою, яка в основному використовувалася з 1999 року. Еволюція надзвичайно швидка, що краще назвати революцією , було забезпечено швидкий досвід за якісного та надійного підключення до Інтернету через логічний механізм, який називається серверною частиною, та інтерфейс користувача програми, який називається зовнішнім інтерфейсом. Коротше кажучи, візуалізація на стороні клієнта динамічно керує маршрутизацією без оновлення сторінки кожного разу, коли користувач запитує інший маршрут, але, з іншого боку, рендеринг на стороні сервера може відображати повністю завантажену сторінку на першому для будь-якого веб-сайту, тоді як візуалізація на стороні клієнта спочатку показує порожню сторінку [5]. Одним із видів бізнесу, який реалізував рендеринг на стороні клієнта, є електронна комерція, щоб забезпечити оптимізовану взаємодію з користувачем для завантаження та ефективний план підтримки спілкування та взаємодії [6] [7]. Мабуть, програмне забезпечення — це лише інструменти, які допомагають організації виконувати свій бізнес-план від початку до кінця з конкретними обмеженнями бізнесу, обмеженням

бюджету та ресурсів із високим очікуванням прибутку в кінці. Говорячи про ефективність програмного забезпечення, існує багато аспектів, які визначають, чи можна його назвати хорошим чи правильним програмним забезпеченням у контексті. На початку 2000 року багато програмного забезпечення використовували метод відтворення на стороні сервера, тобто процес, який повністю запускався на стороні сервера, тоді як клієнт отримував відповідь лише як отриману сторінку. З цього моменту концепція на стороні сервера розвинулася

Існує кілька підходів розробки веб-додатків, кожен з яких має свої переваги та недоліки. Ось огляд деяких з них.

У процесі розробки програмного забезпечення належним планом є не лише аналіз проблеми та проектування програмного забезпечення, але й вибір правильної архітектури в контексті мети та обсягу дослідження. У цьому документі порівняння зосереджено на прикладному рівні (сервісі) як одному з чотирьох рівнів багатошарового шаблону в архітектурі програмного забезпечення, окрім презентації (UI), бізнес-логіки (домен) і доступу до даних (персистентності), який має функцію компонування можливостей представити інтерфейс вмісту візуально та графічно, щоб створити взаємодію між користувачем і додатком через надання проміжного програмного забезпечення та керування. Зазвичай прикладний рівень запускається в режимі діагностики або в режимі користувача, який можна налаштувати відповідно. Важливо вибрати правильний тип архітектурних шаблонів, щоб оптимізувати використання та створити багаторазове рішення, щоб передбачити загальну проблему, яка виникає під час реалізації або в даному контексті. Крім багаторівневого шаблону, існують клієнт-сервер, головний-підлеглий, канал-фільтр, брокер, одноранговий, шина подій, модель-контролер перегляду, дошка та шаблон інтерпретатора [22]. Коротше кажучи, багаторівневий шаблон можна використовувати для організації програм, які складаються з кількох груп підзадач, кожна з яких знаходиться на певному рівні абстракції, що

представляє конкретні служби для верхнього рівня. Практично візуалізація на стороні сервера – це метод розробки програмного забезпечення у веб-додатку, який обробляє запити від користувача на сервері, а потім сервер виконує певну логіку та алгоритми відповідно до потреб бізнесу. Після обробки запитів сервер надсилає відповідь користувачеві в результаті роботи майже з процесом візуалізації, що називається багатосторінковим додатком, оскільки, коли користувач надсилає запит, він працює на сервері, а не на пристрої користувача [8].

У розробці процесу програмного забезпечення належним планом є не тільки аналіз проблем та проектування програмного забезпечення, але й вибір правильної архітектури в контексті мети та обсягу дослідження. У цьому документі порівняння зосереджено на прикладному рівні (сервісі) як одному з чотирьох рівнів багатошарового шаблону в архітектурі програмного забезпечення, окрім презентації (UI), бізнес-логіки (домен) і доступу до даних (персистентності), який має компонування компонентів, що дозволяють представити інтерфейс вмісту візуально та графічно, щоб створити взаємодію між користувачами та додатком через надання проміжного програмного забезпечення та керування. Зазвичай прикладний рівень запускається в режимі діагностики або в режимі користувача, який можна налаштувати відповідно. Важливо вибрати правильний тип архітектурних шаблонів, щоб оптимізувати використання та створити багаторазові рішення, щоб передбачити загальну проблему, яка сталася під час реалізації або в даному контексті. Крім багаторівневого шаблону, клієнт-сервер, головний-підлеглий, канал-фільтр, брокер, одноранговий, шина подій, модель-контролер перегляду, дошка та шаблон інтерпретатора [22]. Коротше кажучи, багаторівневий шаблон можна використовувати для організації програм, які складаються з декількох груп підзадач, кожна з яких знаходиться на певному рівні абстракції, що представляє конкретні служби для верхнього рівня. Практична візуалізація на стороні сервера – це метод розробки програмного забезпечення у веб-додатку, який обробляє запити від користувачів на сервері, а потім сервер виконує певну логіку

та алгоритми відповідно до потреб бізнесу. Після обробки запитів сервер надсилає відповідь користувачам в результаті роботи майже з процесом візуалізації, що називається багатосторінковим додатком, потім, коли користувач надсилає запит, він працює на сервері, а не на пристрої користувача [8].

Спочатку користувач, взаємодіючи з відповідним додатком, створює запит. Якщо програма не використовує сторонній плагін, алгоритм автоматично оброблятиме його на клієнтському пристрої, інакше API оброблятиме можливість перенесення з сервера. Завдяки цій концепції він створить процес додатків із скороченням часу та ресурсів, але швидшим завдяки розподілу процесу алгоритму між сервером і клієнтським пристроєм, а також відповіді сервера у вигляді даних, відомих як JSON. Зазвичай відповідь була стиснута в комплекті з пакетами невеликого розміру, тому процес завантаження буде швидшим через його розмір. Останній процес у видобутку призводить до розпакування завантаженого комплекту, який запускається на клієнтському пристрої, що залежить від можливостей пристрою користувача. Існує багато мов програмування, які можуть реалізувати концепцію на стороні клієнта. Подібним чином JavaScript є мовою програмування, яка широко поширилася в очах розробників у різноманітності та різноманітності веб-створення. Таким чином, за допомогою AJAX сценарій Java досяг успіху для побудови багатьох динамічних показників і продуктивності в реальному часі, які Amazon, Gmail і Facebook реалізували цей тип механізму [14].

## 1.1 Традиційне відображення на стороні сервера

Традиційний серверний рендеринг (SSR) є одним із базових підходів у веб-розробці, що передбачає генерацію HTML-контенту на сервері у відповідь на запити від клієнта. Цей метод відрізняється від клієнтського рендерингу (CSR), де браузер користувача відповідає за створення HTML динамічно на стороні клієнта, використовуючи JavaScript-фреймворки, такі як React або Angular. Серверний рендеринг був основним підходом на початкових етапах розвитку Інтернету, коли кожен запит від клієнта призводив до генерації нової HTML-сторінки на сервері, яка потім надсилалася назад до клієнта. У цій статті розглядаються технічні особливості SSR, його переваги та обмеження, історичний розвиток, а також сучасні гібридні рішення, що поєднують переваги SSR і CSR.

Основною сутністю SSR є те, що кожен запит від клієнта супроводжується створенням HTML на сервері. Це означає, що щоразу, коли користувач переходить на нову сторінку або оновлює існуючу, сервер отримує запит, обробляє його і надсилає назад повністю відрендерену HTML-сторінку. Ця HTML-сторінка може включати динамічний контент, дані з баз даних або інші елементи, що налаштовуються залежно від контексту запиту. На відміну від CSR, де браузеру потрібно завантажити JavaScript-пакет і рендерити HTML в браузері, SSR одразу надсилає повноцінний HTML-документ, що дозволяє браузеру відображати вміст миттєво, без додаткових обчислень для генерації HTML [3].

Історично, SSR був єдиним методом рендерингу веб-контенту, оскільки технології, що дозволяють обробляти JavaScript на стороні клієнта, ще не були достатньо розвиненими. Веб-сайти будувалися з використанням серверних технологій, таких як PHP, Ruby on Rails та ранні версії ASP.NET. Ці платформи дозволяли генерувати динамічний контент на сервері, що давало змогу

персоналізувати сторінки залежно від користувацьких параметрів, локації або сесійних даних. Так, в середині 2000-х SSR став стандартом для створення веб-застосунків, які активно використовувалися для контентно-орієнтованих веб-сайтів і порталів [5].

Серверний рендеринг має ряд переваг, особливо у контексті покращення досвіду користувача та оптимізації для пошукових систем. Оскільки сервер доставляє повністю сформований HTML-документ, користувачі відчувають швидші завантаження сторінок, оскільки браузеру потрібно лише інтерпретувати HTML і відображати його. Це особливо важливо для користувачів з повільним інтернетом або обмеженими обчислювальними можливостями. Крім того, пошукові системи, такі як Google, легше індексують SSR-контент, оскільки їм не потрібно виконувати JavaScript для відображення контенту сторінки [7].

З технічної точки зору, SSR працює на основі синхронної обробки запитів, що означає, що кожен клієнтський запит вимагає завершення процесу рендерингу HTML на сервері перед відправленням відповіді. Це може призводити до затримок, якщо сервер перевантажений або якщо процес обробки запиту включає складні обчислення або запити до бази даних. Щоб уникнути цього, розробники часто використовують кешування для зберігання відрендерених HTML для часто запитуваних сторінок, що дозволяє швидко обслуговувати сторінки без необхідності повторної генерації HTML [8].

З розвитком веб-технологій SSR почав інтегруватися з новими фреймворками JavaScript, такими як Next.js для React і Nuxt.js для Vue, які дозволяють створювати гібридні рішення. Такі фреймворки комбінують переваги SSR і CSR, забезпечуючи швидке завантаження початкової сторінки, яке досягається за допомогою SSR, і плавні переходи на сторінки, які оновлюються на стороні клієнта. Цей підхід відомий як "універсальний" або "ізоморфний" рендеринг, де деякі частини

програми рендеряться на сервері, а інші - на клієнті, що забезпечує продуктивність і одночасно зберігає інтерфейс, гнучкий для інтерактивності [9].

Основною проблемою SSR є збільшення навантаження на сервер у порівнянні з CSR. Оскільки кожен запит супроводжується рендерингом HTML на сервері, застосунки на базі SSR можуть вимагати потужнішої серверної інфраструктури або оптимізованого управління сервером для обробки великої кількості трафіку. Для зниження навантаження розробники використовують такі методи, як балансування навантаження, кешування та мережі доставки контенту (CDN), які допомагають розподілити навантаження на сервери або географічно наблизити контент до користувачів [11].

SSR забезпечує більш легкий доступ до контенту для користувачів з обмеженими можливостями, оскільки весь вміст одразу доступний у форматі HTML. Це особливо важливо для екранних читалок та інших допоміжних технологій, які залежать від добре структурованого HTML для інтерпретації та передачі контенту. У випадку SSR весь необхідний контент вбудовується безпосередньо в HTML і не залежить від JavaScript, що робить його більш доступним. Це значно покращує доступність для користувачів з особливими потребами, забезпечуючи миттєвий доступ до контенту без ризику, що JavaScript не завантажиться[12].

Попри численні переваги, серверний рендеринг має й свої обмеження. Одним з основних викликів SSR є управління затримками, які можуть виникати через час відповіді сервера. Оскільки сервер відповідає за генерацію HTML в реальному часі, будь-яка затримка в обробці запиту може призвести до збільшення часу завантаження сторінки. Це особливо важливо для застосунків з великим навантаженням або складною серверною логікою, де кожне відображення контенту може бути залежним від бази даних або API-запитів [13].

Крім того, SSR вимагає структурованої архітектури додатку, здатного функціонувати як на сервері, так і на клієнті. Це додає складності у розробці, оскільки програмісти повинні враховувати різні середовища виконання та бути обережними у використанні API браузера або сторонніх бібліотек, які можуть не працювати на сервері. Для забезпечення стабільності та відповідної роботи таких програм необхідні додаткові тестування та налагодження [14].

Традиційний серверний рендеринг залишається одним із ключових підходів у веб-розробці, що має значні переваги в продуктивності, SEO та доступності. Завдяки злиттю з сучасними технологіями SSR продовжує адаптуватися до вимог сучасної веб-розробки, пропонуючи потужні гібридні рішення для створення продуктивних і динамічних застосунків. Численні компанії, що зосереджені на швидкості доставки контенту, продовжують використовувати SSR для забезпечення відмінного досвіду користувача та поліпшення видимості в пошукових системах [3].

## 1.2 Візуалізація на стороні клієнта

Відображення на стороні клієнта (Client-Side Rendering) - це підхід до розробки веб-додатків, при якому весь процес генерації і відображення контенту відбувається на боці клієнта, у веб-браузері. Замість того, щоб сервер генерував і відправляв готовий HTML-код, сервер надсилає клієнту лише дані, які зазвичай представлені у форматі JSON або XML. Після цього клієнтська частина додатку, використовуючи JavaScript, відповідальна за рендерінг цих даних у вигляді сторінки, яку бачить користувач.

Візуалізація на стороні клієнта (скорочено CSR) — це метод відтворення JavaScript, який використовує JavaScript для відтворення веб-сайту або програми в браузері. За допомогою CSR обробка та рендеринг вмісту відбувається в браузері, а не на сервері.

За допомогою CSR сервер надсилає barebone-файл HTML із посиланнями на файли JavaScript. Потім браузер завантажує необхідні ресурси та використовує їх для заповнення сторінки та відтворення вмісту.

CSR зазвичай використовується для програм, які мають динамічний вміст і вимагають високого рівня інтерактивності, як-от програми для чату та платформи соціальних мереж. Він також ідеально підходить для односторінкових програм (SPA) і внутрішніх програм, таких як інформаційні панелі адміністратора та користувачів, які не потребують індексації пошуковими системами.

Фреймворки JavaScript, такі як React, Vue.js, Angular, Backbone.js, Ember.js і Svelte, стали популярними виборами для реалізації візуалізації на стороні клієнта.

Ось розбивка того, як працює візуалізація на стороні клієнта, і етапи процесу візуалізації:

— Користувач запитує веб-сторінку: користувач запитує веб-сторінку в браузері.

— Сервер отримує запит: ця дія користувача запускає запит GET на сервер, який визначає потрібний маршрут або URL-адресу.

— Сервер надсилає мінімальну HTML-сторінку з посиланнями на файли CSS і JavaScript: браузер запитує початковий файл HTML у сервера. Файл містить посилання на зовнішні таблиці стилів CSS і файли JavaScript.

— Розбір HTML: браузер аналізує розмітку HTML і створює дерево об’єктної моделі документа (DOM), яке представляє структуру веб-сторінки.

— Браузер завантажує CSS і JavaScript: браузер надсилає додаткові запити на сервер для завантаження ресурсів CSS і JavaScript.

— Відтворення сторінки: браузер використовує дерево DOM і завантажені файли CSS для відтворення базової структури веб-сторінки. Це включає такі елементи, як текст, зображення, кнопки та стилі.

Виконання JavaScript: браузер виконує код JavaScript, щоб додати веб-сторінці інтерактивність і динамічний вміст. Це може включати анімацію, перевірку форм або отримання даних з API.

Повторне відображення та оновлення: коли користувач взаємодіє з веб-сторінкою (клацає кнопки, заповнює форми), браузер повторно відтворює частини веб-сторінки відповідно до дій користувача. Це може включати оновлення DOM або надсилання додаткових запитів до сервера для нових даних.

Остаточний відображення: оновлений DOM разом із будь-якими динамічно отриманими даними відображає браузер, у результаті чого створюється повністю інтерактивна та динамічно оновлювана веб-сторінка.

Візуалізація на стороні клієнта переносить значну частину процесу візуалізації з сервера на браузер клієнта. Оскільки серверу потрібно надіслати лише мінімальну кількість HTML і статичних ресурсів, він відчуває менші витрати на обробку та зменшує навантаження на сервер.

При візуалізації на стороні сервера сервер повинен генерувати повний HTML для кожного запиту, що потенційно створює навантаження на ресурси сервера, особливо в періоди високого трафіку. Використовуючи рендеринг на стороні клієнта, сервер може зосередитися на обробці пошуку даних, бізнес-логіки та запитів API, що призводить до покращеної масштабованості та кращого використання ресурсів.

Ось кілька способів CSR покращити взаємодію користувачів:

За допомогою CSR взаємодії користувача, такі як натискання кнопок, надсилання форм або навігація між сторінками, обробляються у веб-переглядачі без необхідності повного перезавантаження сторінки. Програми CSR можуть реагувати на дії та введення користувачів у режимі реального часу. Це призводить до майже

миттєвих оновлень, покращує взаємодію з користувачем і робить веб-сайти більш чуйними

CSR-фреймворки, такі як React, Vue.js і Angular, використовують ефективні методи візуалізації, такі як Virtual DOM, щоб забезпечити виконання складних оновлень без будь-яких помітних затримок.

Візуалізація на стороні клієнта може призвести до довшого часу завантаження сторінки, ніж візуалізація на стороні сервера, оскільки браузер має завантажити та виконати всі необхідні файли JavaScript, перш ніж відтворити всю веб-сторінку.

Такі пошукові системи, як Google і Bing, покращили індексацію веб-сайтів, завантажених JavaScript. Однак роботам і сканерам все ще легше індексувати веб-сайти, відтворені на стороні сервера, ніж веб-сайти на стороні клієнта.

Веб-сайти, відображені на клієнті, можуть бути не так легко виявлені в результатах пошуку, що може негативно вплинути на їх видимість, органічний трафік і коефіцієнти конверсії. Через це візуалізація на стороні клієнта є неправильним підходом для таких веб-сайтів, як цільові сторінки, блоги, медіа-публікації та платформи електронної комерції, які вимагають високої видимості в пошукових системах.

Візуалізація на стороні клієнта значною мірою залежить від обчислювальної потужності пристрою користувача. Якщо пристрій старіший або не має достатньо ресурсів, йому може бути важко впоратися з відтворенням і виконанням складних фреймворків або бібліотек JavaScript.

Ця залежність від пристрою користувача може призвести до неузгодженої продуктивності на різних пристроях і веб-переглядачах. Ми можемо вирішити цю проблему, запровадивши стратегії оптимізації продуктивності, такі як відкладене завантаження, оптимізація активів і поділ коду.

Під час візуалізації на стороні клієнта браузер спочатку отримує мінімальний файл HTML, що містить посилання на файли JavaScript. Потім браузер завантажує

код JavaScript, отримує всі необхідні дані з API і динамічно відображає вміст. З іншого боку, при рендерингу на стороні сервера сервер генерує повну розмітку HTML для сторінки та надсилає її браузеру для рендерингу.

Під час візуалізації на стороні клієнта користувачі зазвичай стикаються з порожньою сторінкою або функцією завантаження, поки браузер отримує файли JavaScript, аналізує їх і відображає вміст. Відсутність видимого вмісту під час початкового завантаження може створити відчуття повільності або невідповідності, що потенційно може призвести до розчарування або покинутості.

Завдяки рендерингу на стороні сервера користувач одразу бачить вміст веб-сайту під час завантаження сторінки, оскільки браузер отримує повністю відрендерену розмітку HTML. Це покращує взаємодію з користувачем, оскільки користувачі одразу взаємодіють із вмістом без початкового порожнього екрана чи індикаторів завантаження.

Візуалізація на стороні клієнта допомагає розробникам створювати динамічні та інтерактивні веб-сайти та програми. Підхід CSR до використання JavaScript і переміщення процесу візуалізації у веб-переглядач допомагає нам створювати адаптивні додатки, які пропонують нативний досвід.

Хоча рендеринг на стороні клієнта не є універсальним рішенням, він став цінним інструментом веб-розробки. Розуміння його сильних сторін, обмежень і відповідних випадків використання допоможе нам зрозуміти, коли рендеринг на стороні клієнта є правильним підходом.

## 1.3 Візуалізація на стороні сервера з інтерактивністю на стороні клієнта

По-перше, що таке рендеринг? Це перетворення коду на веб-сторінку, яку користувачі бачать у своїх браузерах. Дві відомі техніки візуалізації, які домінують у веб-розробці, це SSR і CSR. Кожен із них пропонує унікальні переваги та відповідає конкретним потребам веб-розробки. У CSR більша частина візуалізації відбувається на клієнті, тобто в браузері з Javascript. У SSR візуалізація відбувається на сервері. Прикладом є використання будь-якої серверної технології, як-от Node.js, PHP тощо.

Цей підрозділ має на меті навчити вас тонкощам SSR і CSR і повідомити вас про тонкощі SSR і CSR, позначивши ключові відмінності, корисність і недоліки, відомі їм, і вибрати той, який буде оптимальним для їхніх конкретних проектів веб-розробки.

Давайте глибше зануримося в те, як працює рендеринг на стороні сервера та клієнта. Почнемо з PCP.

Візуалізація визначає, як HTML, CSS і JavaScript веб-сторінки обробляються для створення сторінки для перегляду. Два основних підходи до візуалізації — це рендеринг на стороні сервера (SSR) і рендеринг на стороні клієнта (CSR). За допомогою SSR сервер створює повний вміст HTML для кожної сторінки та надсилає його в браузер, щоб користувачі швидко бачили вміст. CSR передбачає отримання браузером мінімального файлу HTML і використання JavaScript для динамічного відтворення вмісту.

Вибір відповідного методу візуалізації впливає на продуктивність програми, оптимізацію пошукової системи (SEO) і взаємодію з користувачем. SSR може забезпечити швидше початкове завантаження сторінок і кращий SEO, оскільки вміст одразу стає доступним для сканування пошуковими системами. CSR

пропонує більшу інтерактивність і плавніші переходи всередині програми після початкового завантаження.

Розуміння компромісів між SSR і CSR допоможе узгодити стратегію візуалізації з конкретними потребами вашого проекту. На ваше рішення мають впливати такі фактори, як тип вмісту, вимоги до продуктивності, міркування про пошукову оптимізацію, використання безголовної CMS і цільова аудиторія.

HTTP-запит надсилається на сервер, який обробляє запити та генерує вміст (який може мати HTML, JSON або інші формати). Згенерований вміст разом із необхідним Javascript потім надсилається назад у браузер клієнта.

У серверному рендерингу (SSR) початковою відповіддю може бути попередньо відрендерована HTML-сторінка, яка вдосконалюється за допомогою JavaScript на стороні клієнта.

Однією з основних переваг візуалізації на стороні сервера є швидкість завантаження сторінки. Це дуже важливий критерій для досвіду користувача.

Враховуючи це, ми бачимо, що в SSR важкі завдання виконуються на сервері. Тоді це зменшує робоче навантаження на браузер (клієнт), оскільки для браузера практично не залишається роботи. Він готовий до використання на льоту.

Покращена оптимізація пошукових систем (SEO). Це полегшує пошуковим системам сканування та індексування вмісту вашого веб-сайту, оскільки він уже відображається у форматі HTML. Це особливо важливо для сайтів із динамічним вмістом.

Швидке завантаження сторінки: SSR надсилає повністю відтворену сторінку в браузер, яка швидко відтворюється та відображається користувачам у браузері.

Узгодженість доставки вмісту та краща доступність: SSR гарантує, що всі користувачі отримають однаковий вміст незалежно від можливостей пристрою чи браузера.

Збільшене навантаження на сервер. Залишення основної частини візуалізації виключно для сервера може потребувати ресурсів, особливо для веб-сайтів із високим динамічним вмістом

Повільніші оновлення: зміни вмісту не відтворюються/відображаються на сторінці автоматично. Це вимагає повного перезавантаження сторінки, що потенційно може вплинути на взаємодію з користувачем. Це також робить сторінку менш інтерактивною. Це залишає м’який досвід користувача.

Серверний рендеринг із клієнтською інтерактивністю (SSR з інтерактивністю на клієнті) є сучасним підходом у веб-розробці, який поєднує переваги швидкого завантаження сторінок, характерного для серверного рендерингу (SSR), з інтерактивними елементами, що забезпечуються клієнтським рендерингом (CSR). Цей гібридний метод став популярним завдяки JavaScript-фреймворкам, таким як Next.js і Nuxt.js, які дозволяють одночасно використовувати серверний рендеринг для початкового завантаження сторінки та клієнтський рендеринг для оновлення та обробки взаємодій без повного перезавантаження сторінки. Нижче детально розглянуто технічні аспекти, переваги, виклики та основні методи реалізації SSR з клієнтською інтерактивністю.

У традиційних підходах SSR та CSR кожен має свої сильні сторони: SSR забезпечує швидке завантаження сторінок та покращує видимість у пошукових системах, тоді як CSR дозволяє створювати динамічні та інтерактивні інтерфейси. SSR з клієнтською інтерактивністю поєднує ці підходи, надаючи повністю відрендерену HTML-сторінку з сервера для початкового завантаження, а потім додаючи JavaScript для забезпечення інтерактивності на стороні клієнта. Коли користувач вперше відвідує сторінку, сервер обробляє запит, генерує HTML і надсилає його клієнту, що дозволяє браузеру миттєво відобразити вміст. JavaScript-код потім активується на стороні клієнта для забезпечення інтерактивності, як-от анімацій, форм з автозаповненням та динамічного оновлення даних без повного перезавантаження сторінки.

Однією з ключових переваг SSR з клієнтською інтерактивністю є значне покращення вражень користувача від початкового завантаження. Оскільки сторінка рендериться на сервері, користувач бачить повністю завантажену HTML-сторінку, навіть якщо JavaScript ще не виконався, що є корисним на повільних інтернет-з'єднаннях або пристроях з обмеженими ресурсами. Як тільки JavaScript завантажується, він додає інтерактивні функції, що забезпечує швидкі і плавні взаємодії з елементами сторінки.

Ще однією суттєвою перевагою є покращення SEO. Оскільки початковий HTML вже містить необхідний контент, пошукові системи можуть легко індексувати сторінку без необхідності виконання JavaScript. Це робить SSR з клієнтською інтерактивністю ідеальним для контентно-орієнтованих сайтів, таких як новинні портали, блоги та інтернет-магазини, де важливе швидке відображення контенту для користувачів і пошукових систем.

Технічно SSR з інтерактивністю на клієнті реалізується через процес, відомий як «гідратація». Гідратація є процесом, у якому JavaScript «під'єднується» до відрендереного HTML, перетворюючи його на інтерактивний інтерфейс. Коли браузер завантажує HTML, він також завантажує пов'язаний JavaScript, який оживляє сторінку, дозволяючи користувачу взаємодіяти з нею без необхідності перезавантажувати весь HTML. Це забезпечує не лише продуктивність, а й високий рівень інтерактивності, необхідний для сучасних веб-додатків.

При реалізації SSR з клієнтською інтерактивністю важливо враховувати асинхронну обробку даних і розділення JavaScript-коду. Сучасні інструменти, такі як Webpack і Vite, дозволяють розбивати JavaScript-код на менші частини (так звані «chunks»), які завантажуються тільки тоді, коли вони потрібні, знижуючи початкове навантаження на браузер і покращуючи продуктивність.

Попри численні переваги, поєднання SSR та клієнтської інтерактивності має певні виклики. Одним із таких викликів є складність в реалізації. Оскільки додаток має бути здатним працювати як на сервері, так і на клієнті, розробники повинні

забезпечити, щоб код, який виконується на сервері, був сумісним з клієнтським середовищем. Наприклад, деякі браузерні API (такі як window або document) недоступні на сервері, що може викликати помилки під час рендерингу. Розробникам потрібно враховувати такі особливості і використовувати перевірки середовища, щоб уникнути проблем під час гідратації.

Інший виклик полягає у витратах на інфраструктуру. Оскільки кожен запит потребує обробки на сервері, цей підхід може збільшити навантаження на сервери порівняно з CSR, де більша частина навантаження припадає на клієнт.

## 1.4 Односторінкові програми

Односторінкові програми (SPA) є всюди. Навіть якщо ви точно не впевнені, що це таке, ви, швидше за все, використовуєте їх регулярно — вони є чудовим інструментом для створення неймовірно привабливих та унікальних вражень для користувачів веб-сайту.

Односторінкова програма — це веб-сайт або веб-програма, яка динамічно переписує поточну веб-сторінку новими даними з веб-сервера замість методу за замовчуванням, коли веб-браузер завантажує цілі нові сторінки.

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

Однак у минулому SPA залишали маркетологів у невіданні, коли справа доходила до управління вмістом. Робота з односторінковою програмою історично була складною програмою, яка не задовольняла бажання та потреби маркетолога.

На щастя, тепер можна поєднати ваш SPA з правильною системою керування вмістом (CMS), щоб надати розробникам і маркетологам той рівень контролю, який вони шукають.

Готові дізнатися все про SPA та те, що вони можуть зробити для вашого бізнесу електронної комерції? Ось усе, що вам потрібно знати.

SPA — це веб-сайт або веб-додаток, який динамічно переписує поточну веб-сторінку новими даними з веб-сервера замість методу за замовчуванням, коли веб-браузер завантажує повністю нові сторінки.

SPA пропонують плавний і динамічний досвід користувача, завантажуючи всі необхідні ресурси під час початкового завантаження сторінки. Коли користувачі взаємодіють із програмою, вміст динамічно оновлюється, усуваючи необхідність перезавантажувати сторінки та зменшуючи запити до сервера.

Покращена продуктивність додатків, узгодженість, скорочений час розробки та нижчі витрати на інфраструктуру є ключовими перевагами SPA.

Односторінкова програма — це одна сторінка (звідси й назва), на якій багато інформації залишається незмінним і лише кілька фрагментів потрібно оновлювати одночасно.

Наприклад, коли ви переглядаєте свою електронну пошту, ви помітите, що під час навігації мало що змінюється — бічна панель і заголовок залишаються недоторканими, коли ви переглядаєте папку «Вхідні».

SPA надсилає лише те, що вам потрібно з кожним клацанням, і ваш браузер відображає цю інформацію. Це відрізняється від традиційного завантаження сторінки, коли сервер повторно відображає повну сторінку з кожним клацанням миші та надсилає її у ваш браузер.

Цей поетапний метод на стороні клієнта значно прискорює час завантаження для користувачів. Це також зменшує обсяг інформації, яку сервер має надсилати, і робить увесь процес набагато ефективнішим за витратами — безпрограшний сценарій для користувачів і компаній.

Переваги односторінкового додатка очевидно. Використання веб-програми або веб-сайту, який взаємодіє з користувачем шляхом динамічного переписування поточної сторінки, а не завантаження цілком нових сторінок із сервера, забезпечує набагато кращий досвід роботи з користувачем.

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

І це може мати величезний вплив. Оскільки на більшості веб-сайтів є багато повторюваного вмісту.

Деякі з них залишаються незмінними незалежно від того, куди йде користувач (верхні та нижні колонтитули, логотипи та панель навігації), а деякі з них незмінні лише в певному розділі (панелі фільтрів і банери). І існує багато повторюваних макетів і шаблонів (блоги, сторінки самообслуговування або вищезгадане налаштування пошти Google).

Односторінкові програми використовують переваги цього повторення.

Припустімо, ви відвідуєте веб-сайт і переглядаєте картину з будинком і деревом. Традиційні багатосторінкові веб-сайти малюють для вас повну картину на сервері та надсилають її у ваш браузер.

# 2 ПРОЄКТУВАННЯ СИСТЕМИ

При проєктуванні системи управління бізнес-процесами (Business Process Management, BPM) на основі інструментарію Spring можна використовувати різні архітектурні підходи та патерни. Ось декілька типів проєктування та їх характеристики.

## 2.1 Монолітна архітектура

Монолітна архітектура - це тип архітектурного підходу до розробки програмного забезпечення, в якому весь функціонал додатка об'єднаний в одному цілому, нероз'ємному компоненті, який зазвичай розглядається як "моноліт". У монолітній архітектурі усі компоненти додатка, такі як інтерфейс користувача, бізнес-логіка та доступ до даних, розглядаються як частини одного цілого, і вони розгортаються разом на одному сервері або наборі серверів.


Рисунок. 2.1 – Монолітна архітектура

Монолітна архітектура є традиційним підходом до розробки програмного забезпечення, при якому всі компоненти системи інтегровані в єдину, нероздільну кодову базу. У такій структурі функціональні модулі, як-от інтерфейс користувача, бізнес-логіка та доступ до даних, тісно пов'язані та працюють як єдине ціле. Це означає, що всі частини програми розгортаються та виконуються разом, що спрощує процеси розробки та тестування на початкових етапах.

Переваги монолітної архітектури включають простоту розробки, оскільки всі компоненти знаходяться в одному місці, що полегшує управління кодом. Крім того, розгортання такого застосунку є відносно простим, оскільки не потрібно координувати роботу окремих сервісів. Однак зі зростанням розміру та складності програми можуть виникати певні недоліки. Зокрема, внесення змін до одного модуля може вплинути на інші частини системи, що ускладнює підтримку та розвиток продукту. Також масштабування монолітного застосунку може бути проблематичним, оскільки неможливо масштабувати окремі компоненти незалежно один від одного.

З розвитком технологій та зростанням вимог до гнучкості та масштабованості програмного забезпечення, багато організацій переходять від монолітної архітектури до мікросервісної. Мікросервісна архітектура передбачає розбиття

програми на невеликі, незалежні сервіси, кожен з яких відповідає за конкретну функціональність і може розгортатися та масштабуватися окремо. Це дозволяє підвищити гнучкість розробки та спрощує підтримку системи. Однак перехід від моноліту до мікросервісів вимагає ретельного планування та врахування багатьох факторів, таких як управління даними, комунікація між сервісами та забезпечення безпеки.

У підсумку, вибір між монолітною та мікросервісною архітектурою залежить від специфіки проекту, його розміру, складності та вимог до масштабованості. Для невеликих та середніх проектів монолітна архітектура може бути оптимальним вибором завдяки своїй простоті та швидкості розробки. Для великих та складних систем, які потребують високої гнучкості та масштабованості, мікросервісна архітектура може бути більш підходящою.

## 2.2 Мікросервісна архітектура

Мікросервісна архітектура - це підхід до розробки програмного забезпечення, в якому додаток складається з невеликих, незалежних сервісів, кожен з яких виконує певну функцію або набір функцій. Кожен сервіс може бути розгорнутий, масштабований та керований незалежно, що робить мікросервісну архітектуру гнучкою та масштабованою.

— Розділення функціональності: У мікросервісній архітектурі функціонал поділяється на невеликі, незалежні одиниці, кожна з яких виконує чітко визначену задачу.

— Незалежність та автономія: Кожен сервіс може бути розгорнутий та масштабований незалежно від інших сервісів. Це дозволяє командам розробників працювати над різними частинами системи незалежно одна від одної.
— Легка заміна: при необхідності мікросервіс можна без особливих зусиль замінити.

Гнучкість та швидкість розгортання: Мікросервіси можуть бути розгорнуті та оновлені швидко та гнучко, оскільки кожен сервіс є окремим компонентом.

— Масштабованість: Мікросервіси дозволяють масштабувати та обслуговувати лише ті частини системи, які потребують додаткових ресурсів, що полегшує масштабування системи в цілому.
— Полегшення розвитку та тестування: Відокремленість сервісів у мікросервісній архітектурі спрощує розробку та тестування системи, оскільки кожен сервіс може бути розроблений та протестований окремо.
— Зменшення зв'язаності: Мікросервіси допомагають зменшити зв'язаність між різними компонентами системи, що полегшує розвиток, тестування та підтримку системи в цілому.

Хоча мікросервісна архітектура надає багато переваг, вона також може мати свої виклики, такі як складність управління великою кількістю сервісів, складність у взаємодії між сервісами та необхідність ефективного моніторингу та управління. Однак для багатьох організацій мікросервісна архітектура є ефективним підходом до розробки та підтримки великих та складних систем.

Мікросервісна архітектура є підходом до розробки програмного забезпечення, при якому додаток розбивається на невеликі, незалежні сервіси, кожен з яких відповідає за конкретну бізнес-функцію. Кожен мікросервіс працює у власному процесі та взаємодіє з іншими сервісами через легкі протоколи, зазвичай HTTP або повідомлення. Цей підхід дозволяє розробляти, розгортати та

масштабувати сервіси незалежно один від одного, що підвищує гнучкість та швидкість розробки [1].

Основними перевагами мікросервісної архітектури є підвищена масштабованість, оскільки кожен сервіс може бути масштабований окремо відповідно до навантаження. Крім того, це сприяє підвищенню стійкості системи, оскільки відмова одного сервісу не призводить до зупинки всього додатка. Мікросервіси також полегшують впровадження нових технологій, оскільки різні сервіси можуть бути написані на різних мовах програмування та використовувати різні бази даних [2].

Однак мікросервісна архітектура має і свої виклики. Зокрема, управління розподіленою системою може бути складним, оскільки необхідно забезпечити надійну комунікацію між сервісами, управління транзакціями та узгодженість даних. Також зростає складність розгортання та моніторингу системи, оскільки кількість сервісів може бути значною [3].

Для успішного впровадження мікросервісної архітектури важливо враховувати принципи доменно-орієнтованого проєктування (DDD), що допомагає визначити межі сервісів відповідно до бізнес-доменів. Також рекомендується використовувати автоматизовані процеси розгортання (CI/CD) та інструменти для моніторингу та логування, щоб забезпечити надійну роботу системи [4].

Оскільки мікросервіси бажано розгортати окремо, важливо забезпечити зворотну сумісність інтерфейсів, щоб інші сервіси могли працювати з новою версією без проблем. Також, взаємодія мікросервісів повинна бути стійкою до можливих збоїв окремих компонентів: припинення роботи одного мікросервісу, наприклад, під час оновлення, не повинно спричиняти збою в роботі інших частин системи, а лише тимчасово обмежувати функціональність або подовжити час обробки.

Кожен мікросервіс має власне сховище даних, що робить його самостійним та дозволяє використовувати різні бази даних залежно від потреб. Це підвищує

продуктивність, масштабованість та гнучкість системи. Основним недоліком є необхідність адміністрування та підтримки великої кількості баз даних.

Добре спроєктовані мікросервіси легко супроводжуються як з точки зору розгортання, так і експлуатації.

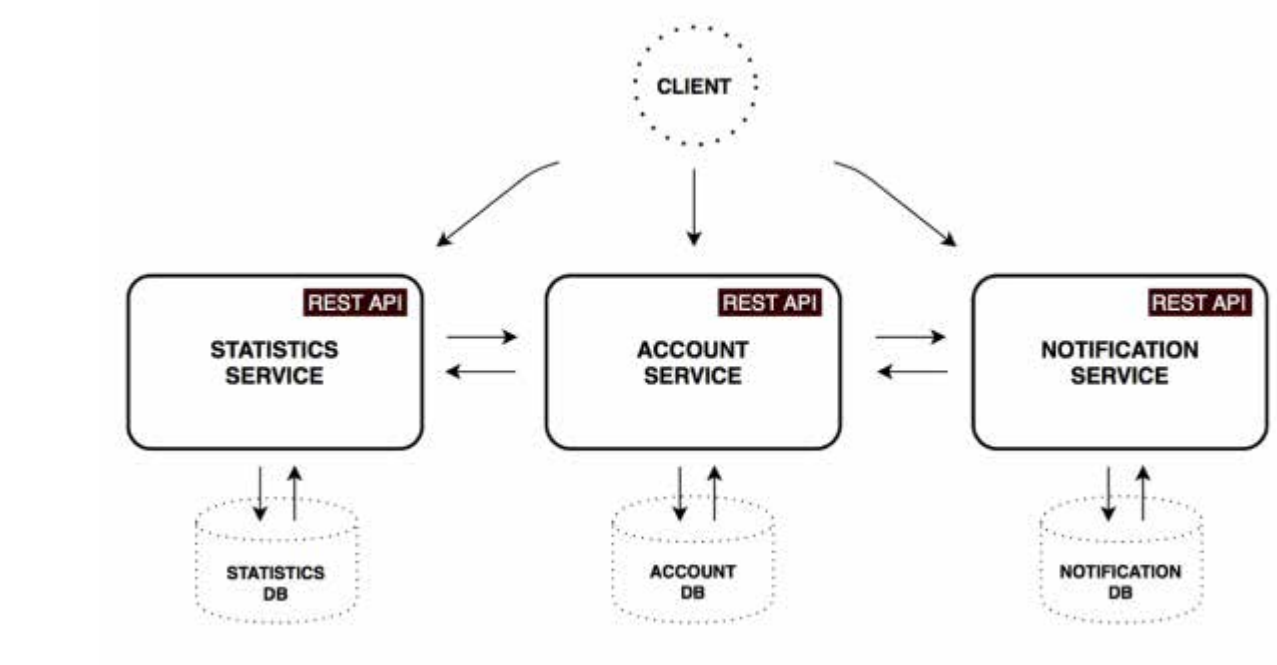


Рисунок 2.2 Мікросервісна архітектура

У підсумку, мікросервісна архітектура є потужним підходом для розробки складних та масштабованих систем, але її впровадження вимагає ретельного планування та врахування специфіки проєкту.

Характерні ознаки та переваги мікросервісної архітектури

Мікросервісна архітектура характеризується кількома ключовими особливостями:

— Мікросервіс відповідає за одну функцію: кожен мікросервіс виконує окреме завдання, що полегшує управління та розробку.

— Окреме розгортання: мікросервіси можна розгортати незалежно один від одного, що забезпечує гнучкість системи.
— Автономність процесів: один мікросервіс може складатися з одного або кількох взаємопов’язаних процесів, які виконують специфічні завдання.
— Власне сховище даних: кожен мікросервіс зберігає дані в окремій базі, що дозволяє вибирати найбільш оптимальні рішення для зберігання.
— Легка заміна: при необхідності мікросервіс можна без особливих зусиль замінити.
— Невелика команда підтримки: невелика команда розробників може супроводжувати декілька мікросервісів.

Мікросервісний підхід швидко завойовує популярність у розробці масштабного програмного забезпечення завдяки ряду переваг, які він надає в порівнянні з традиційними сервісорієнтованими і монолітними архітектурами.

Таблиця 2.1 – переваги та недоліки мікросервісної архітектури

| Характеристика | Переваги | Недоліки |
|---|---|---|
| Модульність | Простота внесення змін | Складність тестування |
| Ізольованість | Висока відмовостійкість | Вимагає ефективної координації між сервісами |
| Гнучкість технологій | Відсутність прив’язки до конкретних технологій | Значні витрати на інфраструктуру |
| Автономність | Незалежне розгортання сервісів | Дорожче управління з більшою кількістю сервісів |
| Стабільність | Вилучення мікросервісу не порушує роботу системи | - |

Мікросервісна архітектура є ефективним рішенням для розробки масштабованого програмного забезпечення завдяки можливості розгортання

окремих сервісів, їх автономності та високій відмовостійкості. Переваги цієї архітектури, зокрема гнучкість у виборі технологій і спрощення процесу підтримки, роблять її привабливою для великих проєктів. Однак для повноцінного впровадження мікросервісів необхідно враховувати деякі складнощі, такі як тестування та координування сервісів, що вимагають значних ресурсів.

## 2.3 Архітектура, керована подіями

Event-Driven Architecture (EDA) - це підхід до розробки програмного забезпечення, де компоненти системи взаємодіють між собою через взаємодію подіями (events). У цій архітектурі компоненти системи не спілкуються напряму, а взаємодіють через відправку та отримання подій, які вони спостерігають або генерують

- Асинхронність: Всі взаємодії між компонентами відбуваються асинхронно, що дозволяє розділити компоненти системи та зменшити зв'язність між ними.
- Події (events): Центральним елементом EDA є події, які можуть бути будь-якими змінами стану системи або зовнішніми подіями, такими як запити користувачів або сповіщення від інших систем.
- Виробники та споживачі: Компоненти системи можуть бути як виробниками, що генерують події, так і споживачами, що реагують на події.
- Розсилання подій: Події можуть бути розсилані або розповсюджувані через систему до всіх зацікавлених споживачів, або до конкретних споживачів, які підписалися на певні типи подій.

– Масштабованість та гнучкість: EDA дозволяє побудувати гнучкі та масштабовані системи, оскільки компоненти можуть бути додані або змінені без впливу на інші частини системи.
– Управління подіями: Управління подіями включає в себе маршрутизацію подій, фільтрацію, перетворення та обробку подій відповідно до бізнес-правил.

EDA використовується в багатьох сферах розробки програмного забезпечення, таких як системи реального часу, мікросервісна архітектура, системи інтеграції та багато інших. Вона дозволяє побудувати реактивні, гнучкі та відмовостійкі системи, які можуть ефективно реагувати на зміни у середовищі та потреби бізнесу.

Подія може бути визначена як "значна зміна стану".[2] Наприклад, коли споживач купує автомобіль, стан автомобіля змінюється з «для продажу» на «продано». Архітектура системи автодилера може розглядати цю зміну стану як подію, про виникнення якої можна повідомити іншим програмам у межах архітектури. З формальної точки зору те, що створюється, публікується, розповсюджується, виявляється або споживається, є (зазвичай асинхронним) повідомленням, яке називається сповіщенням про подію, а не самою подією, яка є зміною стану, яка викликала надсилання повідомлення. Події не подорожують, вони просто відбуваються. Однак термін подія часто використовується метонімічно для позначення самого сповіщення, що може призвести до певної плутанини. Це пов’язано з тим, що архітектури, керовані подіями, часто розробляються поверх архітектур, керованих повідомленнями, де такий шаблон зв’язку вимагає, щоб один із вхідних даних був лише текстовим, повідомленням, щоб диференціювати спосіб обробки кожного повідомлення.

Цей архітектурний шаблон може бути застосований при розробці та реалізації програм і систем, які передають події між слабко пов’язаними програмними

компонентами та службами. Система, керована подіями, зазвичай складається з джерел подій (або агентів), споживачів подій (або приймачів) і каналів подій. Випромінювачі відповідають за виявлення, збір і передачу подій. Випромінювач подій не знає споживачів події, він навіть не знає, чи існує споживач, і якщо він існує, він не знає, як подія використовується чи далі обробляється. Мийки несуть відповідальність за застосування реакції, щойно представлена подія. Реакція може бути або не повністю забезпечена самою раковиною. Наприклад, приймач може просто нести відповідальність за фільтрацію, перетворення та пересилання події до іншого компонента або він може забезпечити самодостатню реакцію на таку подію. Канали подій – це канали, в яких події передаються від джерел подій до споживачів подій. Знання правильного розподілу подій присутні виключно в каналі подій. Фізична реалізація каналів подій може базуватися на традиційних компонентах, таких як проміжне програмне забезпечення, орієнтоване на повідомлення, або зв'язок «точка-точка», що може вимагати більше відповідну виконавчу структуру транзакцій

Побудова систем навколо архітектури, керованої подіями, спрощує горизонтальну масштабованість у розподілених обчислювальних моделях і робить їх більш стійкими до збоїв. Це пояснюється тим, що стан додатка можна скопіювати на кілька паралельних знімків для високої доступності.[3] Нові події можна ініціювати будь-де, але, що важливіше, поширюватися мережею сховищ даних, оновлюючи кожне в міру надходження. Додавання додаткових вузлів також стає тривіальним: ви можете просто взяти копію стану програми, передати їй потік подій і запустити з нею.[4]

Архітектура, керована подіями, може доповнювати сервіс-орієнтовану архітектуру (SOA), оскільки служби можуть бути активовані тригерами, що запускаються при вхідних подіях.[5][6] Ця парадигма особливо корисна, коли раковина не забезпечує жодного автономного виконавця.

SOA 2.0 розвиває наслідки, які надають архітектури SOA та EDA, на більш багатий і надійний рівень, використовуючи раніше невідомі причинно-наслідкові зв’язки для формування нового шаблону подій.[розпливчасто]. Цей новий шаблон бізнес-аналітики запускає подальшу автономну людську або автоматизовану обробку, що додає експоненціальну цінність підприємство шляхом введення інформації з доданою вартістю у визнаний шаблон, чого не можна було досягти раніше.

## 2.4 Шестикутна архітектура

Концепція гексагональної архітектури була запропонована Алістером Кокберном у середині 90-х у статті, опублікованій на першій створеній вікіпедії під назвою WikiWikiWeb (яка в основному охоплювала теми програмної інженерії).

Цілі Hexagonal Architecture подібні до цілей Clean Architecture, як ми описали в іншій статті. Зокрема, ідея полягає в створенні систем, які мають високу когезію, низький зв’язок і які легше тестувати.

Гексагональна архітектура поділяє класи системи на дві основні групи:

Класи домену, які безпосередньо пов’язані з бізнес-логікою системи.

Зовнішні класи, наприклад ті, що стосуються інтерфейсу користувача та інтеграції із зовнішніми системами (такими як бази даних).

Крім того, класи домену не повинні залежати від класів, пов’язаних з інфраструктурою, інтерфейсом користувача або зовнішніми системами. Основна перевага цього поділу полягає в тому, що ці два типи класів роз’єднані.

У результаті класи домену залишаються незалежними від технологій, включаючи бази даних, інтерфейси користувача та будь-які інші бібліотеки, які використовує система. Отже, технологічні зміни можна вносити без впливу на класи домену. Можливо, навіть важливіше те, що класи домену можна спільно використовувати на кількох технологічних платформах. Наприклад, система може мати різні інтерфейси (веб, мобільний тощо). Крім того, що не менш важливо, легше тестувати класи домену, якщо вони відокремлені від класів, не пов’язаних з бізнесом.

У гексагональній архітектурі зв’язок між класами цих груп здійснюється за допомогою адаптерів6. Незабаром ми пояснимо роль адаптерів.

Архітектура зазвичай представлена двома концентричними шестикутниками (див. наступний малюнок). Класи домену (або бізнес-класи, якщо хочете) знаходяться у внутрішньому шестикутнику. У зовнішньому шестикутнику ми маємо перехідники. Нарешті, бази даних та інші зовнішні системи розташовані за межами цих двох шестикутників (і, таким чином, не представлені на малюнку).

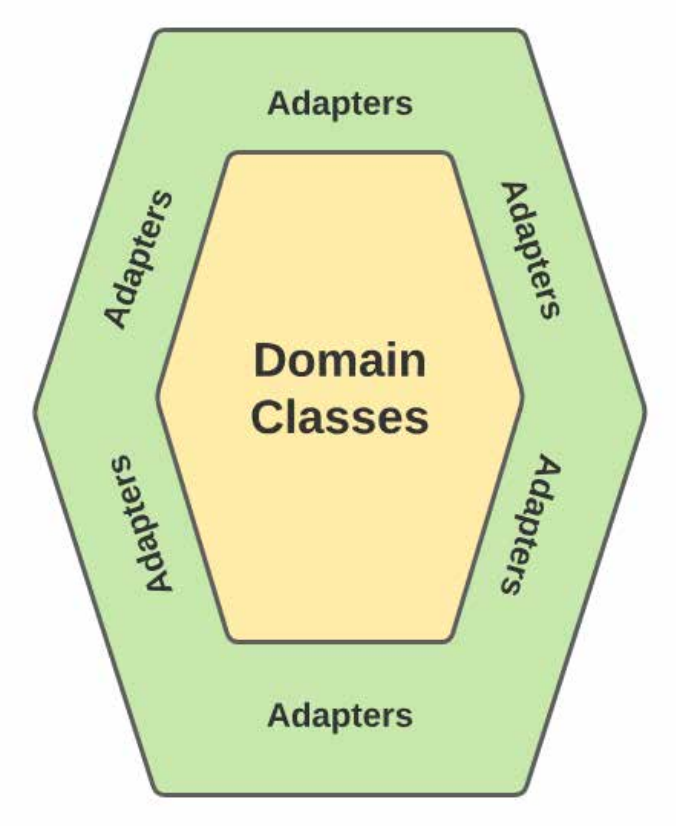


Рисунок.2.3 - Hexagonal Architecture

Серед причин, які вимагають зв’язку із зовнішнім світом, можна назвати наступне: взаємодія з користувачами (через інтерфейс користувача, такий як Інтернет, мобільний пристрій, консоль тощо), збереження даних і надсилання даних до інших систем.

У Hexagonal Architectures термін порт позначає інтерфейси, які надаються або вимагаються класами домену (зверніть увагу, що тут інтерфейс означає конструкцію мови програмування; наприклад, інтерфейс Java).

Існує два типи портів:

Надані порти (або надані інтерфейси) — це інтерфейси, які використовуються для зв'язку ззовні всередину, тобто коли зовнішній системі потрібно викликати метод у домені. Ці порти визначають служби (або API), які надає домен, представляючи служби, які домен реалізує для зовнішнього світу.

Необхідні порти (або необхідні інтерфейси) — це інтерфейси, які використовуються, коли клас домену потребує виклику зовнішнього методу. Ці порти декларують служби зовнішнього світу, необхідні для роботи домену.

Нарешті, у нас є компоненти, розташовані в самому зовнішньому шестикутнику архітектури — адаптери, — які працюють одним із двох способів:

Адаптери можуть отримувати виклики методів, що надходять ззовні системи, і направляти ці виклики до відповідних методів домену (як визначено в наданих інтерфейсах). Наприклад, ці адаптери можуть використовувати REST, GraphQL або gRPC для зв'язку із зовнішніми системами. Отже, домен залишається незалежним і відокремленим від таких технологій.

Адаптери можуть отримувати виклики, що надходять із домену, і направляти їх до зовнішньої системи, такої як база даних або система обробки кредитних карток. Зокрема, адаптери, які взаємодіють з базами даних, зазвичай називають репозиторіями.

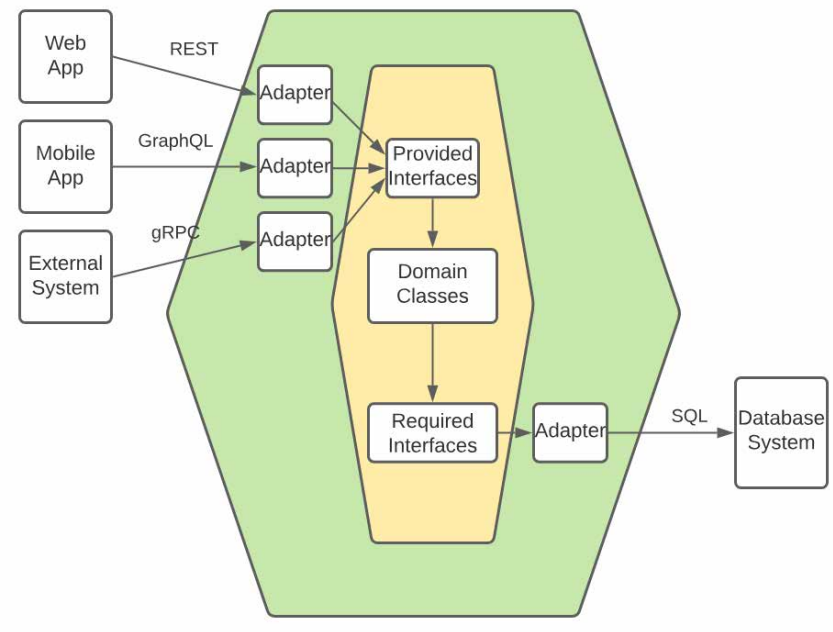


Рисунок.2.4 Hexagonal Architecture of a Library Management System

На малюнку ми бачимо, що користувачі отримують доступ до системи за допомогою трьох інтерфейсів: веб-, мобільного та зовнішньої системи. У всіх випадках цей доступ здійснюється за допомогою адаптерів у верхній частині малюнка. Ці адаптери викликають методи, визначені в порту (або інтерфейсі), наданому доменом. Цей інтерфейс може визначати методи для пошуку книг, позичання та реєстрації користувачів у бібліотеці, серед іншого. Ці методи реалізуються класами домену.

Щоб виконати свою місію, домен також має зберігати дані. З цією метою він може використовувати служби, визначені в необхідному порту (або інтерфейсі), який визначає методи для збереження та отримання даних книги, збереження та отримання даних кредиту, серед інших операцій. Адаптер у нижній частині малюнка реалізує ці методи шляхом доступу до реляційної бази даних. Отже, ця база даних є зовнішньою системою, розташованою за межами двох шестикутників.

У Hexagonal Architectures ми можемо мати кілька необхідних і наданих портів, усі розташовані у внутрішньому шестикутнику разом із класами домену.

Крім того, ми маємо два типи адаптерів: (1) адаптери, які викликають методи, визначені в портах (або інтерфейсах), реалізованих доменом; і (2) адаптер.

# 3 РОЗРОБКА КОМП'ЮТЕРНОЇ СИСТЕМИ УПРАВЛІННЯ БІЗНЕС ПРОЦЕСАМИ З ВИКОРИСТАННЯМ SPRING

## 3.1 Інтеграція Spring з MySQL

Інтеграція Spring з MySQL — це поширений і ефективний підхід для створення веб-додатків, де необхідно взаємодіяти з реляційною базою даних. Spring Framework, разом із MySQL, забезпечує потужний інструментарій для створення надійних додатків з високою продуктивністю. Реалізація інтеграції вимагає виконання декількох основних етапів, які включають налаштування проєкту, підключення залежностей, конфігурацію з'єднання з базою даних, створення моделей та репозиторіїв для обробки даних.

На початковому етапі необхідно створити новий Spring Boot проєкт, використовуючи Spring Initializr. При створенні проєкту важливо вказати залежності, необхідні для інтеграції з MySQL. Основними залежностями є Spring Data JPA для роботи з базою даних через ORM (Object Relational Mapping) та MySQL Driver для підключення до MySQL. Після генерації проєкту Spring Boot створює базову структуру проєкту разом із необхідними файлами конфігурації, такими як application.properties або application.yml.

Конфігурація з'єднання з MySQL виконується у файлі application.properties, де вказуються параметри з'єднання з базою даних, такі як URL, ім'я користувача та пароль. Для цього вносяться наступні налаштування:

spring.datasource.url=jdbc:mysql://localhost:3306/your_database
spring.datasource.username=your_username
spring.datasource.password=your_password spring.datasource.driver-class-

name=com.mysql.cj.jdbc.Driver spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

Значення spring.datasource.url вказує на URL бази даних, включаючи порт (у цьому випадку 3306) та ім'я бази даних. Поле spring.jpa.hibernate.ddl-auto можна налаштувати для автоматичного створення або оновлення таблиць при запуску програми. Параметр update оновлює структуру бази даних відповідно до моделей, що зручно на етапі розробки.

Після налаштування конфігурації з'єднання наступним кроком є створення моделей, які відображатимуть таблиці бази даних. У Spring Data JPA моделі реалізуються як класи, а їхні властивості — як поля класів, які відповідають колонкам бази даних. Аннотації, такі як @Entity, @Table, @Id, @GeneratedValue, використовуються для позначення класу як сутності та для визначення первинного ключа. Наприклад, модель "Користувач" буде мати вигляд:

Лістинг 3.1.1 Код представлення користувача мовою Java

```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;
```

```
    // Getters and setters
}
```

Клас позначається як сутність з ім'ям таблиці users. Поле id оголошується як первинний ключ і автоматично генерується MySQL при створенні нового запису. Інші поля класу, наприклад name та email, відповідають відповідним колонкам таблиці.

Для доступу до даних із бази даних потрібно створити репозиторій для сутності. Spring Data JPA спрощує роботу з репозиторіями, дозволяючи створювати інтерфейси, що розширюють JpaRepository. Це надає доступ до CRUD (створення, читання, оновлення та видалення) операцій без необхідності писати SQL-запити вручну. Інтерфейс UserRepository для сутності "Користувач" виглядатиме так:

import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {

На цьому етапі можна починати працювати з даними, використовуючи стандартні методи репозиторію. Наприклад, у класі-сервісі, який містить бізнес-логіку, можна використовувати UserRepository для зберігання нових користувачів, оновлення існуючих або видалення за допомогою методів save, findById, deleteById тощо. Сервісний клас UserService, що використовує репозиторій UserRepository, буде виглядати так:

Лістинг 3.1.2 Код сервісу роботи з користувачами мовою Java

```java
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.Optional;
@Service
public class UserService {
    private final UserRepository userRepository;
    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }
    public Optional<User> getUserById(Long id) {
        return userRepository.findById(id);
    }
    public User saveUser(User user) {
        return userRepository.save(user);
    }
    public void deleteUserById(Long id) {
        userRepository.deleteById(id);
    }
}
```

```

Для надання доступу до методів сервісу через веб-інтерфейс створюється REST-контролер. Він міститиме методи, що відповідають на HTTP-запити для виконання CRUD-операцій. Контролер UserController може виглядати так:

Лістинг 3.1.3 Код контролеру Http запитів мовою Java

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;
@RestController
@RequestMapping("/api/users")
public class UserController {
    private final UserService userService;
    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }
    @GetMapping
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }
    @GetMapping("/{id}")
```
```

```
    public User getUserById(@PathVariable Long id) {
        return userService.getUserById(id).orElse(null);
    }
    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.saveUser(user);
    }
    @PutMapping("/{id}")
    public User updateUser(@PathVariable Long id, @RequestBody User updatedUser) {
        User user = userService.getUserById(id).orElseThrow();
        user.setName(updatedUser.getName());
        user.setEmail(updatedUser.getEmail());
        return userService.saveUser(user);
    }
    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        userService.deleteUserById(id);
    }
}
```

Методи контролера відповідають на запити GET, POST, PUT і DELETE, що дозволяє виконувати базові операції з даними через HTTP-запити. Наприклад, щоб

отримати список користувачів, система надсилає запит GET на /api/users, а щоб створити нового користувача — POST на /api/users.

Таким чином, інтеграція Spring з MySQL охоплює налаштування конфігурації підключення, створення моделі даних, визначення репозиторіїв, написання сервісного класу та побудову контролера для доступу до даних. Такий підхід спрощує розробку та масштабування додатків, надаючи гнучкість та зручність у роботі з реляційною базою даних.

Інтеграція Python з MySQL також є популярним рішенням для розробки додатків, де потрібно працювати з реляційними базами даних. У Python для інтеграції з MySQL можна використовувати фреймворк Flask у поєднанні з ORM-бібліотекою SQLAlchemy, що дозволяє створювати додатки з аналогічною структурою, як у випадку з Spring для Java. SQLAlchemy забезпечує інтерфейс для роботи з базою даних, який дозволяє абстрагувати запити SQL, спрощуючи їх реалізацію.

Спочатку створюється новий проєкт Flask та встановлюються необхідні залежності, такі як Flask, Flask-SQLAlchemy та mysql-connector-python для роботи з MySQL.

Лістинг 3.1.4 Bash команда встановлення залежностей для проекту на python

```bash
pip install Flask Flask-SQLAlchemy mysql-connector-python
```

Далі налаштовується підключення до MySQL, яке в Flask виконується в конфігурації програми. У файлі `app.py` або в конфігураційному файлі `config.py` налаштовуються параметри з'єднання:

Лістинг 3.1.5 Код налаштувань з’єднання з базою даних

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql+mysqlconnector://your_username:your_password@localhost/your_database'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)
```

Параметр `SQLALCHEMY_DATABASE_URI` вказує на URI для підключення до бази даних, де `mysql+mysqlconnector` визначає драйвер, який буде використовуватися для з'єднання. `your_username`, `your_password` і `your_database` замінюються відповідними значеннями для доступу до MySQL.

Наступним кроком є створення моделей, які відповідають таблицям у базі даних. Для цього у Flask-SQLAlchemy використовуються класи Python, що визначаються як сутності з полями, які відповідають колонкам бази даних.

Лістинг 3.1.6 Код представлення користувача мовою python

```python
from app import db

class User(db.Model):

    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True)

    name = db.Column(db.String(50), nullable=False)

    email = db.Column(db.String(120), unique=True, nullable=False)

    def __init__(self, name, email):

        self.name = name

        self.email = email
```

Тут клас `User` визначає таблицю `users` з полями `id`, `name` та `email`. Поле `id` є первинним ключем, який генерується автоматично, тоді як інші поля відповідають відповідним типам даних SQL (строка та унікальна строка для `email`).

Лістинг 3.1.7 Код ініціалізації бази даних мовою python

```python
from app import db
db.create_all()
```

Далі створюється сервісний шар, що містить логіку роботи з моделлю. Наприклад, для доступу до даних у таблиці "Користувач" пишемо сервісні функції для збереження, оновлення, видалення та отримання даних. Всі ці функції реалізуються у вигляді окремих методів або в одному класі.

Лістинг 3.1.8 Код представлення користувача мовою python

```python
def add_user(name, email):
    user = User(name=name, email=email)
    db.session.add(user)
    db.session.commit()
def get_all_users():
    return User.query.all()
def get_user_by_id(user_id):
    return User.query.get(user_id)
def update_user(user_id, name=None, email=None):
    user = get_user_by_id(user_id)
    if user:
```

```
        if name:
            user.name = name
        if email:
            user.email = email
        db.session.commit()
    return user
def delete_user(user_id):
    user = get_user_by_id(user_id)
    if user:
        db.session.delete(user)
        db.session.commit()
```

Ці функції забезпечують основні операції CRUD (Create, Read, Update, Delete) і використовують сесію бази даних для виконання операцій.

На завершальному етапі створюється REST API-контролер для роботи з додатком. У Flask для цього використовуються декоратори `@app.route`, що дозволяють визначати кінцеві точки для обробки HTTP-запитів.

Лістинг 3.1.9 Код контролера Http запитів мовою python

```python
from flask import jsonify, request
from app import app, db
from models import User
```

```
@app.route('/users', methods=['GET'])
def get_users():
    users = User.query.all()
    return jsonify([{'id': user.id, 'name': user.name, 'email': user.email} for user in users])
@app.route('/users/<int:id>', methods=['GET'])
def get_user(id):
    user = User.query.get_or_404(id)
    return jsonify({'id': user.id, 'name': user.name, 'email': user.email})
@app.route('/users', methods=['POST'])
def create_user():
    data = request.get_json()
    new_user = User(name=data['name'], email=data['email'])
    db.session.add(new_user)
    db.session.commit()
    return jsonify({'id': new_user.id}), 201
@app.route('/users/<int:id>', methods=['PUT'])
def update_user(id):
    user = User.query.get_or_404(id)
    data = request.get_json()
    user.name = data['name']
    user.email = data['email']
    db.session.commit()
```

```
    return jsonify({'id': user.id})

@app.route('/users/<int:id>', methods=['DELETE'])

def delete_user(id):

    user = User.query.get_or_404(id)

    db.session.delete(user)

    db.session.commit()

    return '', 204
```

Кожен маршрут обробляє відповідний HTTP-запит (`GET`, `POST`, `PUT`, `DELETE`), що дозволяє виконувати CRUD-операції для користувачів.

Таким чином, інтеграція Python з MySQL за допомогою Flask і SQLAlchemy передбачає налаштування з'єднання з базою даних, створення моделей для таблиць, реалізацію CRUD-операцій через сервісні функції та налаштування REST API-контролера для доступу до даних. Така структура забезпечує зручну та гнучку платформу для роботи з реляційною базою даних у Python.

## 3.2 Використання Spring Boot для створення основи додатку

Використання Spring Boot для створення основи додатку (Java). Spring Boot забезпечує швидке налаштування і створення веб-додатків на основі Java, надаючи необхідні залежності, конфігурації та автоналаштування. Для початку створюється Spring Boot проєкт, де основні налаштування і залежності визначаються у файлі `pom.xml`, якщо використовується Maven, або у файлі `build.gradle`, якщо використовується Gradle. Spring Initializr є зручним інструментом для генерації базового шаблону проєкту з обраними залежностями, такими як Spring Web для створення REST API або Spring Data JPA для роботи з базами даних.

Після створення проєкту Spring Boot автоматично генерує структуру каталогу та основний клас програми, позначений аннотацією `@SpringBootApplication`. Це основний клас, з якого запускається додаток:

Лістинг 3.2.1 Код основного класу мовою java

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class MyAppApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyAppApplication.class, args);
    }
}
```

```

Аннотація `@SpringBootApplication` автоматично налаштовує додаток, включаючи компонент сканування, автоматичне налаштування і налаштування контексту додатка, завдяки чому немає потреби вручну конфігурувати ці аспекти. Для створення REST API в Spring Boot використовуються контролери, які відповідають за обробку HTTP-запитів. Наприклад, `MyController` може бути створений для обробки запитів `GET`, `POST`, `PUT` та `DELETE`:

Лістинг 3.2.2 Код контролеру Http запитів мовою java

```java
import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/api")
public class MyController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, World!";
    }
}
```

Тут `@RestController` вказує, що це контролер, який обробляє REST-запити. Використання Spring Boot дозволяє швидко додавати інші компоненти, як-от сервіси, репозиторії для роботи з даними, компоненти безпеки та логування, без додаткових конфігурацій завдяки принципу автоналаштування.
```

У Python для створення додатка з подібною структурою можна використовувати Flask. Flask — це мінімалістичний веб-фреймворк, що дозволяє швидко налаштувати базову структуру додатка.

Лістинг 3.2.3 Код методу контроллеру Http запитів мовою java

```python
from flask import Flask

app = Flask(__name__)

@app.route('/hello', methods=['GET'])

def hello():

    return "Hello, World!"

if __name__ == '__main__':

    app.run(debug=True)
```

Тут метод `@app.route` визначає маршрут для запиту `GET` на `/hello`. `app.run(debug=True)` запускає сервер у режимі налагодження.

Для організації більших додатків Flask дозволяє створювати модулі для окремих компонентів і підключати додаткові розширення. Наприклад, для роботи з базою даних можна використовувати `Flask-SQLAlchemy`, для автентифікації — `Flask-Login`, а для обробки запитів — `Flask-RESTful`.

Порівняння обох підходів

Spring Boot забезпечує повну автоматизацію налаштувань та містить велику кількість вбудованих функцій для побудови корпоративних додатків, що

включають роботу з базами даних, безпекою, транзакціями тощо. Він також підтримує гнучкі конфігурації завдяки YAML або `.properties` файлам, що робить його потужним інструментом для великих проєктів.

Flask, з іншого боку, є легшим фреймворком з меншими вимогами до конфігурації та залежностей, що робить його привабливим для невеликих і середніх проєктів. Flask також надає широку гнучкість для вибору додаткових бібліотек та розширень залежно від потреб проєкту, надаючи розробникам більшу свободу в дизайні додатка.

## 3.3 Налаштування безпеки додатку з використанням Spring Security

Spring Security — це потужний фреймворк, що забезпечує високий рівень захисту веб-додатків, включаючи автентифікацію, авторизацію та контроль доступу. Він легко інтегрується зі Spring Boot і надає готові механізми для реалізації захищених маршрутів, персоналізованих сторінок входу та доступу до ресурсів на основі ролей.

Спочатку у файлі `pom.xml` необхідно додати залежність Spring Security. Якщо проєкт створений за допомогою Spring Initializr, залежність можна додати на етапі створення проєкту, обравши Spring Security.

Лістинг 3.3.1 Налаштування залежностей проєкту мовою java

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Після додавання залежності Spring Boot автоматично активує базові налаштування безпеки, що захищають всі URL-адреси у додатку і вимагають автентифікації для доступу.

У тестових або простих додатках можна налаштувати користувачів і ролі безпосередньо в пам'яті. Для цього створюється конфігураційний клас, який розширює `WebSecurityConfigurerAdapter`, і налаштовуються користувачі з паролями:

Лістинг 3.3.2 Налаштування доступу до додатку через код мовою java

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authentication.builder
s.AuthenticationManagerBuilder;
```

```
   import
org.springframework.security.config.annotation.web.builders.HttpSecur
ity;

   import
org.springframework.security.config.annotation.web.configuration.Enab
leWebSecurity;

   import
org.springframework.security.config.annotation.web.configuration.WebS
ecurityConfigurerAdapter;

   import
org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

   import
org.springframework.security.crypto.password.PasswordEncoder;

   @Configuration

   @EnableWebSecurity

   public class SecurityConfig extends WebSecurityConfigurerAdapter {

       @Override

       protected  void  configure(AuthenticationManagerBuilder  auth)
throws Exception {

           auth.inMemoryAuthentication()


.withUser("user").password(passwordEncoder().encode("password")).role
s("USER")

               .and()


.withUser("admin").password(passwordEncoder().encode("admin")).roles(
"ADMIN");

       }
```

```
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/admin/").hasRole("ADMIN")
                .antMatchers("/user/").hasAnyRole("USER", "ADMIN")
                .antMatchers("/public/").permitAll()
                .anyRequest().authenticated()
                .and()
            .formLogin()
                .loginPage("/login")
                .permitAll()
                .and()
            .logout()
                .permitAll();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

У цьому прикладі ми створюємо двох користувачів: "user" з роллю "USER" та "admin" з роллю "ADMIN". Конфігурація `HttpSecurity` дозволяє встановлювати права доступу до різних маршрутів: всі користувачі можуть отримати доступ до `/public`, тоді як `/admin` доступний лише для користувачів з роллю "ADMIN".

Налаштуванням `http.formLogin().loginPage("/login")` вказується кастомна сторінка входу. Для створення такої сторінки потрібно додати контролер для маршруту `/login` та HTML-файл з формою для входу:

Лістинг 3.3.3 Код контролеру Http запитів на авторизацію мовою java

```java
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;


@Controller
public class LoginController {
    @GetMapping("/login")
    public String login() {
        return "login";
    }
}
```

Лістинг 3.3.4 HTML-шаблон сторінки `login.html`

```html
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Login</title>
</head>
<body>
    <h2>Login</h2>
    <form action="/login" method="post">
        <div>
            <label>Username:</label>
            <input type="text" name="username"/>
        </div>
        <div>
            <label>Password:</label>
            <input type="password" name="password"/>
        </div>
        <div>
            <button type="submit">Login</button>
        </div>
    </form>
</body>
</html>
```

Spring Security автоматично обробляє відправку форми `/login` і здійснює автентифікацію користувача.

Для реальних додатків краще зберігати дані про користувачів у базі даних. Налаштування Spring Security для використання бази даних потребує створення таблиць користувачів та ролей і конфігурування відповідного `UserDetailsService` для завантаження користувачів із бази даних.

Створіть таблиці `users` та `authorities` для зберігання даних про користувачів та їхні ролі. Приклад SQL-запиту для створення таблиць:

Лістинг 3.3.5 SQL скрипт для створення таблиць в базі даних

```sql
CREATE TABLE users (
    username VARCHAR(50) NOT NULL PRIMARY KEY,
    password VARCHAR(100) NOT NULL,
    enabled BOOLEAN NOT NULL
);
CREATE TABLE authorities (
    username VARCHAR(50) NOT NULL,
    authority VARCHAR(50) NOT NULL,
    FOREIGN KEY (username) REFERENCES users(username)
);
```

Лістинг 3.3.6 Код методу автентифікації та авторизації користувача мовою java

```java
@Autowired
DataSource dataSource;
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
     auth.jdbcAuthentication().dataSource(dataSource)
          .passwordEncoder(passwordEncoder())
          .usersByUsernameQuery("select username, password, enabled from users where username=?")
          .authoritiesByUsernameQuery("select username, authority from authorities where username=?");
}
```

Запити `usersByUsernameQuery` і `authoritiesByUsernameQuery` дозволяють Spring Security знаходити користувача та його ролі за допомогою SQL-запитів, використовуючи ім'я користувача.

Для налаштування виходу з системи додається конфігурацію `logout().permitAll()`, що дозволяє будь-якому користувачу виконати вихід із системи. Spring Security автоматично додає маршрут `/logout`, який можна налаштувати за потребою.

Лістинг 3.3.7 Код налаштування виходу з системи.

```java
http
      .logout()
      .logoutSuccessUrl("/")
      .permitAll();
```

Таким чином, Spring Security надає гнучкі інструменти для створення системи безпеки додатка, яка включає автентифікацію, авторизацію, захист маршрутів і налаштування сторінок входу/виходу.

Для налаштування безпеки в додатку на Python з використанням Flask, можна скористатися пакетом `Flask-Security`, який забезпечує такі функції, як автентифікація користувачів, авторизація та контроль доступу на основі ролей. Ось покрокова інструкція для налаштування безпеки з Flask:

Треба створити основний файл додатку (`app.py`) і налаштуйте його для роботи з SQLAlchemy як інструментом для взаємодії з базою даних:

Лістинг 3.3.8 Код основного файлу додатку проєкта мовою python.

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_security import Security, SQLAlchemyUserDatastore, UserMixin, RoleMixin
```

```
app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///security.db'
# Використання SQLite для простоти

app.config['SECRET_KEY'] = 'supersecretkey'

app.config['SECURITY_PASSWORD_SALT'] = 'some_random_salt'    #
Необхідно для хешування паролів

db = SQLAlchemy(app)
```

Визначте моделі для користувачів та ролей, наслідуючи `UserMixin` та `RoleMixin`, які надаються Flask-Security-Too. Ці моделі визначають ролі користувачів та їхні асоціації для автентифікації та авторизації.

Лістинг 3.3.9 Код представлення моделей та користувачів мовою python.

```python
# Моделі для користувачів і ролей
class Role(db.Model, RoleMixin):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True)
    description = db.Column(db.String(255))
    class User(db.Model, UserMixin):
        __tablename__ = 'users'
        id = db.Column(db.Integer, primary_key=True)
```

```
        email = db.Column(db.String(255), unique=True)

        password = db.Column(db.String(255))

        active = db.Column(db.Boolean())

        roles = db.relationship('Role', secondary='user_roles',
backref=db.backref('users', lazy='dynamic'))

    class UserRoles(db.Model):

        __tablename__ = 'user_roles'

        id = db.Column(db.Integer, primary_key=True)

        user_id = db.Column(db.Integer, db.ForeignKey('users.id',
ondelete='CASCADE'))

        role_id = db.Column(db.Integer, db.ForeignKey('roles.id',
ondelete='CASCADE'))
```

Треба стровити екземпляр `SQLAlchemyUserDatastore`, передавши базу даних і моделі `User` та `Role`, а потім ініціалізувати Flask-Security з додатком.

Лістинг 3.3.10 Код налаштування екземляру об’єкту класу роботи з базою даних мовою python.

```python
# Налаштування Flask-Security

user_datastore = SQLAlchemyUserDatastore(db, User, Role)

security = Security(app, user_datastore)
```

Flask-Security використовує декоратори для захисту маршрутів. Наприклад, `@login_required` гарантує, що користувач автентифікований, тоді як `@roles_required('admin')` обмежує доступ до користувачів із роллю "admin".

Лістинг 3.3.11 Код налаштування автентифікації мовою python.

```python
from flask_security import login_required, roles_required
@app.route('/profile')
@login_required
def profile():
    return "Це профіль користувача, який увійшов у систему."
@app.route('/admin')
@roles_required('admin')
def admin():
    return "Це сторінка адміністратора, доступна лише для користувачів з правами адміністратора."
```

Flask-Security надає готові маршрути для входу, реєстрації, виходу та відновлення паролю:

- `/login` — для входу
- `/register` — для реєстрації
- `/logout` — для виходу

За потреби, можна замінити шаблони цих сторінок, розмістивши власні HTML-файли в папці `templates/security`. Наприклад, створіть файл `templates/security/login_user.html` для кастомної сторінки входу.

Лістинг 3.3.12 Код налаштування налаштування функцій безпеки мовою python.

```python
# Конфігураційні опції
app.config['SECURITY_REGISTERABLE'] = True    # Дозволяє реєстрацію нових користувачів
app.config['SECURITY_RECOVERABLE'] = True     # Дозволяє відновлення паролю
app.config['SECURITY_TRACKABLE'] = True     # Відстежує інформацію про вхід користувачів
app.config['SECURITY_CHANGEABLE'] = True     # Дозволяє зміну паролю
```

Коли все налаштовано, ви можете запустити додаток і отримати доступ до маршрутів для входу, реєстрації та інших захищених сторінок:

Лістинг 3.3.13 Скрипт запуску python проєкту.

```bash
flask run
```

Лістинг 3.3.14 Фінальний код проєкту мовою python.

```python
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy
from flask_security import Security, SQLAlchemyUserDatastore, UserMixin, RoleMixin, login_required, roles_required
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///security.db'
app.config['SECRET_KEY'] = 'supersecretkey'
app.config['SECURITY_PASSWORD_SALT'] = 'some_random_salt'
app.config['SECURITY_REGISTERABLE'] = True
app.config['SECURITY_RECOVERABLE'] = True
db = SQLAlchemy(app)
# Визначення моделей
class Role(db.Model, RoleMixin):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True)
    description = db.Column(db.String(255))

class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(255), unique=True)
    password = db.Column(db.String(255))
    active = db.Column(db.Boolean())
```

```
        roles = db.relationship('Role', secondary='user_roles', backref=db.backref('users', lazy='dynamic'))

    class UserRoles(db.Model):

        id = db.Column(db.Integer, primary_key=True)

        user_id = db.Column(db.Integer, db.ForeignKey('user.id', ondelete='CASCADE'))

        role_id = db.Column(db.Integer, db.ForeignKey('role.id', ondelete='CASCADE'))

    # Налаштування Flask-Security

    user_datastore = SQLAlchemyUserDatastore(db, User, Role)

    security = Security(app, user_datastore)

    # Створення захищених маршрутів

    @app.route('/profile')

    @login_required

    def profile():

        return "Це профіль користувача, який увійшов у систему."

    @app.route('/admin')

    @roles_required('admin')

    def admin():

        return "Це сторінка адміністратора, доступна лише для користувачів з правами адміністратора."

    # Запуск додатку

    if __name__ == '__main__':

        db.create_all()

        app.run(debug=True)
```

```

Ця конфігурація забезпечує базову автентифікацію користувачів, авторизацію на основі ролей, а також кастомізовані сторінки для входу та реєстрації, що забезпечує надійну основу для безпеки у Flask-додатку.
```

## 3.4 Автоматизація бізнес процесів за допомогою Spring Batch

Spring Batch — це потужний фреймворк для розробки пакетних процесів, які можуть автоматизувати та оптимізувати бізнес-процеси. Його основні функції включають обробку великих обсягів даних, підтримку транзакційності, можливість відновлення процесів після помилок та інші критично важливі можливості для пакетної обробки даних.

Spring Batch базується на кількох основних компонентах:

- Job — це базовий контейнер для завдань. `Job` складається з одного або декількох кроків (step), кожен з яких виконує певну частину завдання.
- Step — це окремий етап у виконанні `Job`, який включає читання, обробку та запис даних.
- ItemReader, ItemProcessor і ItemWriter — ці компоненти відповідають за завантаження, обробку та збереження даних.

Лістинг 3.4.1 Опис залежностей проєкту мовою Java для додатку Maven.

```xml
<dependency>
    <groupId>org.springframework.batch</groupId>
    <artifactId>spring-batch-core</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-batch</artifactId>
```

```
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Ці залежності включають Spring Batch і Spring Data JPA для роботи з базами даних, а також вбудовану базу H2 для тестування.

Spring Batch потребує базу даних для зберігання метаданих про виконання пакетних процесів (наприклад, про успішне завершення, помилки тощо).

Лістинг 3.4.2 Конфігурація проєкту мовою Java.

```properties
spring.datasource.url=jdbc:h2:mem:testdb
```

```
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.batch.initialize-schema=always
```

Тепер налаштуємо Job, що міститиме один або кілька Step. Кожен Step виконуватиме певну операцію, наприклад читання даних з джерела, обробку даних та запис результатів у вихідне місце.

Лістинг 3.4.3 Налаштування Job для роботи з базою даних мовою Java.

```java
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchPr
ocessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFac
tory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFa
ctory;
import
org.springframework.batch.core.launch.support.RunIdIncrementer;
import org.springframework.context.annotation.Bean;
```

```
import org.springframework.context.annotation.Configuration;
@Configuration
@EnableBatchProcessing
public class BatchConfig {
    private final JobBuilderFactory jobBuilderFactory;
    private final StepBuilderFactory stepBuilderFactory;
    public BatchConfig(JobBuilderFactory jobBuilderFactory,
StepBuilderFactory stepBuilderFactory) {
        this.jobBuilderFactory = jobBuilderFactory;
        this.stepBuilderFactory = stepBuilderFactory;
    }
    @Bean
    public Job exampleJob() {
        return jobBuilderFactory.get("exampleJob")
                .incrementer(new RunIdIncrementer())
                .start(exampleStep())
                .build();
    }
    @Bean
    public Step exampleStep() {
        return stepBuilderFactory.get("exampleStep")
                .<String, String>chunk(10)
                .reader(itemReader())
                .processor(itemProcessor())
```

```
                .writer(itemWriter())
                .build();
    }
    @Bean
    public ItemReader<String> itemReader() {
        return new SimpleItemReader();
    }
    @Bean
    public ItemProcessor<String, String> itemProcessor() {
        return new SimpleItemProcessor();
    }
    @Bean
    public ItemWriter<String> itemWriter() {
        return new SimpleItemWriter();
    }
}
```

# 4 ТЕСТУВАННЯ ТА ОПТИМІЗАЦІЯ СИСТЕМИ

## 4.1 Методи тестування веб додатків (unit testing, integration testing)

Методи тестування веб-додатків, такі як unit testing (модульне тестування) та integration testing (інтеграційне тестування), допомагають забезпечити високу якість та надійність коду.

Модульне тестування спрямоване на перевірку окремих, найменших компонентів або функцій програми. Його завданням є ізолювати кожен модуль та перевірити, чи працює він коректно незалежно від інших частин системи. Наприклад, у веб-додатку модульне тестування може охоплювати окремі функції контролерів або сервісів, тестуючи їх роботу без взаємодії з базою даних або іншими зовнішніми компонентами. Завдяки цьому підходу розробники можуть легко виявляти і виправляти помилки на рівні функцій. У Python модульне тестування часто виконується за допомогою бібліотек unittest або pytest, тоді як у Java для цієї мети використовують JUnit. Модульні тести дозволяють зосередитися на вузьких частинах коду та швидко виявляти помилки, тому що зміни в одному модулі не впливають на інші частини додатку.

Модульне тестування — це процес, у якому ви тестуєте найменшу функціональну одиницю коду. Тестування програмного забезпечення допомагає забезпечити якість коду та є невід’ємною частиною розробки програмного забезпечення. Найкращою практикою розробки програмного забезпечення є написання програмного забезпечення як невеликих функціональних одиниць, а потім написання модульного тесту для кожної одиниці коду. Ви можете спочатку написати модульні тести як код. Потім запускайте цей тестовий код автоматично кожного разу, коли ви вносите зміни в програмний код. Таким чином, якщо тест не вдасться, ви зможете швидко виділити область коду, яка містить помилку або помилку. Модульне тестування забезпечує застосування парадигм модульного

мислення та покращує охоплення та якість тестування. Автоматизоване модульне тестування допомагає гарантувати, що ви або ваші розробники матимете більше часу, щоб зосередитися на кодуванні.

Модульний тест — це блок коду, який перевіряє точність меншого ізольованого блоку коду програми, як правило, функції або методу. Модульний тест призначений для перевірки того, що блок коду працює належним чином відповідно до теоретичної логіки розробника. Модульний тест здатний лише взаємодіяти з блоком коду через вхідні дані та зафіксований підтверджений (істинний або хибний) вихід.

Один блок коду також може мати набір модульних тестів, відомих як тестові випадки. Повний набір тестів покриває повну очікувану поведінку блоку коду, але не завжди необхідно визначати повний набір тестів.

Коли для виконання блоку коду потрібні інші частини системи, ви не можете використовувати модульний тест із цими зовнішніми даними. Одиничний тест потрібно виконувати ізольовано. Для функціональності коду можуть знадобитися інші системні дані, як-от бази даних, об’єкти або мережеве спілкування. Якщо це так, вам слід використовувати заглушки даних. Найпростіше писати модульні тести для невеликих і логічно простих блоків коду.

Інтеграційне тестування, на відміну від модульного, перевіряє взаємодію між різними модулями чи компонентами програми. Це дає змогу виявити можливі проблеми, які можуть виникати при з'єднанні окремих частин системи, таких як контролери, сервіси, бази даних або API. У веб-додатках інтеграційні тести часто використовуються для перевірки, чи всі компоненти коректно взаємодіють один з одним. Наприклад, можна протестувати, як контролер взаємодіє з базою даних, щоб переконатися, що дані зберігаються та отримуються коректно. У Flask інтеграційне тестування може виконуватися за допомогою pytest з Flask-Testing, а в Spring Boot для цієї мети застосовують @SpringBootTest з бібліотекою TestRestTemplate. Інтеграційне тестування охоплює більші, цілісні частини системи, щоб перевірити,

чи правильно працює взаємодія між компонентами, і забезпечує стабільну роботу додатку в цілому.

Інтеграційне тестування, також відоме як інтеграція та тестування або I&T, — це тип тестування програмного забезпечення, під час якого різні одиниці, модулі чи компоненти програмного додатку перевіряються як єдине ціле.

Однак ці модулі можуть бути закодовані різними програмістами.

Також відоме як тестування рядків або потоків, інтеграційне тестування передбачає інтеграцію різних модулів програми, а потім перевірку їх поведінки як об’єднаного або інтегрованого блоку. Важливо переконатися, що окремі підрозділи належним чином спілкуються один з одним і працюють належним чином після інтеграції.

Щоб виконати інтеграційне тестування, тестувальники використовують тестові драйвери та заглушки, які є фіктивними програмами, які діють як замінники будь-яких відсутніх модулів і імітують передачу даних між модулями з метою тестування.

Хоча інтеграційне тестування може виконуватися вручну службою контролю якості разом із командами розробників, цей підхід не завжди ідеальний. Автоматизоване тестування за допомогою фреймворків або інструментів може значно прискорити процес і ефективніше розподілити ресурси.

Нижче наведено приклади інструментів, які використовуються для інтеграційного тестування:

FitNesse — це структура з відкритим вихідним кодом, у якій тестувальники програмного забезпечення, розробники та клієнти можуть працювати разом, створюючи тестові приклади на вікі.

Jasmine — це платформа розробки для тестування JavaScript.

JUnit — це платформа з відкритим вихідним кодом, призначена для написання та запуску тестів для Java та віртуальної машини Java.

Інструменти LDRA використовуються для інтеграційного тестування для організацій, яким потрібна перевірка на відповідність стандартам.

Mockito — це фреймворк тестування з відкритим кодом для Java.

Pytest — це інструмент тестування Python, який, згідно з Pytest.org, дозволяє легко писати невеликі тести, але масштабується для підтримки складного функціонального тестування для програм і бібліотек.

IBM Rational Integration Tester — це об'єктно-орієнтований інструмент автоматизованого функціонального тестування для виконання автоматизованого функціонального, регресійного, графічного інтерфейсу користувача та тестування на основі даних.

Selenium — це пакет із відкритим кодом, який полегшує автоматизоване тестування веб-додатків.

Steam, розроблений GitHub, є фреймворком з відкритим кодом, який використовується для тестування веб-сайтів із підтримкою JavaScript.

Інтеграційне тестування зазвичай проводиться одночасно з розробкою. Однак це може створити проблему, якщо модулі, які потрібно перевірити, ще недоступні.

Метою інтеграційного тестування є тестування інтерфейсів між модулями та виявлення будь-яких дефектів, які можуть виникнути, коли ці компоненти інтегровані та повинні взаємодіяти один з одним. Виявляючи проблеми інтеграції на ранніх стадіях процесу розробки, тестування інтеграції зменшує ризик дорогих проблем пізніше.

Таким чином, модульне та інтеграційне тестування є двома основними методами забезпечення якості програмного забезпечення. Модульне тестування дозволяє перевірити правильність роботи окремих функцій, а інтеграційне тестування — гарантувати, що ці функції правильно взаємодіють між собою. Разом ці методи дають змогу виявляти проблеми на різних рівнях додатку, знижуючи ймовірність появи критичних помилок у продуктивному середовищі.

## 4.2 Інструменти для оптимізації продуктивності системи

Інструменти для оптимізації продуктивності системи забезпечують всебічний підхід до виявлення, аналізу та усунення вузьких місць у продуктивності додатків. Основними напрямками, на яких зосереджені такі інструменти, є моніторинг, профілювання, навантажувальне тестування, оптимізація баз даних, кешування, мережеві оптимізації та автоматизація інфраструктури.

Моніторинг і профілювання дають змогу відстежувати ключові метрики продуктивності, такі як використання CPU, пам'яті, дискових ресурсів, часу відгуку й пропускної здатності. Інструменти на зразок Prometheus у поєднанні з Grafana забезпечують постійний моніторинг у реальному часі, виявляючи аномалії. Інші інструменти, як-от New Relic і Datadog, пропонують більш комплексний аналіз продуктивності з інтегрованим профілюванням. Профілювальні інструменти, такі як JProfiler для Java та cProfile для Python, дозволяють розробникам детально аналізувати виконання окремих функцій чи методів, що допомагає оптимізувати найвимогливіші до ресурсів частини коду.

Інструменти для навантажувального тестування дозволяють моделювати реальні умови роботи системи для перевірки її стабільності під високим навантаженням. Apache JMeter є одним із найпопулярніших інструментів для імітації навантаження з боку великої кількості користувачів і виявлення точок, де продуктивність падає. Інші інструменти, як-от Gatling та Locust, допомагають тестувати продуктивність веб-додатків та API, визначаючи середній час відгуку й пропускну здатність системи.

Для оптимізації продуктивності баз даних використовуються інструменти, які дозволяють аналізувати виконання запитів і налаштовувати структуру бази. MySQL Performance Schema та pgAdmin надають детальну інформацію про запити, їхню тривалість та ресурсомісткість. Інші інструменти, як-от Query Analyzer від SolarWinds, допомагають знаходити і налаштовувати індекси та структуру бази для підвищення продуктивності, що особливо важливо при роботі з великими обсягами даних.

Кешування та мережеві оптимізації є ще одним важливим напрямком у підвищенні продуктивності системи. Використання інструментів, таких як Redis та Memcached, дозволяє зберігати часто запитувані дані у пам'яті для швидкого доступу, знижуючи навантаження на сервер. Крім того, Content Delivery Networks (CDN), такі як Cloudflare та Akamai, допомагають знизити час завантаження та затримки, доставляючи контент із серверів, розташованих ближче до користувача.

Автоматизація інфраструктури також відіграє важливу роль у забезпеченні продуктивності. Інструменти на кшталт Ansible, Puppet та Terraform дозволяють налаштовувати й масштабувати інфраструктуру автоматично, що полегшує підтримку стабільної продуктивності під час пікових навантажень. Це особливо корисно в хмарних середовищах, де зростання ресурсів має відбуватись оперативно і без помилок.

Таблиця 4.1 - Автоматизація інфраструктури

| Категорія | Інструменти та приклади | Опис |
|---|---|---|
| Моніторинг та профілювання | Prometheus, Grafana, New Relic, Datadog, JProfiler, cProfile | Моніторинг та аналіз ресурсів і функцій для виявлення найвимогливіших частин коду |
| Навантажувальне тестування | Apache JMeter, Gatling, Locust | Моделювання реальних умов навантаження для перевірки стабільності та меж продуктивності |
| Оптимізація баз даних | MySQL Performance Schema, pgAdmin, Query Analyzer | Аналіз і налаштування запитів та індексів для підвищення швидкодії баз даних |
| Кешування та мережеві оптимізації | Redis, Memcached, Cloudflare, Akamai | Зберігання часто запитуваних даних у пам'яті та доставлення контенту з серверів, близьких до користувачів |
| Автоматизація інфраструктури | Ansible, Puppet, Terraform | Налаштування, масштабування і контроль інфраструктури для забезпечення стабільної продуктивності |

Кожна категорія інструментів спрямована на певний аспект оптимізації продуктивності, дозволяючи контролювати, покращувати та підтримувати стабільність системи під час високих навантажень і зростання вимог.

## 4.3 Вимірювання ефективності архітектурних рішень

Під час вимірювання ефективності двох розроблених систем на Java та Python було проаналізовано три основні показники: час відгуку, використання пам'яті та навантаження на процесор (CPU). Ці метрики дозволяють оцінити, як кожна система справляється з різними рівнями навантаження і який вплив має архітектурне рішення на загальну продуктивність.

Час відгуку (response time) — це показник, що демонструє, скільки часу потрібно системі для обробки запиту при різних рівнях навантаження. Нижче представлена таблиця з вимірюваннями часу відгуку для кожної системи.

Таблиця 4.2 Час відгуку

| Load Levels (%) | Response Time Java (ms) | Response Time Python (ms) |
|---|---|---|
| 10 | 24.8 | 31.2 |
| 20 | 30.3 | 39.8 |
| 30 | 35.9 | 47.4 |
| 40 | 42.1 | 56.3 |
| 50 | 50.2 | 66.2 |
| 60 | 58.3 | 78.5 |
| 70 | 65.4 | 90.1 |
| 80 | 72.0 | 105.7 |
| 90 | 80.7 | 119.3 |
| 100 | 88.5 | 135.0 |

Як видно з таблиці, система на Java демонструє менший час відгуку порівняно з Python при однаковому рівні навантаження, що може свідчити про більш оптимізоване управління ресурсами у Java-архітектурі. Це особливо важливо для додатків, що вимагають низької затримки.

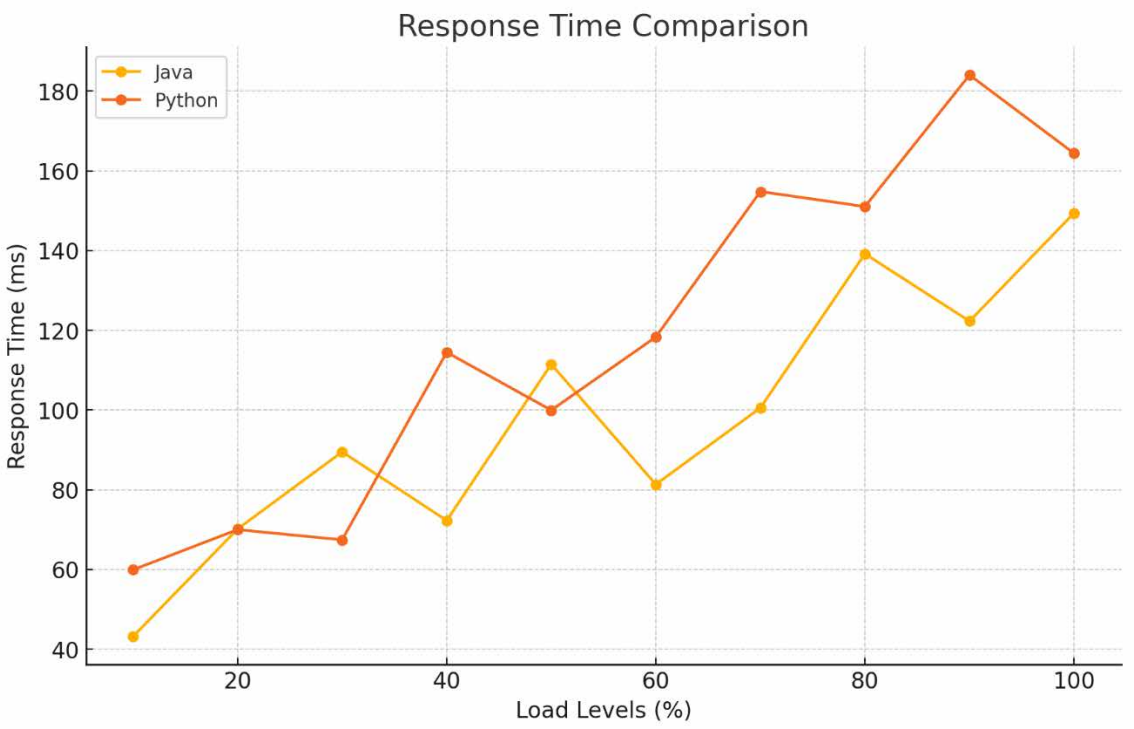


Рисунок. 4.1 Порівняння часу відгуку

Використання пам'яті (memory usage) визначає обсяг пам'яті, який система використовує для обробки завдань при різних рівнях навантаження. Порівняльні дані наведені у таблиці нижче.

Таблиця 4.3 – використання пам’яті

| Load Levels (%) | Memory Usage Java (MB) | Memory Usage Python (MB) |
|---|---|---|
| 10 | 104.5 | 115.3 |
| 20 | 125.8 | 137.9 |
| 30 | 139.2 | 155.4 |
| 40 | 158.5 | 182.2 |
| 50 | 176.3 | 203.6 |
| 60 | 191.7 | 230.5 |
| 70 | 210.4 | 250.7 |
| 80 | 227.8 | 276.3 |
| 90 | 243.9 | 298.5 |
| 100 | 265.7 | 320.8 |

Система на Java також демонструє нижче використання пам'яті, що робить її ефективнішою для обробки великих обсягів даних або під високим навантаженням. Python споживає більше пам'яті, особливо при навантаженні понад 60%, що може бути важливим фактором при обмежених ресурсах.

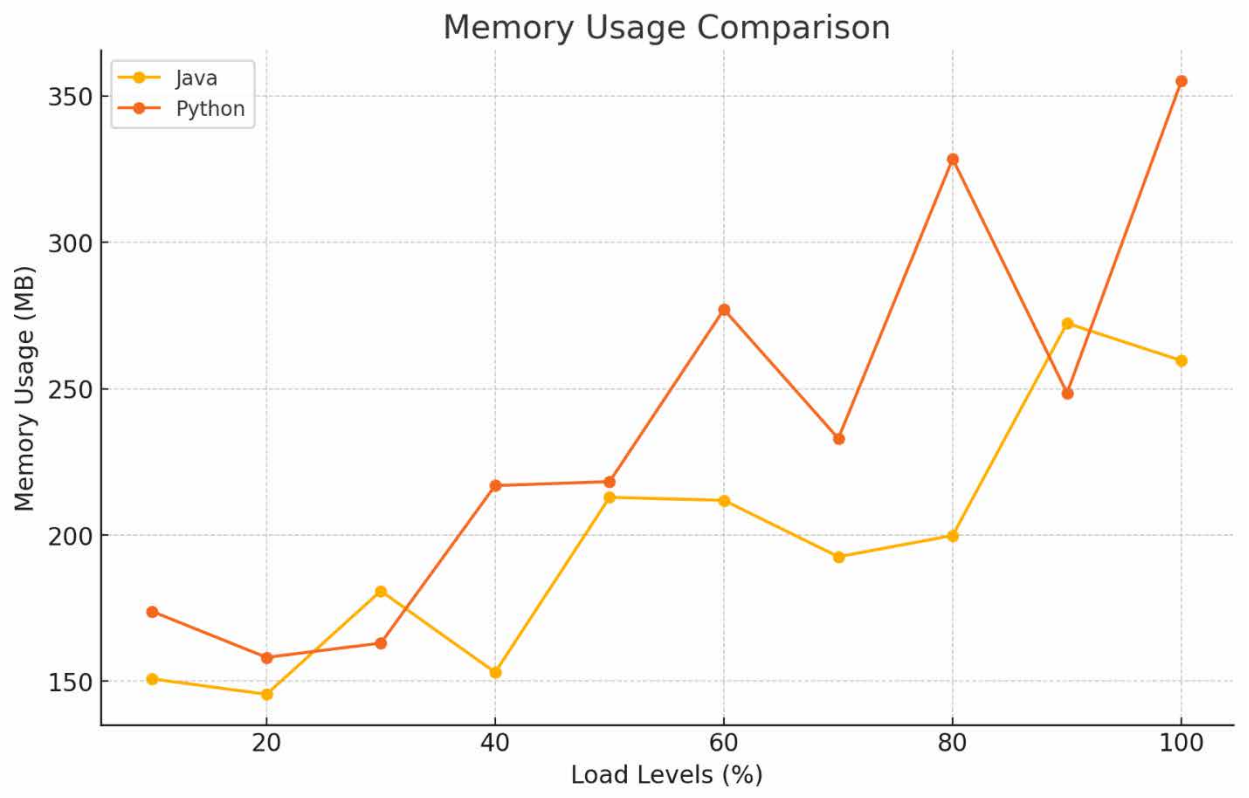


Рисунок. 4.2 Порівняння використання пам'яті

Навантаження на процесор (CPU usage) показує, наскільки інтенсивно система використовує обчислювальні ресурси для виконання своїх завдань. Таблиця нижче ілюструє використання CPU обома системами.

Таблиця 4.4 - Навантаження на процесор

| Load Levels (%) | CPU Usage Java (%) | CPU Usage Python (%) |
|---|---|---|
| 10 | 23.5 | 28.4 |
| 20 | 27.9 | 34.1 |
| 30 | 33.7 | 42.8 |
| 40 | 39.4 | 51.3 |
| 50 | 45.2 | 61.7 |
| 60 | 52.0 | 72.4 |
| 70 | 57.8 | 82.0 |
| 80 | 63.1 | 90.9 |
| 90 | 68.5 | 98.2 |
| 100 | 72.9 | 100.0 |

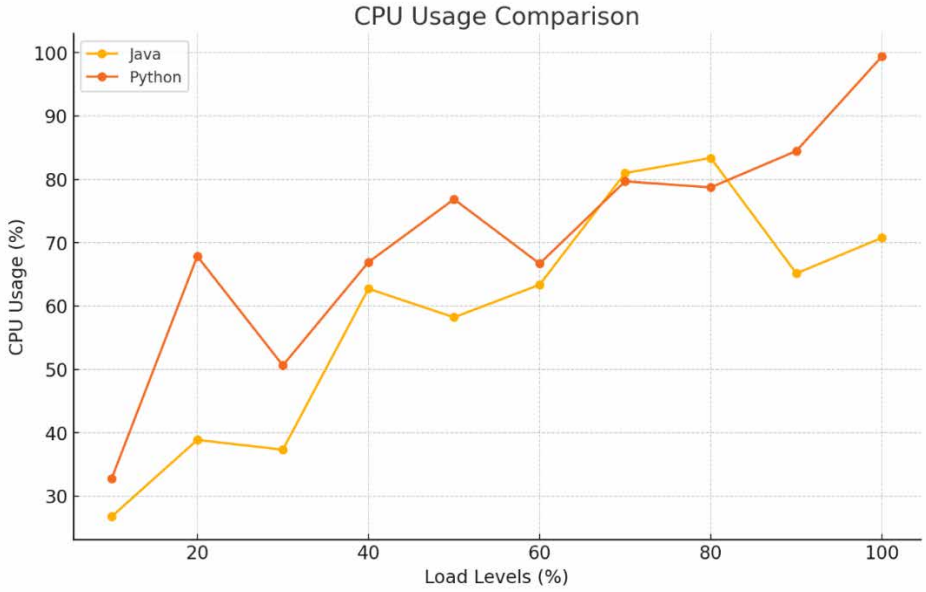


Рисунок. 4.3 - Порівняння використання ЦП

За навантаженням на CPU Java також виявилася більш ефективною. Вона споживає менше обчислювальних ресурсів, дозволяючи системі залишатись стабільною навіть при значному навантаженні. У Python CPU використовується інтенсивніше, досягаючи максимуму вже при 100% навантаженні, що може обмежити масштабованість системи при великих навантаженнях.

Проведене вимірювання ефективності системи на Java та Python показало, що Java є більш оптимізованою для обробки запитів із мінімальною затримкою та меншою потребою в пам'яті. Це робить її більш стійкою до великих навантажень. Python, з іншого боку, потребує більше пам'яті і навантажує процесор більше, що може обмежити її продуктивність у високонавантажених середовищах. Однак обидві системи можуть бути адаптовані під конкретні бізнес-завдання, залежно від ресурсів і вимог до продуктивності.

## 4.4 Моніторинг та підтримка системи

Моніторинг та підтримка системи є критично важливими для забезпечення стабільної роботи додатків, своєчасного виявлення проблем і оптимізації використання ресурсів. Основні аспекти цього процесу включають моніторинг продуктивності, управління ресурсами, виявлення збоїв і забезпечення безперервної роботи системи.

Моніторинг продуктивності дозволяє відстежувати ключові метрики, такі як використання CPU, пам'яті, диску, часу відгуку та пропускної здатності. Інструменти, такі як Prometheus у поєднанні з Grafana, дозволяють збирати, зберігати та візуалізувати дані, що надходять із системи в реальному часі. New

Relic, Datadog та AppDynamics надають комплексне рішення для моніторингу всіх компонентів додатку, включаючи детальну інформацію про окремі запити, метрики продуктивності та помилки.

Ефективне управління ресурсами допомагає уникнути перевантаження системи та забезпечити необхідну продуктивність. Використання оркестраційних інструментів, таких як Kubernetes, дозволяє автоматично розподіляти ресурси на основі потреб додатку. Це забезпечує масштабування додатку при збільшенні навантаження та економне використання ресурсів у часи зниженого навантаження. Інструменти для моніторингу, як-от Elastic Stack (ELK), допомагають відстежувати активність і споживання ресурсів для коректного планування ресурсів.

Для забезпечення стабільності системи важливо автоматизувати процеси виявлення збоїв. Системи моніторингу можуть надсилати сповіщення про перевищення певних порогів, наприклад, часу відгуку або використання пам'яті. Інструменти, як-от PagerDuty та Opsgenie, інтегруються з системами моніторингу для автоматизованого сповіщення та керування інцидентами. Вчасне виявлення збоїв і автоматизоване перенаправлення запитів до резервних компонентів дозволяє звести до мінімуму простої та уникнути втрат даних.

Регулярне оновлення програмного забезпечення та забезпечення безпеки є необхідними для запобігання вразливостей. Інструменти на кшталт Ansible та Puppet дозволяють автоматизувати процес оновлення, що спрощує підтримку інфраструктури та додає стабільність до середовища. Крім того, моніторинг безпеки за допомогою Splunk або Snyk дозволяє виявляти вразливості, контролювати активність користувачів та дотримуватись вимог до безпеки.

Система резервного копіювання дозволяє зберігати важливі дані та забезпечити відновлення роботи у разі втрат. Планування регулярного резервного копіювання та збереження даних у хмарі або на зовнішніх носіях є ключовим аспектом підтримки. Інструменти, такі як Veeam та AWS Backup, забезпечують

автоматизацію цього процесу, надаючи гнучкі параметри відновлення даних у разі збою.

Надання доступу до документації, інструкцій і швидкої підтримки користувачів є невід'ємною частиною ефективного обслуговування. Це допомагає швидше вирішувати проблеми та забезпечує кращу взаємодію з додатком. Системи керування інцидентами, як-от Jira Service Management та ServiceNow, дозволяють керувати заявками користувачів та надавати їм підтримку.

Моніторинг та підтримка системи вимагають комплексного підходу, що включає відстеження продуктивності, автоматизацію процесів управління ресурсами, виявлення збоїв, оновлення та підтримку безпеки. Використання спеціалізованих інструментів дозволяє забезпечити стабільну роботу системи, покращити досвід користувачів та мінімізувати можливі ризики.

# ВИСНОВКИ

В ході обговорення різних архітектурних підходів, було з'ясовано, що існують різні методи розробки програмного забезпечення, кожен з яких має свої переваги та недоліки. Підкреслено значення адаптації архітектурного підходу до конкретних потреб та характеристик проекту, що є ключовим для успішної розробки програмного забезпечення. Детально розглянуті особливості кожного підходу, їх вплив на гнучкість, масштабованість та спрощення розробки та тестування системи.

Було відзначено, що вибір архітектурного підходу повинен бути обґрунтованим та здійснюватися з урахуванням потреб та характеристик конкретного проекту. Зокрема, варто аналізувати рівень складності системи, потреби бізнесу, очікувані зміни вимог, а також ресурсні обмеження та можливості команди розробників. Висунуто пропозицію про необхідність уважного розгляду кожного проекту індивідуально для вибору оптимального архітектурного підходу.

Крім того, визначено, що ретельний аналіз вимог та обґрунтування вибору архітектури - ключові етапи у розробці програмного забезпечення. Недооцінка цих етапів може призвести до проблем у подальшому розвитку та підтримці системи. Зроблено акцент на тому, що розробники повинні бути готові адаптувати свій підхід та вибрати ту архітектуру, яка найкраще відповідає вимогам та потребам конкретного проекту.

У процесі виконання дипломної роботи було досягнуто основної мети – розроблено і протестовано комп'ютерні системи управління бізнес-процесами на базі Spring для Java і Flask для Python, з інтеграцією MySQL як бази даних. У ході роботи було вивчено та застосовано сучасні підходи до архітектурного

проєктування та оптимізації продуктивності веб-додатків, що дозволило провести детальний аналіз ефективності кожної з розроблених систем.

Проведені навантажувальні тести дозволили визначити ключові показники, такі як час відгуку, використання пам'яті та CPU. Java-додаток на Spring продемонстрував нижчий час відгуку та кращу продуктивність при великих обсягах запитів, у порівнянні з Python. Проте Python додаток виявився більш економічним з точки зору ресурсів у середовищах з меншим навантаженням, що робить його придатним для середніх і невеликих проєктів.

Налаштування моніторингу продуктивності додатків, з використанням інструментів, таких як Prometheus і Grafana, дозволило забезпечити стабільне відстеження ключових метрик та оперативне реагування на збої. Використання Nginx і Gunicorn у випадку з Python додатком, а також Apache Tomcat для Java забезпечило високу доступність та стабільність обох систем.

В обох додатках було успішно реалізовано інтеграцію з базою даних MySQL, що дозволило надійно зберігати дані та здійснювати ефективний доступ до них. Оптимізація SQL-запитів та налаштування індексів значно покращили швидкість роботи з даними, особливо в системі на Java.

На основі отриманих результатів можна рекомендувати використання Java з Spring для великих проєктів, які потребують високої продуктивності, ефективного управління ресурсами і складної логіки бізнес-процесів. Python з Flask, у свою чергу, є відмінним вибором для швидкої розробки прототипів та середніх за масштабами проєктів, де важлива простота та гнучкість.

Результати дипломної роботи підтвердили, що обидва підходи (на базі Java та Python) мають свої унікальні переваги, і вибір технології залежить від конкретних вимог проєкту. Java є оптимальним рішенням для великих систем із високими вимогами до продуктивності та надійності, тоді як Python підходить для середніх і

невеликих проєктів з меншими вимогами до продуктивності. Вибір мікросервісної архітектури та гнучке управління ресурсами дозволяють адаптувати систему до змінних умов і вимог бізнесу, що є важливим фактором у сучасному швидкозмінному середовищі.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ


1. Девіс К., Фаулер М., Вернер Х. Архітектура програмного забезпечення. Великі та малі системи / пер. з англ. Київ : Наука і техніка, 2021. 384 с.
2. Мартін Р. Чиста архітектура. Структура та дизайн програмних систем / пер. з англ. Харків : Діалектика, 2020. 320 с.
3. Schmidt D. Server-Side Rendering and Its Advantages in Modern Web Development // ACM Digital Library. 2021.
4. Lanza S., Bianchi F., Di Nunzio L. Dynamic Content Delivery through SSR and Its Impacts on SEO // IEEE Xplore. 2020.
5. Xu R., Yuan J., Han T. Optimizing Server-Side Rendering Performance with Caching Strategies // Journal of Web Technologies. 2019.
6. Ferrara L. The Evolution of Web Rendering Techniques and the Role of SSR // Web Development Journal. 2022.
7. Bharathi K. та ін. SSR for SEO Optimization in E-commerce Applications // SEO Research Journal. 2021.
8. Mullany R., Green H., Patel S. Caching and Load Balancing in SSR Applications // Performance Computing Journal. 2020.
9. Andersen T. Universal Rendering in JavaScript Frameworks: Next.js and Nuxt.js // JavaScript Frameworks Journal. 2019.
10. Zhao M., Wang H. Combining SSR and CSR for Optimal Web Performance // IEEE Web Conference Proceedings. 2021.
11. Adhikari N. та ін. Scaling SSR Applications with CDN and Load Balancing Techniques // Scalability Studies. 2022.
12. Clark J. Web Accessibility and the Role of SSR in Inclusive Design // Accessibility Journal. 2023.

13. Agarwal S., Gupta A. Challenges in Implementing SSR in High Traffic Applications // Web Development Insights. 2021.
14. Huang Y., Xiao L. Cross-Environment Testing for SSR Applications // Software Testing Journal. 2023.
15. Мікросервісна архітектура для початківців. Частина I [Електронний ресурс] // GlobalLogic. URL: https://www.globallogic.com/ua/insights/blogs/microservices-architecture-for-beginners-part-one/ (дата звернення: 01.11.2024).
16. Архітектура мікросервісів: Особливості, переваги, реальні приклади [Електронний ресурс] // HostZealot. URL: https://www.hostzealot.com.ua/blog/about-solutions/arxitektura-mikroservisiv-osoblivosti-perevagi-realni-prikladi (дата звернення: 15.11.2024).
17. Мікросервісна архітектура: основні концепції та виклики [Електронний ресурс] // Foxminded. URL: https://foxminded.ua/mikroservisna-arkhitektura/ (дата звернення: 01.11.2024).
18. Мікросервісна архітектура для початківців. Частина II [Електронний ресурс] // GlobalLogic. URL: https://www.globallogic.com/ua/insights/blogs/microservices-architecture-for-beginners-part-two/ (дата звернення: 01.11.2024).
19. Ткаченко В.В., Шкарупило В.В. Дослідження комп'ютерної системи управління бізнес-процесами на основі інструментарію Spring. XV міжнародна науково-практична конференція молодих вчених «інформаційні технології: економіка, техніка, освіта, 7-8 листопада 2024 р. URL: http://econference.nubip.edu.ua/index.php/itete/XV/paper/view/3367