

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Дослідження принципів побудови архітектури та
дизайну масштабних веб-застосунків»**

Виконав студент групи ІПЗм-21-1
СІКАЙЛО Владислав Олегович

Керівник роботи:
КРАВЧЕНКО Світлана Миколаївна

Рецензент:
ВЛАСЕНКО Олег Васильович

Житомир – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
 ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
 ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
 КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

зав. кафедри інженерії
 програмного забезпечення

Тетяна ВАКАЛЮК

«18» вересня 2023 р.

ЗАВДАННЯ на кваліфікаційну роботу

Здобувач вищої освіти: **СІКАЙЛО Владислав Олегович**

Керівник роботи: **КРАВЧЕНКО Світлана Миколаївна**

Тема роботи: **«Дослідження принципів побудови архітектури та дизайну масштабних веб-застосунків»**,

затверджена Наказом закладу вищої освіти від «18» вересня 2023 р. № 479/с

Вихідні дані для роботи: **Предметом дослідження є процес побудови архітектури та дизайну масштабних веб-застосунків. Об'єктом дослідження є конкретні аспекти архітектури та дизайну масштабних веб-застосунків.**

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Кравченко С.М.	22.09.2023р.	22.09.2023р.
2	Кравченко С.М.	22.09.2023р.	22.09.2023р.
3	Кравченко С.М.	22.09.2023р.	22.09.2023р.

РЕФЕРАТ

Кваліфікаційна робота присвячена науковому дослідженню принципів побудови архітектури та дизайну масштабних веб-застосунків.

Кваліфікаційна робота складається з прототипу програмного комплексу який побудований використовуючи декілька архітектурних стилей та пояснювальну записку з детальним теоритичним дослідженням проблематики та дизайну архітектури.

Пояснювальна записка складається з вступу, трьох розділів, висновків та списку використаної літератури, що містить 25 найменувань. Також в роботі наведено 29 рисунків та 5 таблиць.

Кваліфікаційна робота складається з трьох розділів.

В першому розділі було розкрито проблематику дизайну та архітектури програмного забезпечення, а також вибору коректного архітектурного стилю.

В другому розділі було детально висвітлено поняття чистого коду як основи для конструювання архітектури системи. Також проведено докладний огляд різних архітектурних стилів та їх топологій, включаючи оцінку їх характеристик.

В третьому розділі було здійснено практичне впровадження стилів архітектури та дизайну систем. Описано методи практичного впровадження архітектурних стилів та приведено приклади дизайну, де детально розглянуто застосунок-прототип з використанням мішаного архітектурного стилю, який об'єднує Microservices, Layered Architecture та Event-Driven architecture.

Висновок містить в собі підсумки проведеного дослідження.

Ключові слова: АРХІТЕКТУРА, ДИЗАЙН, DDD, ЮНІТ-ТЕСТУВАННЯ, TDD, МІКРОСЕРВІСИ, БАГАТОШАРОВІСТЬ, МОНОЛІТ, МОДУЛЬНІСТЬ.

ABSTRACT

The qualification work is dedicated to a scientific investigation of the principles of building architecture and design for large-scale web applications. The qualification work consists of a prototype of a software system built using several architectural styles and an explanatory note with a detailed theoretical study of the issues of architecture design.

The explanatory note consists of an introduction, three chapters, conclusions, and a list of references, which includes 25 titles. Additionally, the work includes 29 figures and 5 tables.

The qualification work is divided into three chapters.

In the first chapter, the problems of software design and architecture, as well as the selection of the correct architectural style, were explained and discussed.

The second chapter provided a detailed overview of the concept of clean code as the basis for constructing the system's architecture. A thorough examination of various architectural styles and their topologies was also conducted, including an assessment of their characteristics.

The third chapter implemented the practical application of architectural styles and system design. Methods for the practical implementation of architectural styles were described, and examples of design were provided. The application prototype was examined in detail, utilizing a mixed architectural style that combines Microservices, Layered Architecture, and Event-Driven architecture.

The conclusion contains the summaries of the conducted research.

Keywords: ARCHITECTURE, DESIGN, DDD, UNIT-TESTING, TDD, MICROSERVICES, MULTILAYERING, MONOLITH, MODULARITY.

ЗМІСТ

РЕФЕРАТ	3
ABSTRACT	4
ВСТУП	6
РОЗДІЛ 1. ГНУЧКИЙ ДИЗАЙН ТА СТИЛІ АРХІТЕКТУРИ	9
1.1. Проблематика дизайну та архітектури програмного забезпечення	9
1.2. Проблематика вибору архітектурних стилів	14
1.3. Постановка задачі дослідження	17
Висновки до першого розділу	19
РОЗДІЛ 2. АНАЛІЗ ІСНУЮЧИХ АРХІТЕКТУРНИХ СТИЛІВ ТА ОЦІНКА ЇХ ХАРАКТЕРИСТИК	20
2.1. Поняття чистого коду як фундаменту для архітектури системи	20
2.2. Архітектурні стилі та їх топології	23
2.3. Оцінка архітектурних характеристик	38
Висновки до другого розділу	45
РОЗДІЛ 3. ПРАКТИЧНЕ ВПРОВАДЖЕННЯ СТИЛІВ АРХІТЕКТУРИ ТА ДИЗАЙНУ	47
3.1. Методи практичного впровадження архітектурних стилів	47
3.2. Приклади дизайну систем	49
3.3. Застосунок-прототип з використанням мішаного архітектурного стилю	52
Висновки до третього розділу	62
ВИСНОВКИ	63
ЛІТЕРАТУРА	64

ВСТУП

Актуальність теми. Тема дослідження є актуальною в сучасному інформаційному суспільстві, де роль веб-застосунків надзвичайно важлива для різних галузей людської діяльності. З кожним днем кількість користувачів мережі Інтернет зростає, і разом з цим зростають вимоги до ефективності, надійності та масштабованості веб-застосунків. Проблема побудова архітектури та дизайну веб-застосунків є актуальною, оскільки масштаб таких застосунків тільки збільшується з розвитком інформаційних технологій.

Доцільно виділити наступні ключові аспекти актуальності дослідження:

- Підвищення продуктивності та масштабованості: Розробка ефективних архітектурних стилів дозволяє оптимізувати продуктивність та масштабованість інформаційних-систем. Розширення функціональності та підтримка зростання обсягів даних та трафіку стає складнішою задачею, і правильно обрана архітектура може допомогти покращити ці параметри.

- Забезпечення безпеки: Безпека є однією з найбільш важливих аспектів розробки програмного забезпечення. Архітектурні стилі можуть визначати ефективні методи обробки та захисту від різноманітних загроз, таких як атаки на безпеку даних, зловживання прав або інші види кіберзлочинності.

- Сумісність із сучасними технологіями: Із постійним розвитком технологій з'являються нові вимоги та можливості. Актуальні дослідження архітектурних стилів дозволяють враховувати та впроваджувати інновації, такі як обчислення в хмарі, блокчейн, штучний інтелект тощо.

- Оптимізація витрат ресурсів: Розробка бекенд-систем часто пов'язана з оптимізацією використання ресурсів, таких як обчислювальна потужність, пам'ять та мережеві ресурси. Архітектурні стилі можуть надавати ефективні підходи до управління ресурсами та забезпечувати оптимальну роботу системи.

- Покращення тестування та супроводу: Архітектурні рішення можуть суттєво впливати на процеси тестування та супроводу програмного забезпечення. Ефективні архітектурні стилі сприяють розробці стійких до помилок та легко супроводжуваних систем.

Мета роботи. Метою є дослідження принципів побудови архітектури та дизайну масштабних веб-застосунків з підтримкою можливості вдосконалення їхнього функціонування. Основною областю дослідження є розробка оптимальних стратегій та рекомендацій для створення веб-застосунків, які забезпечують високу продуктивність, безпеку та масштабованість.

Завдання атестаційної магістерської роботи. Завдання включають:

1. Аналіз існуючих підходів до побудови архітектури веб-застосунків.
2. Вивчення сучасних технологій та методів, що використовуються у розробці масштабних веб-застосунків.
3. Вивчення існуючих архітектурних стилів, їх детальне дослідження
4. Створення лістингу архітектурних стилів з описом доцільності використання.
5. Проведення логічного аналізу для оцінки їх ефективності, виставлення оцінок по кожній з характеристик.

Об'єктом дослідження є процес побудови архітектури та дизайну масштабних веб-застосунків, який включає в себе взаємодію різних компонентів системи, оптимізацію ресурсів та забезпечення високої якості обслуговування для користувачів.

Предметом дослідження є конкретні аспекти архітектури та дизайну масштабних веб-застосунків, такі як розподілені системи, бази даних, безпека, а також інтерфейс та взаємодія з користувачем.

Методи дослідження включають літературний аналіз, проведення експериментів та аналіз результатів. Ці підходи сприяють розумінню принципів побудови масштабних веб-застосунків.

Наукова новизна роботи проявляється у визначенні та систематизації лістингу оцінок архітектурних характеристик для різних стилів архітектури програмного забезпечення. Основною метою цього лістингу є спрощення складного процесу вибору оптимального архітектурного стилю залежно від вимог та особливостей конкретного проекту.

В першу чергу, наукова новизна виявляється у врахуванні не лише технічних аспектів, а й функціональних та нефункціональних вимог. Такий глобальний підхід дозволяє розробникам не лише забезпечити ефективну реалізацію бізнес вимог, але й врахувати важливі аспекти (нефункціональні вимоги), такі як безпека, масштабованість та гнучкість.

Крім того, робота включає в себе визначення вагомості кожної характеристики в контексті конкретного стилю архітектури. Наприклад, для одних проектів може бути ключовою вагомістю продуктивність, тоді як для інших – безпека чи гнучкість. Такий аспект дозволяє розробникам та архітекторам персоналізувати свій вибір відповідно до конкретних потреб проекту.

Загалом, наукова новизна цієї роботи полягає у створенні методології для об'єктивної оцінки та вибору архітектурного стилю, що враховує широкий спектр факторів та спрощує процес розробки інформаційних систем.

Апробація результатів дослідження. Отримані результати магістерської кваліфікаційної роботи були оприлюднені у V Всеукраїнській науково-технічній конференції: «Комп'ютерні технології: інновації, проблеми, рішення» [17]; III Всеукраїнській науково-практичній інтернет-конференції: «Сучасні комп'ютерні та інформаційні системи і технології» [18]

РОЗДІЛ 1. ГНУЧКИЙ ДИЗАЙН ТА СТИЛІ АРХІТЕКТУРИ

1.1. Проблематика дизайну та архітектури програмного забезпечення

В сучасному інформаційному суспільстві важливість дизайну архітектури програмного забезпечення набуває ще більшого значення, особливо у контексті постійних технологічних змін та вимог ринку. Існує переконання, що добре спроектована архітектура є ключовим фактором для успішності та довгострокової стабільності програмного продукту.

Архітектура програмного забезпечення - це високорівневий дизайн програмної системи, який визначає її структуру та взаємозв'язки між її основними компонентами. Це планування, яке визначає загальний спосіб побудови та організації програми або системи, забезпечуючи базові рамки для подальшого розвитку та утримання. Основні аспекти архітектури програмного забезпечення включають компоненти та модулі, де визначається, які частини системи будуть взаємодіяти між собою та як вони будуть структуровані. Архітектура також охоплює структуру та організацію системи, визначаючи, як будуть організовані компоненти та як вони будуть взаємодіяти. Це охоплює питання шарування, модульності та структурної ієрархії. Додатково, в рамках архітектури вирішуються питання вибору патернів та стилів програмування. Це включає в себе вибір оптимальних підходів та прийомів, які служать основою для створення програмного продукту з необхідною функціональністю та ефективністю.

Однією з ключових переваг оптимальної архітектури є здатність до подальшого розширення програм. Наприклад, компанії, що розвивають електронні комерційні платформи, повинні забезпечити не лише швидкість та безпеку, але й гнучкість системи для ефективної інтеграції нових функціональностей, таких як розширені опції платежів чи персоналізовані рекомендації для користувачів.

Проблематика розширення програмного забезпечення стає особливо актуальною в умовах постійних змін у бізнес-вимогах, технологічних та

функціональних вимогах користувачів. Розширення програми може зіткнутися з рядом викликів, особливо якщо архітектура не була належно спроектована з урахуванням цієї потреби. Однією з основних проблем може бути недостатня гнучкість або відсутність модульності в архітектурі програми. Якщо код не розділений на логічні модулі чи компоненти, додавання нового функціоналу може вимагати великих змін в існуючому коді. Багатошарова архітектура, така як Model-View-Controller (MVC) або Model-View-ViewModel (MVVM), може сприяти модульності, що полегшує внесення змін без значного впливу на інші частини системи. Також, розширення може впливати на продуктивність системи. Неправильний вибір алгоритмів чи недостатня оптимізація може призвести до втрати ефективності, особливо при роботі з великим обсягом даних. Використання архітектурних шаблонів, таких як CQRS (Command Query Responsibility Segregation) чи Event Sourcing, може сприяти оптимізації та забезпечити потрібний рівень продуктивності.

Отже, детальний аналіз архітектури програмного забезпечення, врахування принципів модульності, забезпечення слабкої залежності та використання оптимальних паттернів може допомогти уникнути проблем, пов'язаних з розширенням програми та забезпечити її стабільність та ефективність в умовах постійних змін. Вибір архітектури повинен тісно адаптуватися до бізнес-вимог. Наприклад, для розробників онлайн-ігор важливо врахувати не тільки функціональні вимоги (геймплей, графіка), але і нефункціональні аспекти, такі як масштабованість під час великої кількості одночасних гравців та безпека, щоб уникнути можливих атак чи обману в грі.

Програми без належної архітектури стають невпорядкованими та важкими у підтримці. Наприклад, у веб-розробці недбало розроблена архітектура може призвести до незручностей при оновленні та розширенні функціональності веб-сайту, а їх зміна може вимагати значних витрат часу та ресурсів.

Ігнорування архітектурних стилів у розробці програмного забезпечення може викликати ряд проблем, які впливають на якість, підтримку та

розширюваність системи. Архітектурні стилі визначають загальну структуру програми та взаємодію між її компонентами, що допомагає забезпечити модульність, гнучкість та ефективність. Розглянемо основні проблеми, які можуть виникнути при невикористанні архітектурних стилів:

1. Неструктурованість коду:

- Відсутність чіткої архітектурної концепції може призводити до непорозумінь у розробці, що веде до неструктурованості коду. Без визначення модульності та логічного розподілу функцій програма стає складною у розумінні та відлагодженні.

2. Змішування відповідальностей:

- Без використання архітектурних стилів може виникати схильність до змішування різних відповідальностей у компонентах програми. Наприклад, веб-додаток, який обробляє і логіку бізнес-логіку, і відображення, може стати важким у підтримці та розширенні.

3. Важкість управління залежностями:

- Невикористання архітектурних стилів може спричинити велику кількість жорстких залежностей між компонентами. Це ускладнює управління та зміну системи, оскільки зміна в одному компоненті може вимагати змін у багатьох інших частинах програми.

4. Проблеми з розширенням:

- Відсутність архітектурної структури може ускладнити додавання нового функціоналу або розширення існуючого. Без чіткого поділу на модулі і компоненти може знадобитися значний час та зусилля для інтеграції нових можливостей.

5. Погіршення масштабованості:

- Відсутність архітектурного стилю може призвести до складнощів у масштабованості системи. Збільшення обсягів та комплексності може призвести до зменшення продуктивності та збільшення ризику виникнення помилок.

Для розв'язання цих проблем рекомендується використовувати архітектурні стилі, які дозволяють чітко розділити відповідальності, полегшують розширення та сприяють модульності. Додатково, проблема може виникнути через невідокремленість компонентів або сильну залежність між ними. Якщо функції програми тісно пов'язані, вставка нового функціоналу може порушити роботу існуючих частин системи. Принципи, такі як інверсія залежностей (Dependency Inversion) або використання інтерфейсів, можуть допомогти вирішити цю проблему, забезпечуючи слабку залежність між компонентами. Проблема змішаних залежностей виникає, коли конкретні класи або модулі вбудовуються прямо в клієнтський код, створюючи тісні зв'язки між різними компонентами системи. Це веде до жорстких та негнучких систем, де будь-які зміни в одній частині можуть впливати на інші, і ускладнює тестування, підтримку та розширення програми. Рішення для проблеми змішаних залежностей надходить з принципу Dependency Inversion (зокрема, в рамках SOLID-принципів). Dependency Inversion Principle (DIP) встановлює, що модулі високого рівня не повинні залежати від модулів низького рівня, обидва повинні залежати від абстракцій; абстракції не повинні залежати від деталей реалізації, деталі реалізації повинні залежати від абстракцій.

Основні принципи Dependency Inversion:

1. Створення абстракцій:

- Визначення інтерфейсів чи абстрактних класів для взаємодії між компонентами програми, а не прив'язка до конкретних реалізацій.

2. Виокремлення абстракцій:

- Виділення абстракцій у власні компоненти, які можна використовувати для передачі залежностей.

3. Інверсія залежностей:

- Заміна жорстких залежностей на зворотні, де модулі високого рівня залежать від абстракцій, а конкретні реалізації залежать від цих абстракцій.

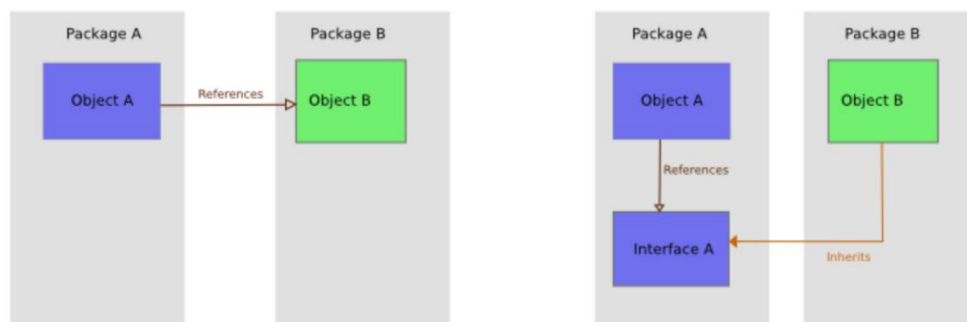


Рис. 1.1. Схема принципу інверсії залежностей

При використанні DIP, клієнтський код не знає про конкретні реалізації, а лише використовує абстракції, що забезпечує більшу гнучкість, зручність в тестуванні та зменшує залежність між різними частинами системи.

Розробка програмного забезпечення – це складний і динамічний процес, який вимагає глибокого розуміння принципів дизайну та архітектури. Слідування найкращим практикам галузі дизайну програмного забезпечення є важливим елементом успішної розробки програм та забезпечує стабільність, гнучкість та розширюваність системи. На перший погляд, виглядає привабливо розпочати розробку програми ітеративно та швидко навчатися на власних помилках. Однак найкращі результати досягаються завдяки глибокому плануванню та визначенню архітектури з самого початку проекту. Врахування бізнес-вимог, функціональних та нефункціональних вимог, а також вибір відповідної архітектурної парадигми стають критичними у визначенні фундаменту програмного продукту.

Важливо також дотримуватися принципів SOLID, зокрема Dependency Inversion Principle (DIP), щоб уникнути змішаних залежностей та забезпечити легку розширюваність та підтримку системи. Використання багатошарової архітектури та інших паттернів дозволяє зберігати код чистим, організованим та готовим до масштабування.

Усі ці аспекти сприяють створенню програмного забезпечення, яке не тільки відповідає поточним потребам бізнесу, але і готове до змін та розвитку в майбутньому. Незважаючи на те, що деякі витрати часу на планування та проектування можуть здатися великими на початку, це інвестиція, яка

відсиплється відмінною стабільністю та продуктивністю програми в подальшому. Такий підхід є критичним для досягнення високого рівня якості, гнучкості та успіху в сучасному світі розробки програмного забезпечення.

1.2. Проблематика вибору архітектурних стилів

Вибір правильної архітектури для програмного забезпечення є ключовим етапом в розробці, який визначає подальший успіх проекту. Архітектура визначає структуру, взаємодію та організацію програми, і в цьому контексті правильний вибір є важливим, оскільки від нього залежить надійність, ефективність і масштабованість системи. Еволюція архітектури програмного забезпечення є неперервним процесом, який відбувається відповідно до технологічного прогресу та змін у вимогах ринку. Нові архітектурні підходи і стилі регулярно з'являються, спрямовані на полегшення розробки, підтримки та оптимізації програмного забезпечення.

Однією з основних причин еволюції архітектури є постійний розвиток технологічного ландшафту. Впровадження нових технологій та підходів створює можливості для виникнення нових архітектурних концепцій. Також, потужності апаратного забезпечення постійно зростають, що дозволяє використовувати нові архітектурні рішення для оптимізації продуктивності та ефективності систем. З'явлення нових вимог та трендів на ринку також визначає еволюцію архітектури. Наприклад, зростання кількості користувачів, необхідність великої масштабованості, або вимоги до безпеки можуть впливати на вибір архітектурних стилів, спрямованих на вирішення конкретних завдань. Також, важливо враховувати досвід розробників та команди. Із набуттям нових знань та практики, розробники можуть відкривати нові підходи до створення архітектури, що впливає на загальний шлях розвитку галузі. Отже, еволюція архітектури програмного забезпечення визначається поєднанням технологічних інновацій, зростаючих вимог галузі та накопиченням досвіду розробників. Цей постійний процес дозволяє створювати більш ефективні, масштабовані та надійні програмні рішення.

Проблема 1. Невизначеність бізнес-вимог

Одним з головних викликів при виборі архітектури є невизначеність щодо майбутніх бізнес-вимог. На початковому етапі розробки програмного продукту невизначеність бізнес-вимог є поширеним явищем, що може мати негативний вплив на процес вибору архітектури системи. Перш за все, невизначені або змінювані вимоги ускладнюють завдання визначення конкретних потреб системи. Це може виникнути з різних причин, таких як неповне розуміння власниками продукту чи непередбачені зміни на ринку.

Однією з основних проблем є відсутність стабільних вимог. Без чітко визначених та стійких вимог стає важко обрати архітектуру, оскільки остання повинна відповідати функціональним та нефункціональним вимогам системи. Постійні зміни в бізнес-вимогах можуть призвести до того, що обрана архітектура не буде оптимально відповідати остаточним вимогам продукту.

Другою причиною є складніше прогнозування майбутніх потреб системи. Невизначеність у вимогах робить важким передбачення, як розвиватиметься програмний продукт. Це особливо важливо при виборі архітектури, оскільки слід розглядати не лише поточні потреби, але й можливі зміни в майбутньому. Невизначеність в цьому контексті може призвести до обрання архітектурного рішення, яке буде неадекватним для майбутніх вимог.

У контексті вибору архітектури, невизначеність бізнес-вимог ускладнює завдання оцінки і вибору оптимального архітектурного підходу, оскільки неможливо чітко визначити, які вимоги будуть найбільш важливими та стійкими в майбутньому. Відсутність стабільних фундаментальних вихідних точок може призвести до того, що обрана архітектура буде менш адаптованою до реальних потреб продукту з плином часу.

Проблема 2. Конфлікт пріоритетів

Наступним критичним викликом вибору архітектури системи є конфлікт пріоритетів. Конфлікт пріоритетів виникає, коли розробники стикаються з різними аспектами, які вимагають пріоритизації в процесі вибору архітектури для програмного продукту. Це може бути спричинено різницею в думках між

різними стейкхолдерами чи навіть внутрішнім розбіженням у команді розробників. Конфлікт пріоритетів може негативно вплинути на можливість вибору оптимальної архітектури з декількох причин.

По-перше, різні пріоритети можуть виникати внаслідок розбіжностей в цілях інтересів стейкхолдерів. Наприклад, менеджери можуть більше цінувати швидкість розробки та вчасне випуск нового функціоналу, тоді як команда забезпечення якості може підкреслювати необхідність надійності та безпеки системи. Це створює конфлікт між потребами різних стейкхолдерів, що важко врахувати при виборі архітектури, оскільки різні архітектурні рішення можуть мають відмінні впливи на цільові характеристики.

По-друге, конфлікт пріоритетів між різними якостями програмного продукту також може бути причиною невизначеності. Наприклад, бажання досягти високої продуктивності та ефективності може суперечити меті створення легкого підтримання коду. Вибір архітектури, що оптимізована для одного аспекту, може призвести до компромісу іншого, що може вплинути на якість та довготривалу підтримку системи. Враховуючи конфлікт пріоритетів, розробники можуть стикатися з викликами прийняття рішень. Необхідно балансувати вимоги різних стейкхолдерів і знаходити компромісні рішення, що враховують різні аспекти якості та функціональності системи. Такий конфлікт може зробити вибір оптимальної архітектури викликом, оскільки ідеальне рішення для одного аспекту може не враховувати інші важливі характеристики системи.

Проблема 3. Широкий вибір технологій та інструментів

Додатковим викликом при виборі архітектури системи є широкий вибір технологій та інструментів. Різноманітність технологічного ландшафту може зробити вибір оптимальних інструментів складним завданням, а нові технології можуть швидко змінюватися, впливаючи на життєвий цикл архітектури. З одного боку, це дає розробникам можливість вибирати інструменти, які найкращим чином відповідають їхнім потребам та навичкам. З іншого боку, цей

широкий спектр може стати джерелом складнощів та викликів при розгляді архітектурних рішень.

Однією з проблем широкого вибору є неоднозначність вибору оптимального інструменту чи технології. Велика кількість доступних інструментів може виглядати обіцяюче, але при цьому може виникнути питання: які саме інструменти і технології будуть оптимальними для конкретного проекту? Розробникам слід враховувати не лише їхні функціональні можливості, але й сумісність із загальною архітектурою системи, масштабованість, швидкодію та інші фактори.

Ще однією трудністю є швидкий розвиток технологій. Нові інструменти і технології виходять на ринок і швидко змінюються. Це може впливати на довгострокове утримання та підтримку програмного продукту, оскільки інструменти, які сьогодні є популярними, можуть застаріти завтра. Додатково, гетерогенність технічного стеку може призвести до труднощів у спілкуванні та інтеграції різних компонентів системи. Різні технології можуть вимагати різних підходів до архітектури, що може зробити систему менш однорідною та важкою у підтримці.

Отже, широкий вибір інструментів і технологій вимагає розсудливого та уважного вибору, оскільки невміле використання може призвести до неузгодженості, складнощів у розвитку та утриманні системи, а також ускладнити вибір архітектури, яка враховує всі ці аспекти.

1.3. Постановка задачі дослідження

Архітектурні стилі визначають ключові принципи та підходи до структурування програмного забезпечення, що надає важливий фундамент для розробки систем. Їх дослідження має критичне значення з кількох причин. По-перше, архітектурні стилі надають загальні принципи організації коду, що спрощує процес проектування. Це дозволяє розробникам розуміти, як система буде виглядати на високому рівні та які принципи будуть використовуватися для забезпечення її стабільності та розширюваності. Другий аспект полягає в

тому, що архітектурні стилі надають готові шаблони та рішення для типових проблем. Дослідження цих стилів дозволяє розробникам використовувати вже перевірені підходи до реалізації певних архітектурних аспектів, зменшуючи ймовірність помилок та прискорюючи розробку. Третє, архітектурні стилі грають ключову роль у масштабованості та підтримці системи. Системи, побудовані на основі правильно обраних архітектурних стилів, зазвичай є більш гнучкими та легко змінюваними з часом. Це дозволяє адаптувати систему до нових вимог та технологічних трендів. Нарешті, важливість дослідження архітектурних стилів виявляється в їхньому внеску в розуміння принципів ефективного дизайну програмного забезпечення. Вони надають важливі вказівки для розробників, допомагаючи їм створювати системи, які не лише виконують свої завдання, але й є ефективними, легко зрозумілими та підтримуваними.

Через проблему вибору архітектурного стилю для програмних систем, їх різноманітність, не чітке розуміння варіантів використання конкретних стилів є гостра необхідність для їх детального вивчення, їх топологій та варіантів використання. Метою цієї роботи є таке дослідження, що дасть змогу зрозуміти доречність використання того чи іншого архітектурного стилю згідно до функціональних (бізнес вимог) та нефункціональних вимог системи. Результат дослідження ставить на меті створення лістингу оцінок архітектурних стилів за їх характеристиками.

Для успішного дослідження архітектурних стилів необхідно почати з вивчення літератури та класичних робіт, таких як "Design Patterns" або "A Pattern Language". Це дозволить освоїти основні принципи та здобути контекст для подальшого дослідження. Після ознайомлення з основами слід звернутися до академічних досліджень та статей, які конкретизують та розширюють знання про певні архітектурні стилі. Це допоможе у глибшому розумінні та оволодінні сучасними підходами. Додатково, вивчення архітектурних характеристик є важливим аспектом. Розгляньте такі аспекти, як продуктивність,

масштабованість, зручність у використанні, та інші, щоб визначити, як певний стиль впливає на систему в цілому.

В області архітектури програмного забезпечення існує значна кількість впливових наукових праць, які визначають основи та розвиток цієї галузі та є обов'язковими для вивчення. Ці праці відіграють ключову роль у формуванні сучасного уявлення про архітектуру програмного забезпечення і служать основою для розробників та архітекторів у їхній професійній діяльності. Саме тому для досягнення мети цієї роботи є необхідним детально вивчити такі фундаментальні праці.

Висновки до першого розділу

Враховуючи виклики до розробки масштабних веб застосунків, важливо підходити до вибору архітектури як до постійного процесу, який може адаптуватися до змін у вимогах, технологіях та умовах розробки програмного забезпечення. Вибір архітектури програмного забезпечення є складним і відповідальним процесом, оскільки стикається з великою кількістю проблем та нюансів, що потребують уважного вивчення та розгляду.

Незважаючи на ці виклики, правильний вибір архітектури має потенціал робити розробку програмного продукту більш простою та ефективною. Якщо розглядати всі аспекти, враховувати поточні і майбутні вимоги, а також використовувати оптимальні інструменти та технології, можна створити застосунок, який буде не лише технічно правильним, але і більш придатним для майбутнього розвитку та підтримки. Такий підхід сприяє покращенню якості програмного продукту та спрощенню процесу розробки.

РОЗДІЛ 2. АНАЛІЗ ІСНУЮЧИХ АРХІТЕКТУРНИХ СТИЛІВ ТА ОЦІНКА ЇХ ХАРАКТЕРИСТИК

2.1. Поняття чистого коду як фундаменту для архітектури системи

Чистий код - це концепція, яка приділяє особливу увагу якості програмного забезпечення, зрозумілості та підтримуваності. Основна ідея полягає в тому, щоб написати програмний код так, щоб його було легко читати, розуміти та підтримувати.

Одним з ключових аспектів чистого коду є його читабельність. Код повинен мати зрозумілі та описові імена змінних, функцій та класів, що дозволяє розробникам швидко отримати уявлення про його призначення. Крім того, форматування коду також має бути консистентним, сприяючи однорідності і створюючи приємний зовнішній вигляд коду.

Простота - інший важливий аспект. Чистий код не повинен бути заплутаним або перенавантаженим зайвими деталями. Замість цього, він має бути елегантним та легким для розуміння. Простота сприяє не лише зручності читання, але і полегшує підтримку коду в майбутньому.

Ефективність в контексті чистого коду не означає лише швидкість виконання, але й раціональне використання ресурсів. Лише ефективний код може працювати швидко і надійно, забезпечуючи необхідну функціональність.

Масштабованість також є важливою характеристикою чистого коду. Коли ми говоримо про чистий код в контексті масштабованості, маємо на увазі не тільки його поточну зручність для розробників, але й його здатність легко адаптуватися до зростання об'єму роботи чи змін у вимогах системи з часом.

Чистий код сприяє масштабованості, оскільки він чітко структурований і добре організований. Збереження чітких інтерфейсів, правильне розміщення функцій та модулів, а також уникання надмірної складної логіки роблять код більш гнучким та підготовленим до масштабування. Коли система масштабується, розробники можуть вносити зміни без значних зусиль, оскільки чистий код забезпечує легкість розуміння внутрішньої логіки та легкість

виявлення місць для оптимізацій чи модифікацій. Без чистого коду може виникнути ситуація, коли велика кількість складного, непорозумілого коду ускладнює розширення системи та може призвести до каскаду проблем при зміні або додаванні нового функціоналу. Отже, чистий код і масштабованість взаємодіють, створюючи основу для стабільної та розширюваної системи, яка може легко адаптуватися до змін у вимогах та зростання в об'ємі роботи.

Інкапсуляція є ще однією важливою частиною чистого коду. Вона дозволяє обгортати функціональність та дані в окремі класи чи модулі, надаючи чіткі інтерфейси для взаємодії. Це полегшує утримання коду та сприяє його безпеці та стійкості до змін.

Нарешті, чистий код не повинен містити зайвого або мертвого коду, що не використовується для роботи програми. Він має бути лаконічним, концентруючись на вирішенні конкретних завдань та униканні надмірних деталей.

Паралельно до написання чистого коду є необхідним покриття такого коду юніт тестами. Юніт-тестування є методологією тестування програмного забезпечення, спрямованою на перевірку найменших можливих одиниць коду - "юнітів". Це може бути функція, метод, чи навіть окремий клас. Основна ідея полягає в тому, щоб перевірити, чи працюють окремі частини програми вірно та відповідно до специфікацій.

Юніт-тести дозволяють розробникам швидко перевіряти правильність функцій чи методів. Ця можливість є важливим процесом в розробці програмного забезпечення. Це означає, що розробники створюють автоматизовані тести для перевірки окремих частин коду (юнітів) на відповідність очікуваному поведінці.

Раннє виявлення проблем стає можливим завдяки тому, що юніт-тести запускаються автоматично під час розробки чи змін в коді. Якщо в коді є помилка, тест негайно сигналізує про це. Цей підхід дозволяє виявляти та виправляти помилки на етапі їхнього виникнення, коли вони ще не накопичилися та не стали проблемами в продукційному середовищі.

Зменшення вартості виправлення полягає в тому, що розробники витрачають менше часу на виявлення та усунення проблем, оскільки вони не потрапляють у робочу систему. Також це сприяє більш ефективному використанню часу, оскільки виправлення помилок на ранніх етапах є менш трудомістким порівняно з тим, якщо проблеми виявляються пізніше.

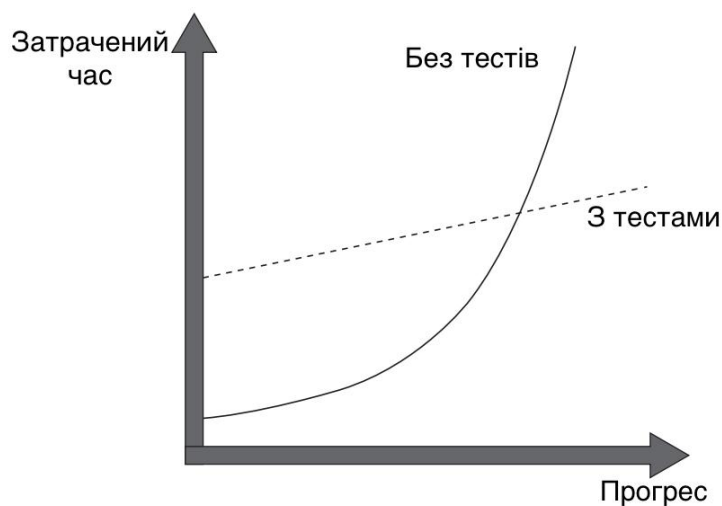


Рис. 2.1. Ефективність розвитку системи з юніт-тестуванням

Отже, виявлення помилок на ранніх етапах через юніт-тестування допомагає створювати більш стабільне, ефективне та надійне програмне забезпечення, зменшуючи ризик виникнення серйозних проблем у виробничому середовищі.

Юніт-тести гарантують, що окремі фрагменти коду працюють так, як очікується, що сприяє підтриманню високого рівня надійності та відповідності вимогам. При змінах у коді юніт-тести допомагають виявляти можливі негативні впливи на існуючий функціонал, забезпечуючи оперативне виявлення та виправлення проблем. Тести служать документацією для розробників, яка допомагає швидко з'ясувати, як повинен взаємодіяти кожен юніт коду, полегшуючи розуміння для розробників. Також юніт-тести дозволяють проводити рефакторинг коду з впевненістю, що зміни не порушують очікуваний функціонал, оскільки тести виявлять будь-які проблеми. Юніт-тести легко автоматизувати та інтегрувати в процес розробки, що сприяє швидкій та надійній поставці програмного продукту.

Чистий код, орієнтований на принципи читабельності, простоти та ефективності, створює фундамент для зрозумілого та підтримуваного програмного забезпечення. Такий код полегшує спільну роботу команди розробників, сприяє легкому виявленню та виправленню помилок, а також забезпечує гнучкість системи для подальших змін. Покриття тестами, у свою чергу, додає додатковий шар надійності та стабільності. Виявлення помилок на ранніх етапах розробки завдяки тестам дозволяє уникати їхнього перетворення на проблеми у виробничому середовищі, забезпечуючи безперебійну роботу системи. Спільна взаємодія чистого коду та покриття тестами є важливим етапом у побудові надійних, ефективних та розширюваних програмних продуктів. Цей фундамент є ключовим елементом для подальшого розвитку гарної архітектури системи, що визначає її успіх у довгостроковій перспективі.

2.2. Архітектурні стилі та їх топології

Архітектурні стилі представляють собою загальні концепції та підходи до структурування програмного забезпечення. Їх вибір визначає, як система буде організована, як компоненти будуть взаємодіяти та як буде забезпечена масштабованість та підтримка. Архітектурні стилі класифікуються за кількома основними типами, враховуючи їхні основні принципи та організаційні підходи. Найрозповсюдженіші типи стилів це монолітні та розподілені.

Монолітна архітектура представляє собою підхід до розробки програмного забезпечення, де весь функціонал системи концентрований в одному єдиному кодовому базисі та виконується в одному процесі. У цьому стилі весь програмний код, бізнес-логіка та інші компоненти об'єднані в єдиний цілісний блок, який дозволяє легше керувати та розвивати програму. В монолітній архітектурі система зазвичай поділена на логічні шари, такі як інтерфейс, бізнес-логіка та доступ до даних. Кожен шар має свою функціональність та взаємодіє з іншими шарами. Наприклад, шар інтерфейсу взаємодіє з користувачем, передаючи запити бізнес-логіці, яка в свою чергу взаємодіє з базою даних через шар доступу до даних.

Однією з переваг монолітної архітектури є простота. Розробка, тестування та розгортання можуть бути менш складними порівняно з іншими архітектурними стилями, такими як мікросервіси або розподілені системи. Також управління версіями та моніторинг можуть бути більш простими через консолідований кодовий базис. Однак у монолітних системах може виникнути проблема масштабованості та гнучкості. Розширення функціоналу або підтримка великої кількості користувачів може вимагати значних змін у великому кодовому базисі. Також, розвиток може стати складнішим і повільнішим з часом. Окрім того, монолітна архітектура може негативно впливати на можливість розділення обов'язків між командами розробників та швидкість впровадження нових функцій.

Розподілені архітектурні стилі представляють собою підхід до розробки програмного забезпечення, де функціонал системи розподіляється між різними компонентами. Ці компоненти можуть виконуватися на різних вузлах мережі та взаємодіяти через мережеві протоколи. У розподілених системах ключовим елементом є взаємодія між компонентами через мережу. Клієнт-серверна модель є однією з основних форм розподіленої архітектури, де клієнти відправляють запити серверам, які обробляють ці запити та повертають результати. Це дозволяє розділити відповідальності між клієнтською та серверною частинами системи. Ще однією формою розподіленої архітектури є мікросервісна модель, де система розділена на невеликі, незалежні сервіси. Кожен сервіс виконує конкретну функцію і може бути розгорнутий та масштабований незалежно від інших. Цей підхід дозволяє створювати гнучкі, масштабовані та легко розширювані системи. Втім, він також вносить виклики, такі як управління консистентністю даних між сервісами та ускладнення тестування та розгортання. Враховуючи особливості розподіленої архітектури, розробники повинні уважно планувати та враховувати принципи масштабованості, ефективності та безпеки для створення стабільних та продуктивних розподілених систем.

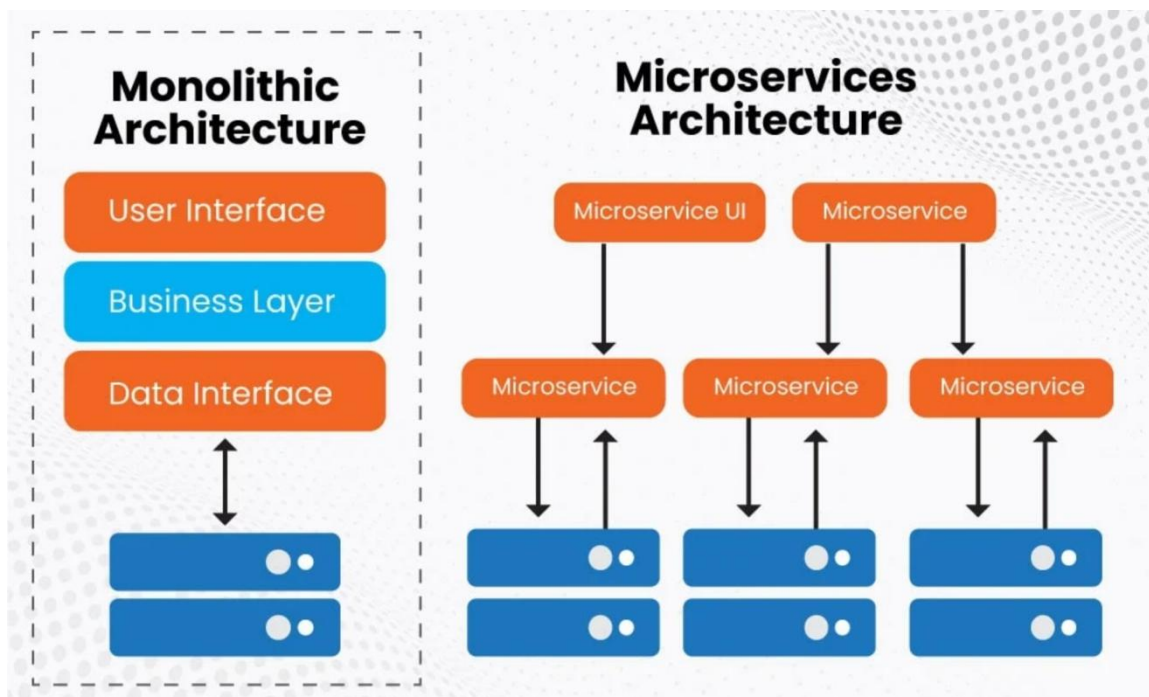


Рис 2.2. Монолітний та розподілений архітектурні стилі

Важливо зазначити, що завданням вивчення архітектурних стилів є визначення їх архітектурних характеристик та оцінка. Архітектурні характеристики представляють собою властивості та особливості архітектури програмного забезпечення, які визначають його структуру, організацію та взаємодію між компонентами. Ці характеристики визначають спосіб, яким система буде задовольняти функціональні та нефункціональні вимоги.

Детальне дослідження архітектурних стилів:

Архітектурний стиль "Layered Architecture" є популярним підходом, який розділяє систему на логічні шари чи рівні, де кожен рівень виконує певну функцію та взаємодіє тільки з певними сусідніми рівнями. Кожен шар представляє собою логічну групу функціонально пов'язаних компонентів, які мають однаковий рівень абстракції та завдання. Цей стиль є одним з базових архітектурних стилів у світі програмного забезпечення. Він визначає базову структуру системи, розподіляючи її функціональність між логічними шарами чи рівнями. Кожен шар відповідає за конкретний аспект функціоналу та взаємодіє тільки з певними сусідніми рівнями. Основна ідея цього базового стилю полягає в тому, щоб створити структуру, де кожен рівень відділений від інших та має чітко визначені відповідальності. Це полегшує утримання,

розширення та розробку системи. Layered Architecture забезпечує чітку схему взаємодії між компонентами системи, роблячи її більш стійкою до змін та розширення.

Однією з основних переваг цього базового стилю є його простота та зрозумілість. Це дозволяє команді розробників швидко орієнтуватися в структурі системи та ефективно співпрацювати. Завдяки розділенню відповідальностей між шарами, команди можуть працювати над конкретними частинами системи незалежно одна від одної.

Враховуючи його базовість та простоту, Layered Architecture залишається популярним та ефективним стилем для багатьох видів програмного забезпечення.

Давайте розглянемо основні аспекти та переваги Layered Architecture:

Структура шарів:

- Інтерфейсний (Presentation) шар: Цей рівень відповідає за взаємодію з користувачем та виведення інформації. Він містить компоненти, які відображають дані та обробляють введення від користувачів.

- Бізнес-логічний (Business Logic) шар: Цей рівень містить бізнес-логіку, відповідає за обробку бізнес-правил та додаткову обробку даних. Він незалежний від інтерфейсного та доступу до даних і виконує основний функціонал системи.

- Шар доступу до даних (Data Access) шар: В цьому рівні реалізована взаємодія з базою даних або іншими джерелами даних. Він відповідає за зберігання та отримання даних.

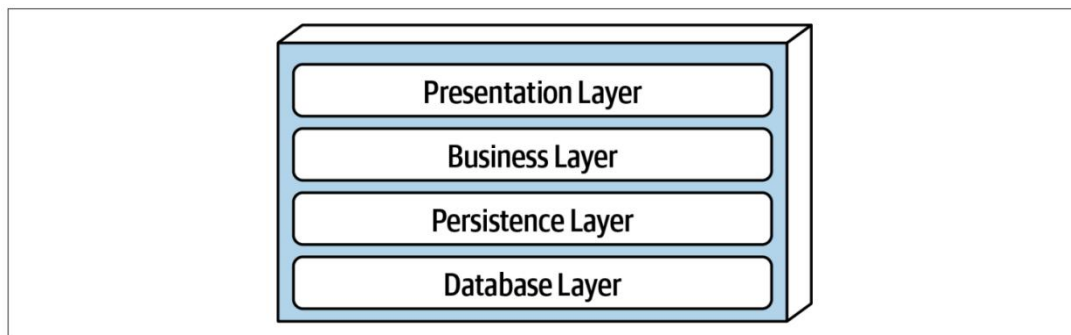


Рис. 2.3. Топологія Layered Architecture

2. Переваги Layered Architecture:

- Модульність та розширюваність: Кожен шар може бути розглядений як окремий модуль, що дозволяє простоту утримання та розширення системи.
- Схема взаємодії: Взаємодія між шарами відбувається визначеними інтерфейсами, що робить систему більш стійкою до змін та полегшує тестування окремих компонентів.
- Розділення відповідальностей: Кожен шар має конкретні відповідальності, що полегшує розуміння та розподіл завдань між командами розробників.

3. Недоліки Layered Architecture:

- Збільшення затримок взаємодії: Звертання до різних шарів може викликати певні затримки, особливо у великих системах.
- Обмежена гнучкість: Використання строгої структури шарів може призвести до обмежень у реалізації певних функціональностей.

Все враховуючи, Layered Architecture залишається популярним вибором для багатьох застосувань, де важлива модульність, зручність утримання та зрозуміла структура системи. Розглянемо деякі ситуації, де цей стиль може бути доречним та де, можливо, не найкращим вибором:

Доречне використання Layered Architecture:

1. Ентерпрайз-застосунки: У великих ентерпрайз-проектах, де модульність та легкість утримання є ключовими вимогами, використання Layered Architecture може бути доречним. Цей стиль дозволяє легко розподілити відповідальності між різними командами та забезпечити структуру для розвитку.

2. Системи з чітко визначеною бізнес-логікою: У випадках, коли бізнес-логіка системи легко поділяється на різні шари (наприклад, інтерфейс, обробка даних, доступ до даних), Layered Architecture може сприяти зрозумілості та простоті розробки.

3. Проекти з обмеженим бюджетом та строками: У випадках, коли важлива швидкість розробки та простота архітектурного рішення, Layered Architecture може бути швидким та ефективним вибором.

Недоречне використання Layered Architecture:

1. Системи з високою масштабованістю: У випадках, коли важлива горизонтальна масштабованість, де велика кількість користувачів чи великий об'єм даних, Layered Architecture може стати обмежуючим фактором. Інші архітектурні стилі, такі як мікросервісна архітектура, можуть бути більш адаптованими до таких вимог.

2. Системи із складним потоком даних: У випадках, коли потік даних у системі дуже складний та вимагає більш гнучкого управління, можуть бути ефективніші інші архітектурні підходи, наприклад, Event-Driven Architecture.

3. Модерні технології та тренди: У випадках, коли проект потребує використання сучасних технологій, таких як контейнеризація, мікросервіси та хмарні рішення, може бути вигідно розглядати інші архітектурні стилі, які легше адаптуються до цих технологічних тенденцій.

В кожному конкретному випадку розглядується специфіка проекту, вимоги та мета, і вибір архітектурного стилю визначається цими факторами для досягнення оптимального результату.

Архітектурний стиль "Microkernel Architecture" або "Мікроядро" представляє собою підхід до побудови систем, де базовий функціонал системи розділяється на невеликі компоненти, які виконують мінімальний набір обов'язків. Основною ідеєю мікроядра є мінімізація функцій ядра, виводячи більшість служб і додаткового функціоналу за межі ядра. Цей архітектурний стиль може бути розглянуто як розширення Layered Architecture через свою унікальну можливість модульності та розділення функцій між різними компонентами системи. В той час як Layered Architecture також передбачає розділення системи на шари, Microkernel підіймає цей підхід на новий рівень, дозволяючи кожному модулю реалізувати різні шари системи.

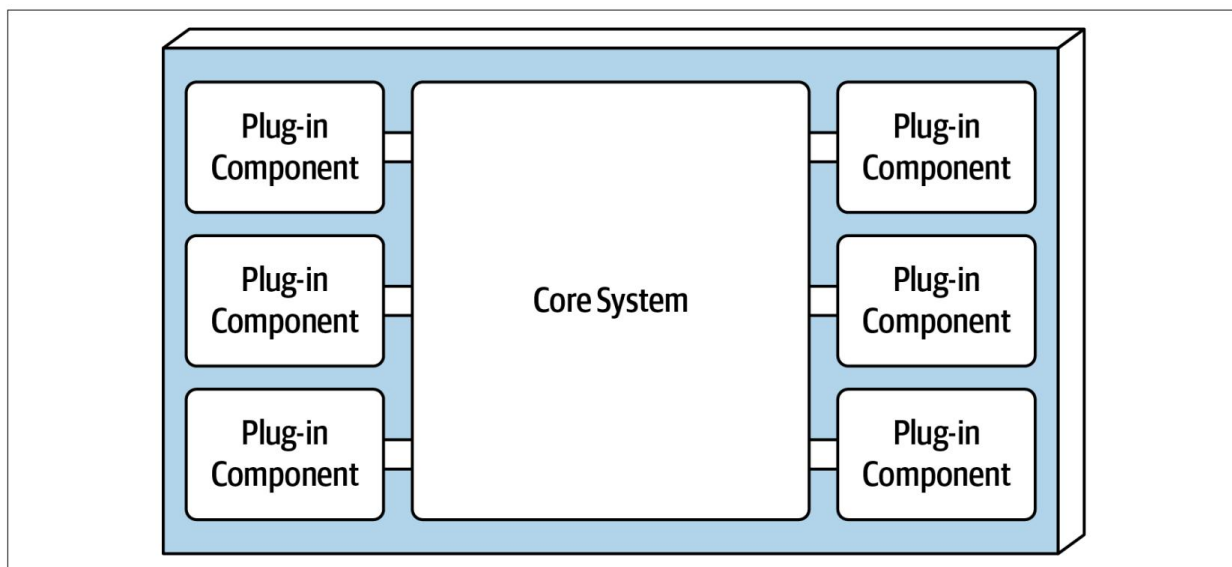


Рис. 2.4. Топологія Microkernel Architecture

Microkernel Architecture вказує на використання модульного та розділеного підходу до побудови системи. Давайте розглянемо це з погляду бекенд-застосунків:

1. Мінімізація ядра:

- В бекенд-застосунках Microkernel дозволяє мінімізувати ядро, обмежуючи його базовими операціями, такими як обробка запитів, керування ресурсами, та інші фундаментальні функції.

- Весь додатковий функціонал, такий як обробка бізнес-логіки, взаємодія з базою даних, може бути виокремлений в окремі модулі.

2. Модульність та Розділені Компоненти:

- Кожен компонент бекенд-застосунку може виконувати свою функцію відокремлено від інших. Наприклад, окремий модуль може відповідати за автентифікацію, інший - за авторизацію, третій - за обробку та зберігання даних.

- Це дозволяє підтримувати велику кількість невеликих та самостійних компонентів, які легко модифікувати та розвивати.

3. Гнучкість та Легкість Оновлення:

- Бекенд, побудований на основі Microkernel Architecture, дозволяє замінювати та оновлювати окремі компоненти без перезапуску всієї системи.

- Це полегшує розгортання нових функцій або виправлень без великих втрат часу та ресурсів.

4. Розділена Розробка:

- Розробка різних компонентів може проводитися паралельно та незалежно один від одного.
- Це дозволяє різним командам або розробникам працювати над різними частинами системи, що підвищує продуктивність та швидкість розробки.

5. Надійність та Масштабованість:

- Розділення компонентів дозволяє легше керувати та масштабувати окремі частини системи відповідно до потреб бізнесу.
- Невдачі або проблеми в одному компоненті не обов'язково впливають на решту системи.

Microkernel Architecture у бекенд-застосунках відкриває можливості для гнучкого та швидкого розвитку систем, особливо в умовах постійної зміни вимог. Вона полегшує утримання та розширення системи, забезпечуючи високу рівень модульності та гнучкість.

Висновок щодо доречності використання Microkernel Architecture залежить від конкретних вимог та характеристик проекту. Однак, загалом можна сказати, що Microkernel Architecture є відмінним вибором в ряді сценаріїв:

1. Гнучкість та Розділення Відповідальностей:

- Якщо система має велику кількість компонентів з різними відповідальностями, Microkernel Architecture дозволяє ефективно розділити ці компоненти, забезпечуючи гнучкість та можливість швидкої адаптації до змін.

2. Масштабованість:

- Для систем, де масштабованість є ключовою вимогою, Microkernel Architecture дозволяє незалежно масштабувати окремі компоненти, що дозволяє ефективно використання ресурсів.

3. Підтримка Розподіленості:

- Microkernel може бути використаний для створення розподіленої системи, де різні модулі можуть виконуватися на різних вузлах мережі, що полегшує розширення та підтримку великих систем.

4. Легкість Оновлень:

- Для проектів, де важливо швидко вносити зміни та оновлення в систему, Microkernel Architecture дозволяє оновлювати або замінювати окремі компоненти без перезапуску всієї системи.

5. Висока Надійність:

- Використання Microkernel Architecture може сприяти високій надійності, оскільки кожен компонент може бути розглянутий як окремий, самостійний модуль.

Однак важливо враховувати, що використання Microkernel Architecture також може вести до деяких недоліків, таких як витрати на комунікацію між компонентами та додатковий рівень складності. Тому вибір даного архітектурного стилю повинен ґрунтуватися на конкретних вимогах та особливостях проекту.

Microservices Architecture є структурним підходом до розробки програмних систем, що визначається розбиттям застосунку на невеликі, незалежні та функціонально визначені компоненти - мікросервіси. Кожен мікросервіс виконує конкретну бізнес-функцію і може бути розроблений, розгортаний та масштабований незалежно від інших сервісів. Взаємодія між мікросервісами здійснюється за допомогою стандартизованих інтерфейсів, таких як API.

Основні характеристики мікросервісної архітектури включають:

- Незалежність: Кожен мікросервіс фокусується на конкретній функціональності, що робить його незалежним від інших сервісів.
- Незалежність: Кожен мікросервіс фокусується на конкретній функціональності, що робить його незалежним від інших сервісів.
- Самостійність: Мікросервіси можуть бути розроблені, розгортані та оновлювані незалежно один від одного.

- Розділення Бізнес-Функцій: Система розбивається на мікросервіси відповідно до їхньої функціональності для легкості розуміння та підтримки.
- Стандартизовані Інтерфейси: Взаємодія здійснюється через стандартизовані API, що забезпечує взаємодію без залежностей від конкретних технологій.
- Незалежне Розгортання та Оновлення: Можливість незалежного розгортання та оновлення мікросервісів сприяє швидкому впровадженню нових функцій.
- Легкість Вдосконалення та Заміни: Зміни в одному мікросервісі не впливають на інші, сприяючи легкості вдосконалення та заміни компонентів системи.
- Масштабованість: Можливість масштабування окремих мікросервісів дозволяє ефективно використовувати ресурси системи.

Мікросервісна архітектура покликана полегшити розробку, вдосконалення та утримання програмних систем за умов постійних змін та вимог ринку.

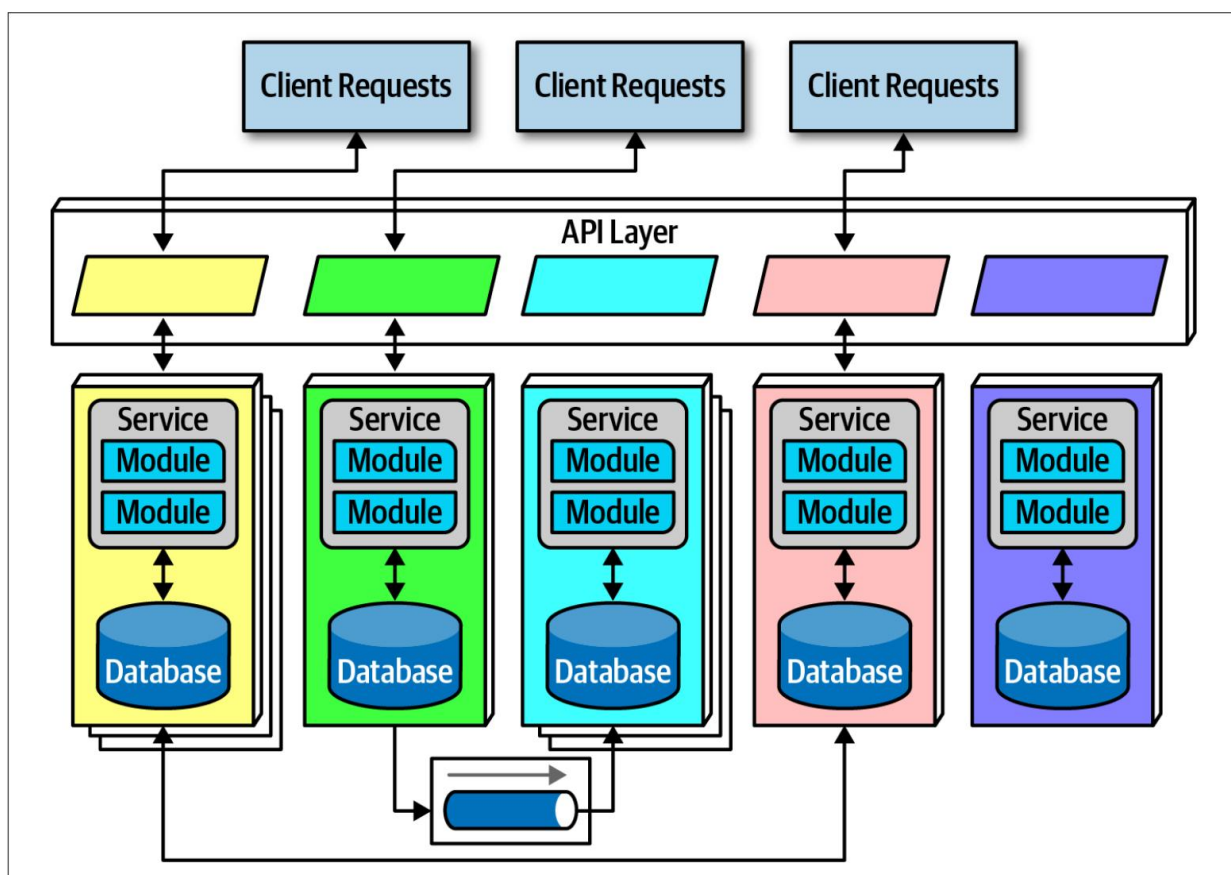


Рис. 2.5. Топологія Microservices Architecture

Вибір між використанням та невикористанням Microservices Architecture повинен базуватися на конкретних потребах та вимогах проекту. Ось деякі випадки, коли Microservices може бути доречно використовувати, і випадки, коли це може бути менш обґрунтовано:

Доречно використовувати Microservices:

- Складні Бізнес-Вимоги: Коли проект має складну функціональність та вимоги, розбиття системи на невеликі та незалежні мікросервіси полегшить розробку та управління.
- Гнучкість та Швидкість Розробки: Коли швидкість внесення змін та гнучкість розробки є ключовими факторами, Microservices дозволяє ефективно вдосконалювати окремі компоненти без впливу на інші.
- Масштабованість: Коли система повинна масштабуватися горизонтально, Microservices Architecture дозволяє незалежне масштабування окремих сервісів.
- Незалежні Відділи Розробки: Коли розробка та управління різними частинами системи відбувається незалежно, Microservices дозволяє різним відділам працювати безперешкодно.

Менш Обґрунтовано використовувати Microservices:

- Прості Бізнес-Вимоги: Для простих проектів або тих, що не мають складних вимог, використання Microservices може бути зайвим та збільшити складність розробки.
- Невеликі Масштаби: У випадках, коли не передбачається значне зростання обсягів даних або навантаження, вартість впровадження та утримання Microservices може бути необґрунтованою.
- Тісно Зв'язані Функціональності: Якщо функціональності системи сильно тісно пов'язані, розділення їх на окремі мікросервіси може взаємодіяти з продуктивністю та зростанням складності.
- Обмежені Ресурси: Якщо проект має обмежені ресурси та бюджет, впровадження та утримання Microservices може бути витратним.

Загалом, вибір між використанням та невикористанням Microservices Architecture повинен ґрунтуватися на глибокому розумінні бізнес-вимог та характеристик проекту, а також на здатності команди ефективно впроваджувати та управляти такою архітектурою.

Event-Driven Architecture є архітектурним підходом, при якому компоненти системи взаємодіють, відправляючи та слухаючи події (events). Цей підхід взятий на озброєння для побудови розподілених систем, і в контексті використання message broker, події відправляються та отримуються через централізований посередник - брокер подій.

Для реалізації архітектурного стилю Event-Driven Architecture (EDA) розробники можуть використовувати різні message brokers. Ось деякі популярні message brokers, які використовують для побудови систем EDA:

- Apache Kafka
- RabbitMQ
- Google Cloud Pub/Sub

Основні компоненти Event-Driven Architecture в розподіленій системі:

- Подія (Event): Подія визначає значущий стан або зміну, про яку система має інформувати інші частини. Це може бути будь-яка подія, яка має важливе значення для системи.
- Відправник Подій (Event Producer): Компонент, який відповідає за генерацію та відправлення подій у систему. Він виступає як джерело інформації та генерує події при виникненні певних умов.
- Брокер Подій (Event Broker): Централізований сервіс, який відповідає за прийом, розподіл та маршрутизацію подій у системі. Брокер подій служить посередником для відправників та отримувачів подій.
- Підписник Подій (Event Consumer): Компонент, який підписується на отримання конкретних подій від брокера. Підписники реагують на події та виконують відповідні дії в системі.

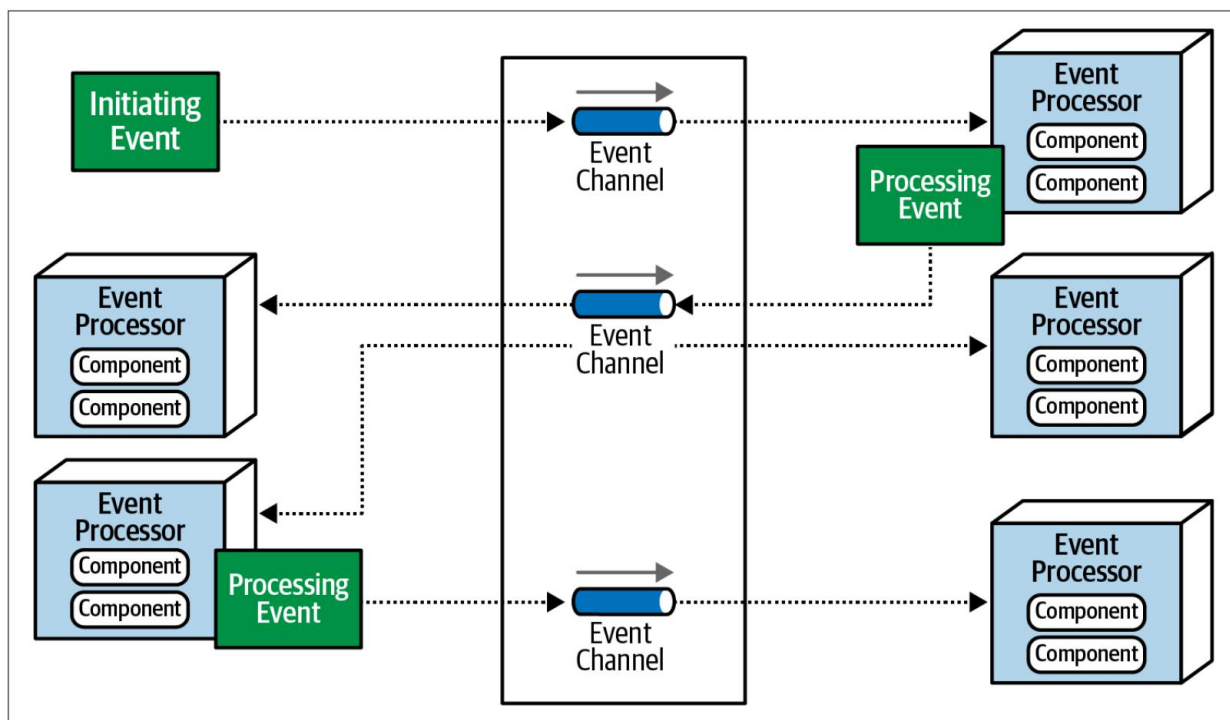


Рис. 2.6. Топологія Event Driven Architecture

Принципи та Характеристики Event-Driven Architecture:

Легкість Розширення: Додається новий функціонал, створюючи нові відправники та підписники подій, що не впливає на існуючі компоненти системи.

Еластичність: Система стає більш еластичною, оскільки компоненти можуть функціонувати незалежно, а невдача в одній частині системи не впливає на інші.

Легка Інтеграція: Дозволяє легко інтегрувати нові компоненти, оскільки вони можуть взаємодіяти через стандартизований інтерфейс подій.

Підтримка Розподіленості: Ідеально підходить для розподілених систем, оскільки дозволяє взаємодіяти з компонентами, які можуть знаходитися на різних вузлах мережі.

Легка Масштабованість: Систему можна легко масштабувати, додаючи або видаляючи відправники та підписники подій.

Використання Event-Driven Architecture в розподіленій системі через message broker робить систему більш гнучкою, легко розширюваною та адаптованою до змін у середовищі.

Похідні архітектурні стилі

Вищезгадані архітектурні стилі є найросповсюдженішими та, в якомусь роді, базовими. Еволюційно ці стилі є фундаментальним рішенням для найросповсюдженіших проектних проблем. Проте, якщо дизайн проекту потребує специфічного рішення, необхідно використовувати вузьконаправлені архітектурні стилі. Короткий лістинг таких проблемно-орієнтованих стилей нижче:

Pipeline Architecture - це стиль, в якому дані або задачі обробляються послідовно у вигляді конвеєра чи трубопроводу. Кожен етап конвеєра відповідає за виконання конкретної функції або обробку певної частини даних.

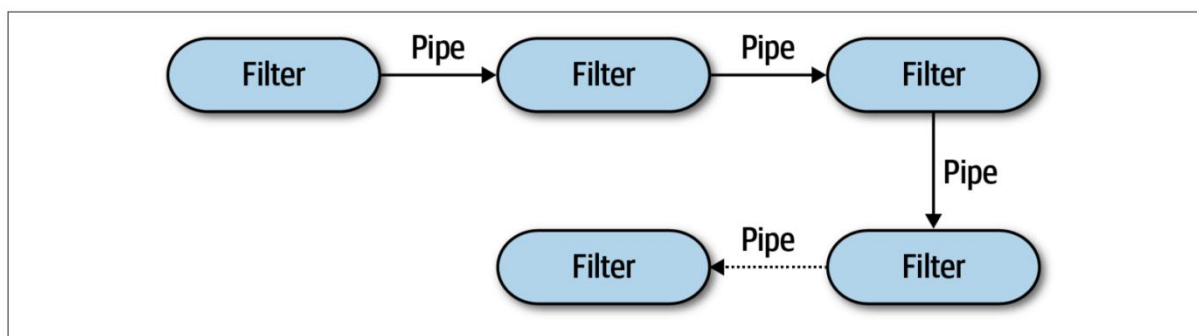


Рис. 2.7. Топологія Pipeline Architecture

Плюси:

- Простота: Простий та легко зрозумілий дизайн.
- Можливість Оптимізації: Кожен етап може бути оптимізований незалежно.

Мінуси:

- Обмежена Гнучкість: Структура конвеєра може бути обмеженою та важко змінюватися.
- Затримка: Завдяки послідовній обробці, може виникнути затримка у виконанні завдань.

Space-Based Architecture - це стиль, в якому система розбивається на простори (spaces), що можуть бути розподілені та масштабовані. Кожен простір обслуговує певний функціонал та може взаємодіяти з іншими просторами. Цей

стиль спеціалізується на оптимізації бази даних, переносячи її в середу оперативної пам'яті.

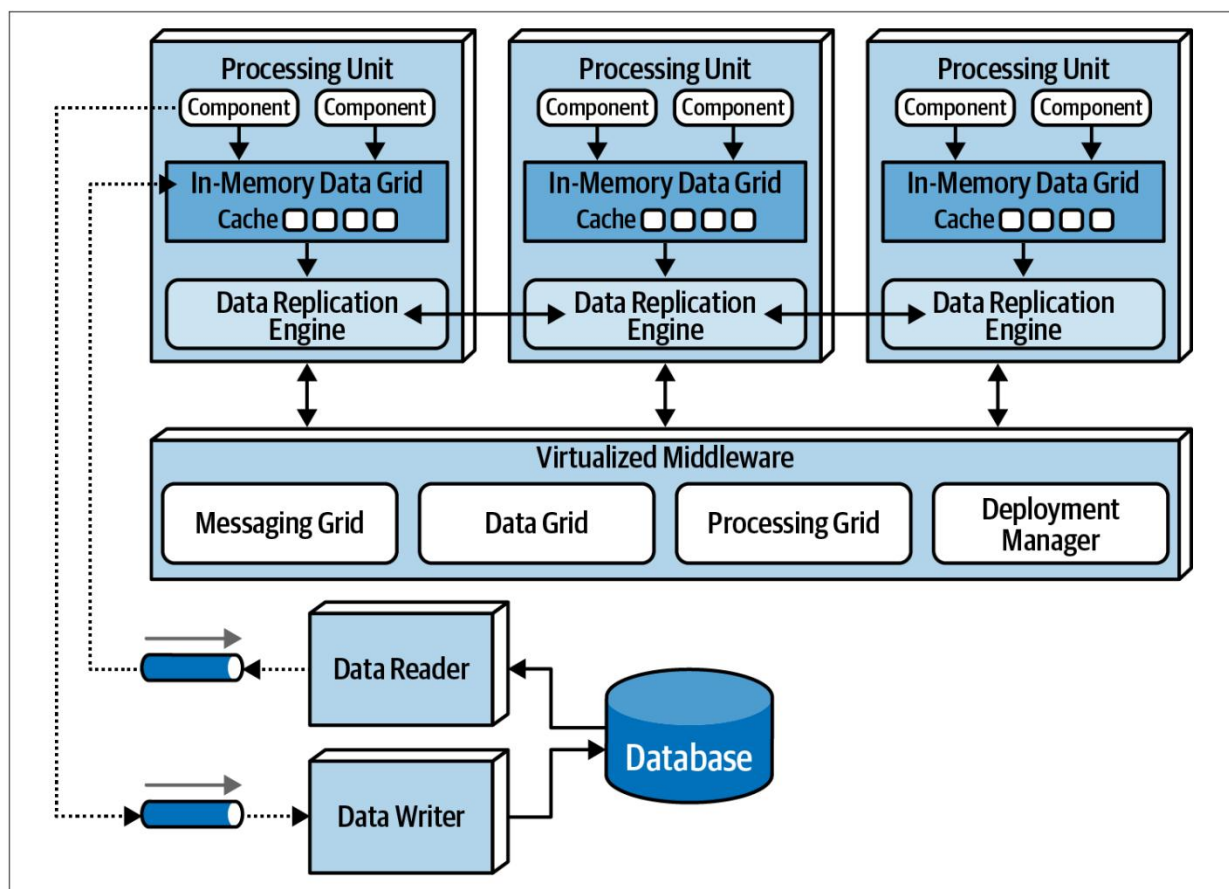


Рис. 2.8. Топологія Space-Based Architecture

Плюси:

- Масштабованість: Простір може бути легко масштабованим горизонтально.
- Висока Надійність: Завдяки розподіленому характеру системи, можлива висока надійність та доступність.

Мінуси:

- Складність Проектування* Вимагає детального проектування та розуміння взаємодії просторів.
- Вартість: Впровадження та утримання може бути дорогим.

Orchestration-Based Architecture - це стиль, де основна логіка системи управляється централізованим контролером (оркестратором), який координує взаємодію між різними компонентами. Цей стиль був популярний для поєднання декількох інформаційних систем в одну.

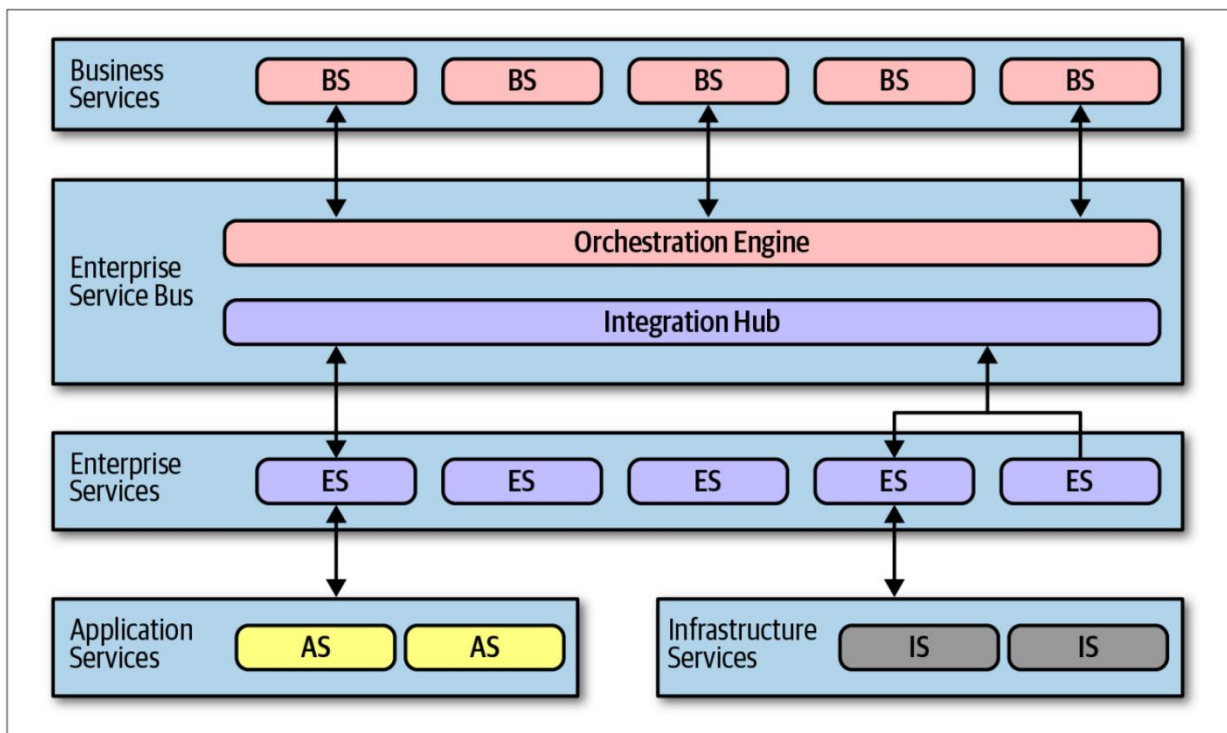


Рис. 2.9. Топологія Orchestrator-Based Architecture

Плюси:

- **Централізований Контроль:** Забезпечує централізований та контрольований підхід до управління системою.
- **Легка Інтеграція:** Зручний для інтеграції різних компонентів.

Мінуси:

- **Складність Розширення:** Може бути складним для розширення та модифікації.
- **Однопоточність:** Однопоточність може призвести до затримок у виконанні завдань.

Кожен із зазначених архітектурних стилів має свої плюси та мінуси, і вибір конкретного стилю повинен враховувати вимоги та характеристики конкретного проекту.

2.3. Оцінка архітектурних характеристик

Архітектурні характеристики в контексті програмної архітектури є важливими властивостями, що описують ключові аспекти системи. Вони визначають архітектурний образ системи та взаємозв'язки між її складовими

частинами. Ці характеристики служать основним фреймворком для аналізу, проектування та оцінки архітектури програмного забезпечення.

Основні архітектурні характеристики включають: [4]

1. Deployability (Можливість розгортання): Дана характеристика визначає легкість та швидкість розгортання системи в новому середовищі чи інфраструктурі. Вона враховує автоматизацію процесів розгортання та забезпечує мінімальні зусилля для впровадження змін.

2. Elasticity (Еластичність): Еластичність описує здатність системи динамічно змінювати свою потужність в залежності від зміни навантаження чи вимог. Ця характеристика підтримує масштабування ресурсів в реальному часі.

3. Evolutionary (Еволюційність): Ця характеристика вказує на легкість та ефективність системи у внесенні змін та розвитку з часом. Еволюційність передбачає гнучкість у впровадженні нового функціоналу та адаптації до змін в вимогах.

4. Fault Tolerance (Стійкість до відмов): Стійкість до відмов визначає, наскільки система може ефективно працювати в умовах виникнення помилок чи відмов. Це включає механізми виявлення, відновлення та запобігання помилкам.

5. Modularity (Модульність): Модульність вказує на розділення системи на невеликі та незалежні модулі. Це полегшує розробку, тестування та модифікацію коду, сприяючи гнучкості та легкості управління системою.

6. Overall Cost (Загальна вартість): Загальна вартість включає усі етапи життєвого циклу системи та пов'язані з ними витрати. Ця характеристика оцінює ефективність та економічність проекту в цілому.

7. Performance (Продуктивність): Продуктивність визначає швидкодію та ефективність системи у виконанні своїх завдань, включаючи обробку даних та відповідь на запити.

8. Reliability (Надійність): Надійність вказує на стабільність та безперебійність роботи системи, враховуючи можливість виникнення відмов чи помилок.

9. Scalability (Масштабованість): Масштабованість визначає, наскільки легко система може збільшувати свою потужність для врахування зростаючого обсягу даних чи виконання завдань.

10. Simplicity (Простота): Простота визначає зрозумілість та лаконічність структури та реалізації системи, що полегшує розуміння та управління нею.

11. Testability (Тестованість): Тестованість вказує на зручність та ефективність проведення тестування системи для перевірки її функціональності та відповідності вимогам.

Ці архітектурні характеристики допомагають інженерам програмного забезпечення приймати обгрунтовані рішення щодо проектування та вдосконалення системи, враховуючи різні аспекти її функціонування та вимог користувачів.

Таблиця 2.1

Оцінка архітектурних характеристик Layered Architecture стилю

Архітектурна Характеристика	Оцінка
Можливість розгортання	☀
Еластичність	☀
Еволюційність	☀
Стійкість до відмов	☀
Модульність	☀ ☀
Загальна вартість	☀ ☀ ☀ ☀ ☀
Продуктивність	☀ ☀
Надійність	☀ ☀ ☀
Масштабованість	☀
Простота	☀ ☀ ☀ ☀ ☀
Тестованість	☀ ☀ ☀

Загальна вартість та простота є основними перевагами стилю архітектури зі шаровим підходом. Будучи монолітичними за своєю природою, шарові

архітектури не мають складностей, що властиві розподіленим стилям архітектури, вони прості та легкі для розуміння, а також є відносно низькими за вартістю для створення та утримання. Однак, слід зауважити, що ці рейтинги швидко зменшуються, коли монолітичні шарові архітектури стають більшими та, відповідно, складнішими.

Якщо йдеться про можливість розгортання та тестування, цей стиль архітектури отримує дуже низькі оцінки. Знижена оцінка для розгортання обумовлена складністю розгортання, великим ризиком та відсутністю можливості частих розгортань. Проста зміна трьох рядків у файлі класу у стилі архітектури зі шарами вимагає повного перерозгортання всього блоку, включаючи можливі зміни в базі даних, зміни конфігурації або інші зміни коду, що потрапляють поруч із початковою зміною. Більше того, ця проста зміна зазвичай впливає на десятки інших місць, що ще більше збільшує ризик розгортання (а також частоту розгортання). Оцінка за тестованість також відображає цей сценарій: при простій зміні трьох рядків більшість розробників не буде витрачати години на виконання повного набору тестів на регресію.

Надійність отримує середню оцінку (три зірки) для цього стилю архітектури, головним чином через відсутність мережевого трафіку, проблем пропускну здатності та затримок, які характерні для більшості розподілених архітектур. Ми надали шаровій архітектурі три зірки за надійність через монолітне розгортання.

Еластичність та масштабованість заслуговують на низькі оцінки для шарової архітектури, переважно через монолітичні розгортання та відсутність архітектурної модульності.

Щодо продуктивності, це завжди цікава характеристика для оцінки для шарової архітектури. Вона заслуговує лише дві зірки через те, що цей стиль архітектури просто не сприяє високопродуктивним системам через відсутність паралельної обробки та закритості шарів.. Як і з масштабованістю, продуктивність може бути вирішена через кешування, багатопотоковість тощо, але це не є природною характеристикою цього стилю архітектури.

Шарові архітектури не підтримують відмовостійкість через монолітичні розгортання та відсутність архітектурної модульності. Якщо навіть одна невелика частина шарової архітектури спричинює виникнення умови вичерпання пам'яті, це вплине на всю одиницю застосунку та спричинить його збій.

Таблиця 2.2

Оцінка архітектурних характеристик Microkernel Architecture стилю

Архітектурна Характеристика	Оцінка
Можливість розгортання	☀ ☀ ☀
Еластичність	☀
Еволюційність	☀ ☀ ☀
Стійкість до відмов	☀
Модульність	☀ ☀ ☀
Загальна вартість	☀ ☀ ☀ ☀ ☀
Продуктивність	☀ ☀ ☀
Надійність	☀ ☀ ☀
Масштабованість	☀
Простота	☀ ☀ ☀ ☀
Тестованість	☀ ☀ ☀

Подібно до стилю архітектури зі шарами, простота та загальна вартість є основними перевагами стилю мікроядра, а основними його слабкостями є масштабованість, відмовостійкість та розширюваність. Ці слабкості виникають через типові монолітичні розгортання, характерні для стилю архітектури мікроядра. Тут закінчуються подібності цих стилів.

Стиль архітектури мікроядра є унікальним тим, що він може бути як областєво розподіленим, так і технічно розподіленим. Тестовість, розгортаність та надійність отримали трошки вищі середні оцінки (три зірки), головним чином тому, що функціональність може бути ізольована в незалежні

підключені компоненти. Якщо це зроблено правильно, це скорочує загальний обсяг тестування змін та також зменшує загальний ризик розгортання, особливо якщо компоненти підключаються в режимі виконання.

Модульність та розширюваність також отримали трошки вищі середні оцінки (три зірки). За допомогою стилю архітектури мікроядра, можливо додавати, видаляти та змінювати додаткові функціональності через незалежні, автономні підключені компоненти, зробивши це відносно легким для розширення та вдосконалення додатків, створених з використанням цього стилю архітектури, і дозволяючи командам швидше реагувати на зміни.

Таблиця 2.3

Оцінка архітектурних характеристик Microservices Architecture стилю

Архітектурна Характеристика	Оцінка
Можливість розгортання	☀ ☀ ☀ ☀
Еластичність	☀ ☀ ☀ ☀ ☀
Еволюційність	☀ ☀ ☀ ☀ ☀
Стійкість до відмов	☀ ☀ ☀ ☀
Модульність	☀ ☀ ☀ ☀ ☀
Загальна вартість	☀
Продуктивність	☀ ☀
Надійність	☀ ☀ ☀ ☀
Масштабованість	☀ ☀ ☀ ☀ ☀
Простота	☀
Тестованість	☀ ☀ ☀ ☀

Визначні риси цієї архітектури - масштабованість, еластичність та еволюційність. Деякі з найбільш масштабованих систем, які коли-небудь були створені, використовували цей стиль з великим успіхом. Аналогічно, через те, що архітектура сильно покладається на автоматизацію та інтелектуальну інтеграцію з операціями, розробники також можуть вбудовувати підтримку

еластичності в архітектуру. Продуктивність часто є питанням в мікросервісах - розподілені архітектури повинні здійснювати багато мережових викликів для завершення роботи, що має велике продуктивне навантаження. Крім того, для перевірки ідентифікації та доступу для кожного ендпоінта вони повинні викликати перевірки безпеки. У мікросервісному світі існують численні шаблони для покращення продуктивності, включаючи інтелектуальне кешування даних та реплікацію для запобігання зайвих мережових викликів.

Таблиця 2.4

Оцінка архітектурних характеристик Event-Driven Architecture стилю

Архітектурна Характеристика	Оцінка
Можливість розгортання	☀ ☀ ☀
Еластичність	☀ ☀ ☀
Еволюційність	☀ ☀ ☀ ☀ ☀
Стійкість до відмов	☀ ☀ ☀ ☀ ☀
Модульність	☀ ☀ ☀ ☀
Загальна вартість	☀ ☀ ☀
Продуктивність	☀ ☀ ☀ ☀ ☀
Надійність	☀ ☀ ☀
Масштабованість	☀ ☀ ☀ ☀ ☀
Простота	☀
Тестованість	☀ ☀

Архітектура, заснована на подіях, отримує п'ять зірок за продуктивність, масштабованість та стійкість до відмов, як основні переваги цього стилю. Висока продуктивність досягається завдяки асинхронному спілкуванню в поєднанні з високопродуктивною паралельною обробкою. Велика масштабованість реалізується програмним балансуванням навантаження. Зі збільшенням навантаження запитів можна програмно додавати додаткові обробники подій для обробки додаткових запитів. Стійкість до відмов

досягається завдяки сильно роз'єднаним та асинхронним обробникам подій, які забезпечують послідовність та обробку робочих процесів.

Загальна простота та тестуваність оцінюються відносно низько для архітектури, заснованої на подіях, головним чином через недетерміновані та динамічні потоки подій, які зазвичай знаходяться в цьому стилі архітектури. В той час як детерміновані потоки в рамках моделі запит-відповідь взагалі досить прості для тестування, такий же сценарій не є типовим для моделі, заснованої на подіях. Іноді не відомо, як обробники подій реагуватимуть на динамічні події та які повідомлення вони можуть виробляти. Ці "діаграми подій" можуть бути надзвичайно складними, створюючи сотні або навіть тисячі сценаріїв, що робить їх важкими до управління та тестування.

Необхідно зазначити, що застосунки які засновані на event-driven архітектурі, є еволюційні, отже, отримали п'ять зірок за це. Додавання нових функцій через існуючі або нові обробники подій є досить простим, особливо в event-broker топології.

Висновки до другого розділу

Вивчення та розуміння різноманітних архітектурних стилів відкриває безліч аспектів, які впливають на процес проектування та реалізації програмного забезпечення. Кожен з цих стилів представляє собою унікальний підхід до вирішення проблем та задач розробки, при цьому враховуючи особливості конкретного контексту та завдань проекту.

Архітектурні стилі визначають не лише структуру системи, але й впливають на її функціональні та нефункціональні характеристики. Наприклад, мікросервісна архітектура, з своєю здатністю до автономного масштабування та високої доступності, надає гнучкість та стійкість до великої кількості запитів. З іншого боку, шарова архітектура полегшує утримання та модифікацію коду завдяки його структурованості та модульності.

Однак при виборі архітектурного стилю необхідно дбайливо аналізувати вимоги та особливості саме вашого проекту. Навіть найкращий стиль може

мати свої недоліки у конкретних умовах. Наприклад, мікросервісна архітектура може призвести до складнощів у керуванні та координації між сервісами. Окрім того, важливо враховувати, що архітектурні стилі можуть еволюціонувати, і нові підходи можуть з'являтися з часом. Тому важливо слідкувати за тенденціями у світі програмного забезпечення та постійно вдосконалювати свої знання.

Враховуючи ці аспекти та взаємодіючи з усіма можливими факторами, розробник може зробити обґрунтований вибір архітектурного стилю, що відповідає унікальним потребам та цілям конкретного проекту.

РОЗДІЛ 3. ПРАКТИЧНЕ ВПРОВАДЖЕННЯ СТИЛІВ АРХІТЕКТУРИ ТА ДИЗАЙНУ

3.1. Методи практичного впровадження архітектурних стилів

При впровадженні архітектурних стилів ключово враховувати зважені рішення для досягнення оптимальних параметрів системи. Зваженість полягає у врахуванні та балансуванні різних факторів, таких як продуктивність, масштабованість та надійність, з метою забезпечення оптимального результату. Важливо уникати однозначних рішень та враховувати потреби конкретного проекту. Наприклад, при розгляді архітектурного стилю слід враховувати його вплив на продуктивність, але не на шкоду іншим аспектам, таким як надійність чи масштабованість. Забезпечення відповідності вимогам різних сторін проекту вимагає уважного аналізу і зважених рішень. Наприклад, при виборі розподіленої архітектури можна отримати певні переваги в масштабованості, але це може вплинути на складність розробки та впровадження. Здатність знаходження компромісів у важливих аспектах системи визначає успіх впровадження архітектурних рішень. Збалансований підхід дозволяє максимізувати користь від вибраного стилю та одночасно задовольняти всі релевантні вимоги проекту.

Практичне впровадження архітектурних стилів включає в себе кілька етапів та методів, спрямованих на ефективну розробку та впровадження архітектур для певного програмного застосунку. Нижче розглянемо детально кожен етап та метод:

- Збір вимог: Ретельний аналіз функціональних та нефункціональних вимог до системи. Важливо визначити потреби користувачів, характеристики продуктивності, надійності, масштабованості та інші параметри.

- Оцінка вимог: Оцінка, які архітектурні стилі найкраще задовольняють вимоги системи. Рішення може базуватися на масштабованості, рівні надійності, зручності розробки та інших критеріях.
- Створення концептуальної моделі: Визначення основних компонентів, їхньої взаємодії та ролей в системі.
- Обрання підсистем: Визначення підсистем та їх функціональності. Обрання, чи будуть вони монолітними, розподіленими чи використовуватимуть інші архітектурні стилі.
- Розробка компонентів: Написання коду для окремих компонентів системи, дотримуючись визначеної архітектури.
- Тестування: Використання тестів для перевірки відповідності компонентів та системи в цілому вимогам та очікуванням.
- Деплоймент: Розгортання системи в реальному середовищі. Включає інсталяцію, конфігурацію та запуск додатків.
- Моніторинг: Спостереження за роботою системи, виявлення та виправлення можливих проблем.
- Підтримка: Надання технічної підтримки та виправлення помилок після впровадження системи.
- Оптимізація: Здійснення модифікацій та удосконалень для підвищення продуктивності, надійності чи інших характеристик системи.

Важливо зауважити, що впровадження архітектурних стилей є ітеративний процес. Це важливо для можливості адаптувати систему до змін у вимогах в процесі її розвитку. Ітеративний підхід дозволяє здійснювати поступові зміни та вдосконалення, надаючи можливість системі еволюціонувати відповідно до змін в бізнес-середовищі чи вимог користувачів. За допомогою ітерацій можна не лише ефективно впроваджувати архітектурні стилі, але й проводити експерименти, тестування та вдосконалення вже існуючих рішень. Це особливо корисно в умовах швидкозмінюваного ринкового середовища, де потрібно оперативно реагувати на нові вимоги та технологічні тенденції. Ітеративне впровадження дозволяє також зменшити

ризик впровадження нових архітектурних рішень. Замість того, щоб чекати на повний завершений продукт, команди можуть швидко впроваджувати і тестувати окремі частини системи, перевіряти їх ефективність та вносити корективи за потреби.

Такий підхід сприяє не лише вдосконаленню функціональних характеристик системи, але й підтримує її гнучкість та здатність швидко адаптуватися до змін, що робить ітеративне впровадження архітектурних стилів важливим елементом сучасного розробки програмного забезпечення.

3.2. Приклади дизайну систем

Вивчення архітектурних рішень у інших системах сприяє розширенню розуміння різноманітності підходів та їх ефективності в різних умовах. Аналізуючи, як інші проекти реалізували концепції архітектурних стилів, розробники та архітектори можуть зрозуміти, як ці стилі впливають на роботу системи під час реального використання. Це важливо для забезпечення оптимальної відповідності архітектурного рішення конкретним потребам та умовам проекту.

Додатково, аналіз існуючих систем може розкрити проблеми, які виникають при використанні певних архітектурних стилів та допомагає уникнути цих труднощів у власному проекті. При цьому важливо враховувати не лише позитивні аспекти, а й негативні або труднощі, з якими зіткнулися інші команди при впровадженні тих чи інших архітектурних рішень. Такий глибокий огляд сприяє створенню більш досконалої, стійкої та ефективної системної архітектури.

Для прикладу розглянемо архітектурні рішення усім відомих застосунків:

Magento 2 — eCommerce платформа з відкритим кодом, призначена для створення інтернет-магазинів. Основна функціональність цієї платформи побудована в Layered Architecture стилі і є, фактично, монолітним застосунком.

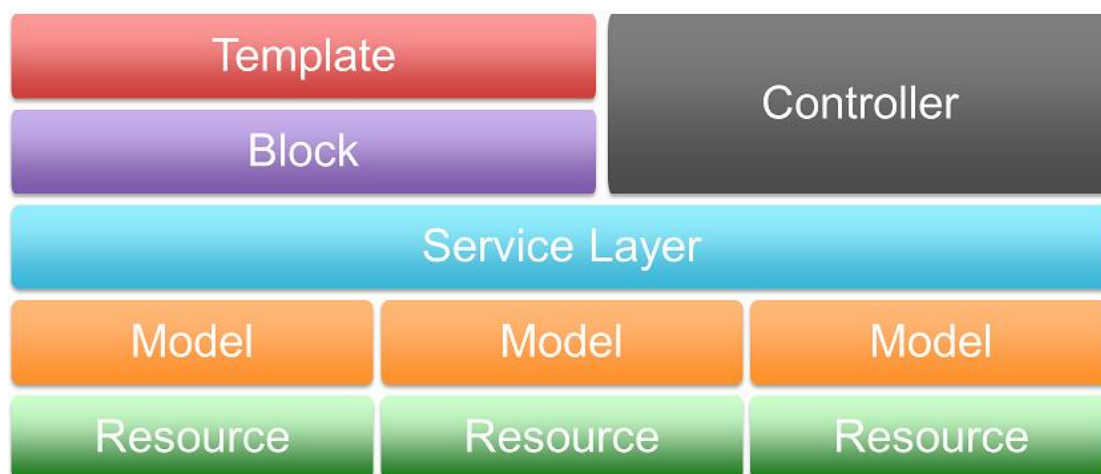


Рис. 3.1. Шари сервісів в платформі Magento 2

Інтеграція нових комерційних сервісів в Magento 2 відбувається з дотриманням принципів Service Oriented Architecture

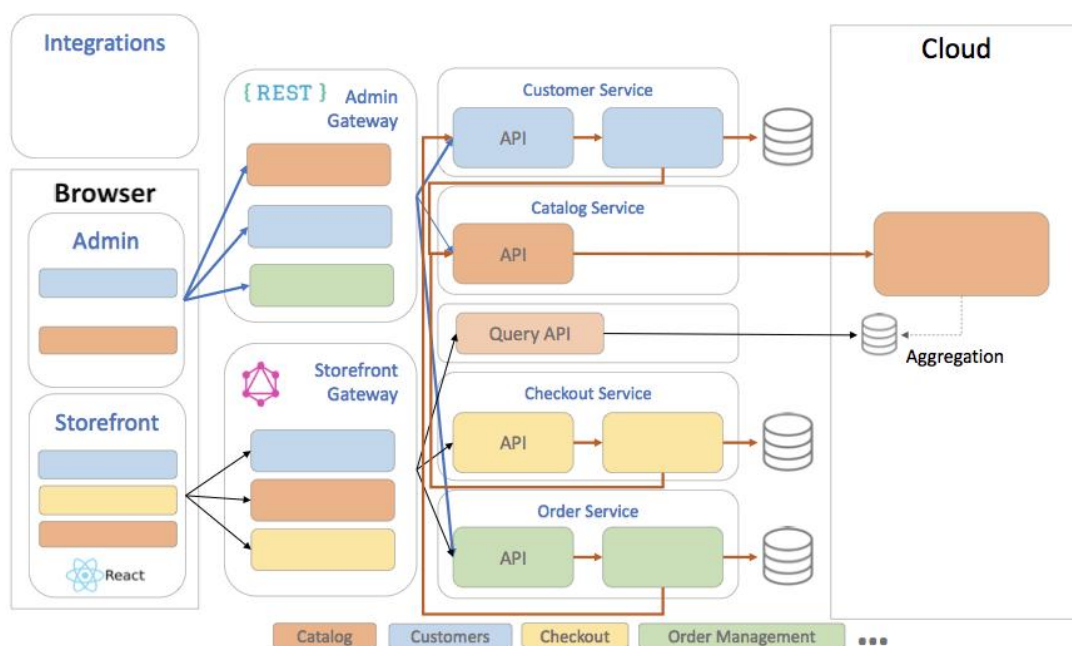


Рис. 3.2. Ілюстрація сервісів в платформі Magento 2

Вихідний код цієї системи частково розповсюджується у відкритому доступі, тому є можливість детально вивчити її.

Netflix - застосунок для надання передплатуваних послуг із мережевої трансляції бібліотеки кінофільмів та телепередач. Згідно з статтями архітекторів [24] які займалися чи займаються розробкою архітектури продукту Netflix - 15% трафіку світової мережі інтернет опрацьовується серверами Netflix. Саме через це їх архітектура є цікавою.

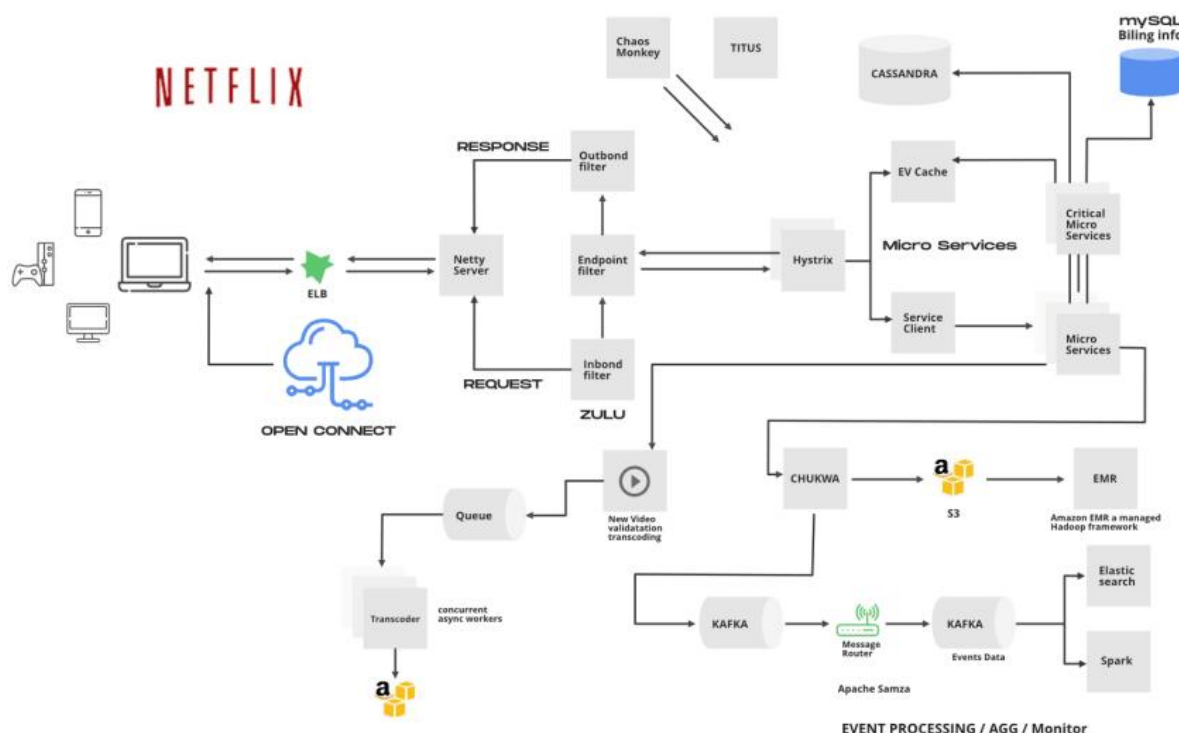


Рис. 3.3. Загальна архітектура Netflix

На діаграмі вище ми можемо бачити що продукт використовує велику кількість своїх, саморобних, технологій. Але між тим основний акцент поставлений на Microservices Architecture.

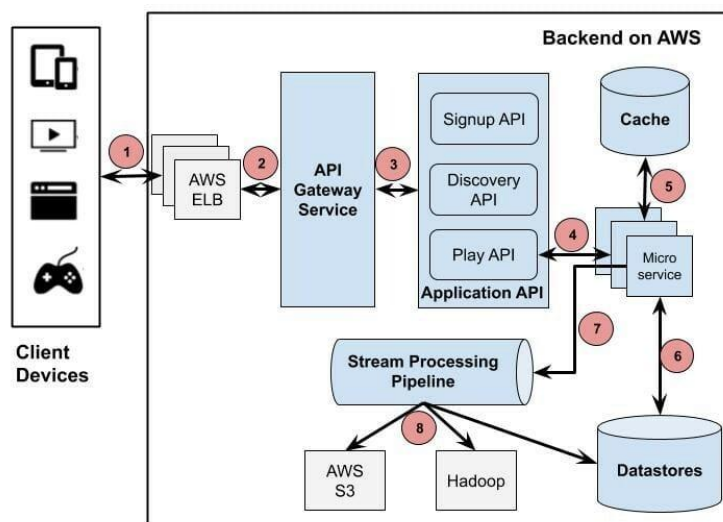


Рис. 3.4. Архітектура бекенду Netflix

На діаграмі бекенду ми бачимо що мікросервіси виконують основну частину роботи в сервісі. Доступ до мікросервісів відбувається через Шлюз (API Gateway) та API авторизації.

Instagram - соціальна мережа, що базується на обміні світлинами, дозволяє користувачам робити світлини, застосовувати до них фільтри, а також поширювати їх через свій сервіс і низку інших соціальних мереж. Фактично це застосунок з мільйонним трафіком, саме тому їх архітектура цікава в наукових цілях.

Бекенд архітектура Instagram не є публічною, тому інформації про неї мало. Проте, інженери з цієї компанії поділились в офіційному блозі [25] про архітектуру пошукового рушія в цій системі.

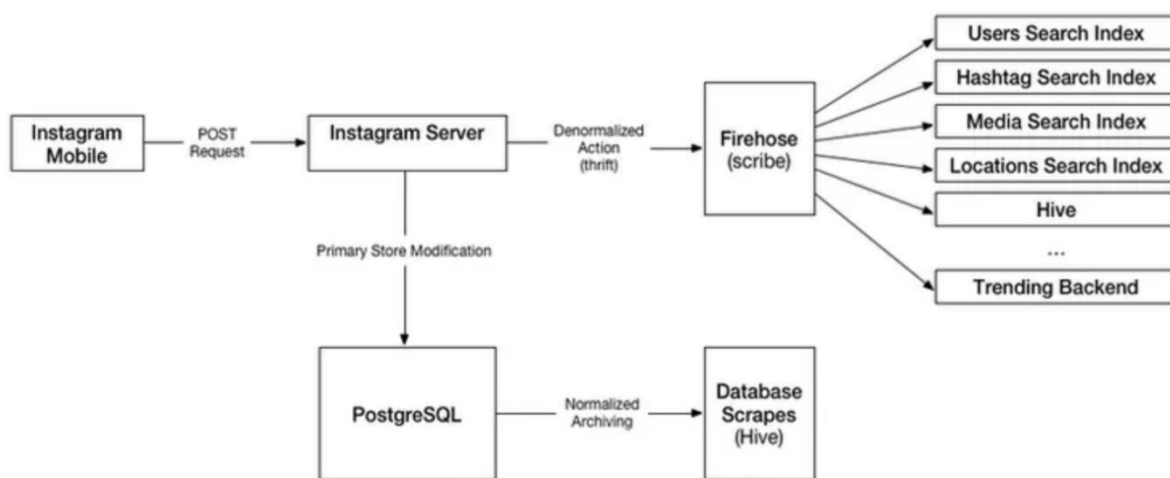


Рис. 3.5. Архітектура пошукового рушія в Instagram

Фактично основний архітектурний стиль цієї системи є Event-Driven Architecture. За допомогою цього при завантаженні нових даних до системи, усі залежні сервіси отримують інформацію про цю подію, додають нові дані до своїх баз даних та індексують їх.

Як ми бачимо, дослідження архітектурних стилів великих та масштабних застосунків є досить корисним в наукових та навчальних цілях.

3.3. Застосунок-прототип з використанням мішаного архітектурного стилю

Перевірка архітектурних стилів на практиці є ключовою для забезпечення їх відповідності конкретним вимогам проекту. Практичне тестування дозволяє виявити можливі проблеми та неочевидні обмеження стилів в реальних умовах

використання. Важливо перевіряти теоретичні знання про архітектурні стилі на практиці з кількох причин:

1. Валідація Теорії: Практичне застосування теоретичних концепцій і стилів дозволяє перевірити, наскільки вони відповідають реальному програмному середовищу. Це дозволяє підтвердити або виправити теоретичні уявлення, а також виявити можливі обмеження чи недоліки.

2. Застосування в Реальних Проектах: Перевірка теорії на практиці дає можливість визначити, наскільки успішно архітектурні стилі можуть бути використані в реальних програмних проектах. Це дозволяє науковцям розробляти рекомендації для практиків на основі практичного досвіду.

3. Виявлення Типових Помилки: Робота над реальними проектами дозволяє виявити типові помилки чи труднощі, які можуть виникнути при впровадженні архітектурних стилів. Це сприяє розвитку практичних рекомендацій для підвищення ефективності розробки.

4. Адаптація до Змін: Практичне тестування теорій дозволяє адаптувати свої підходи до нових вимог та змін у галузі розробки програмного забезпечення. Забезпечення гнучкості та актуальності у наукових дослідженнях допомагає їм бути корисними для індустрії.

Саме тому було прийнято рішення імплементувати прототип проекту використовуючи декілька архітектурних стилів для їх тестування та детального розгляду. В якості ідеї прототипу було обрано просту функціональність - система для перегляду інформації про кінострічки. Така система надає можливість зареєструватись, переглянути загальну інформацію про кінострічки, додати кінострічку в обране.

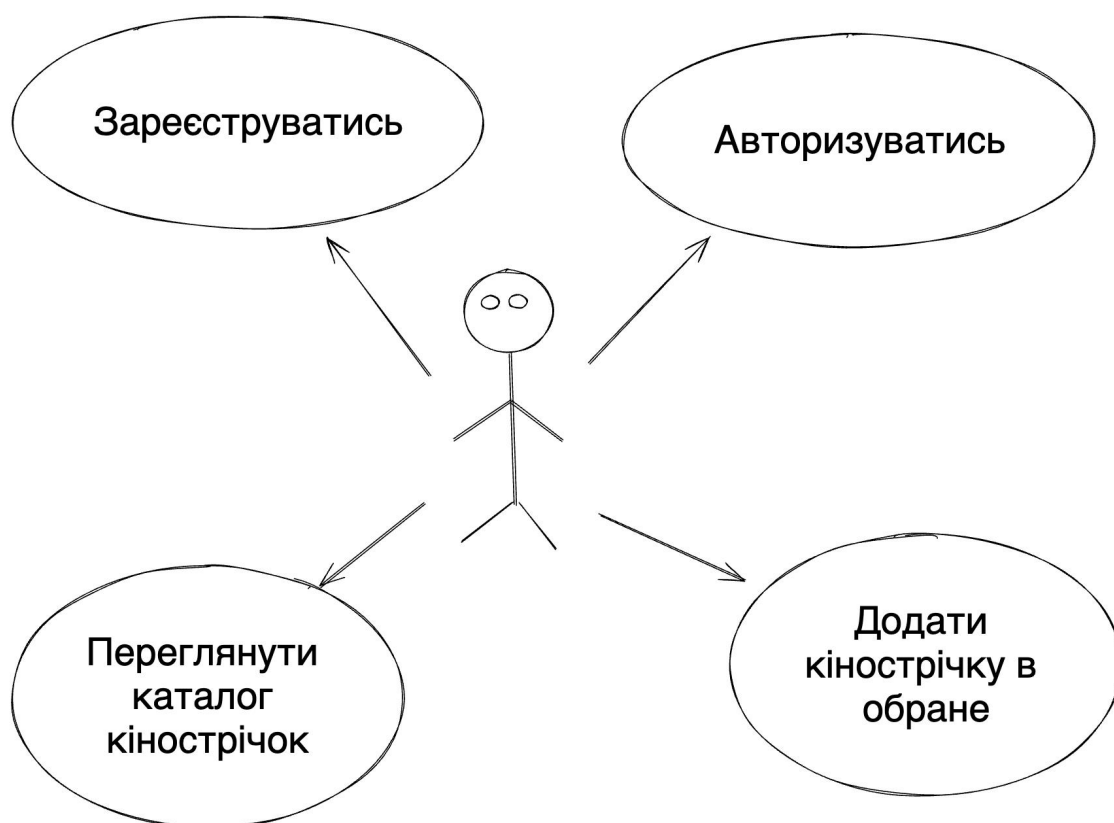


Рис. 3.6. Діаграма варіантів використання прототипу

Було прийняте рішення розробити систему базуючись на мікросервісному архітектурному стилі задля можливості масштабування під час теоретично можливих великих навантажень. Архітектура кожного мікросервісу використовує Layered Architecture стиль, для розподілення відповідальностей по шарам системи. Також, однією з бізнес вимог є прорахунок кількості разів додавання до обраного для кожної кінострічки. Для імплементації цієї функціональності було обрано Event-Driven Architecture стиль, що дозволяє асинхронно сповістити усі зацікавлені компоненти в тому, що кінострічка була додана до обраного.

Важливо додати, що в мікросервісному архітектурному стилі прийнято мати окреме сховище даних під кожен мікросервіс. Це дає змогу краще ізолювати функціональність від зовнішнього світу. Саме тому в прототипі було імплементовано саме такий підхід - кожен мікросервіс має своє особисте сховище даних.

В цілому, архітектура системи виглядає наступним чином:

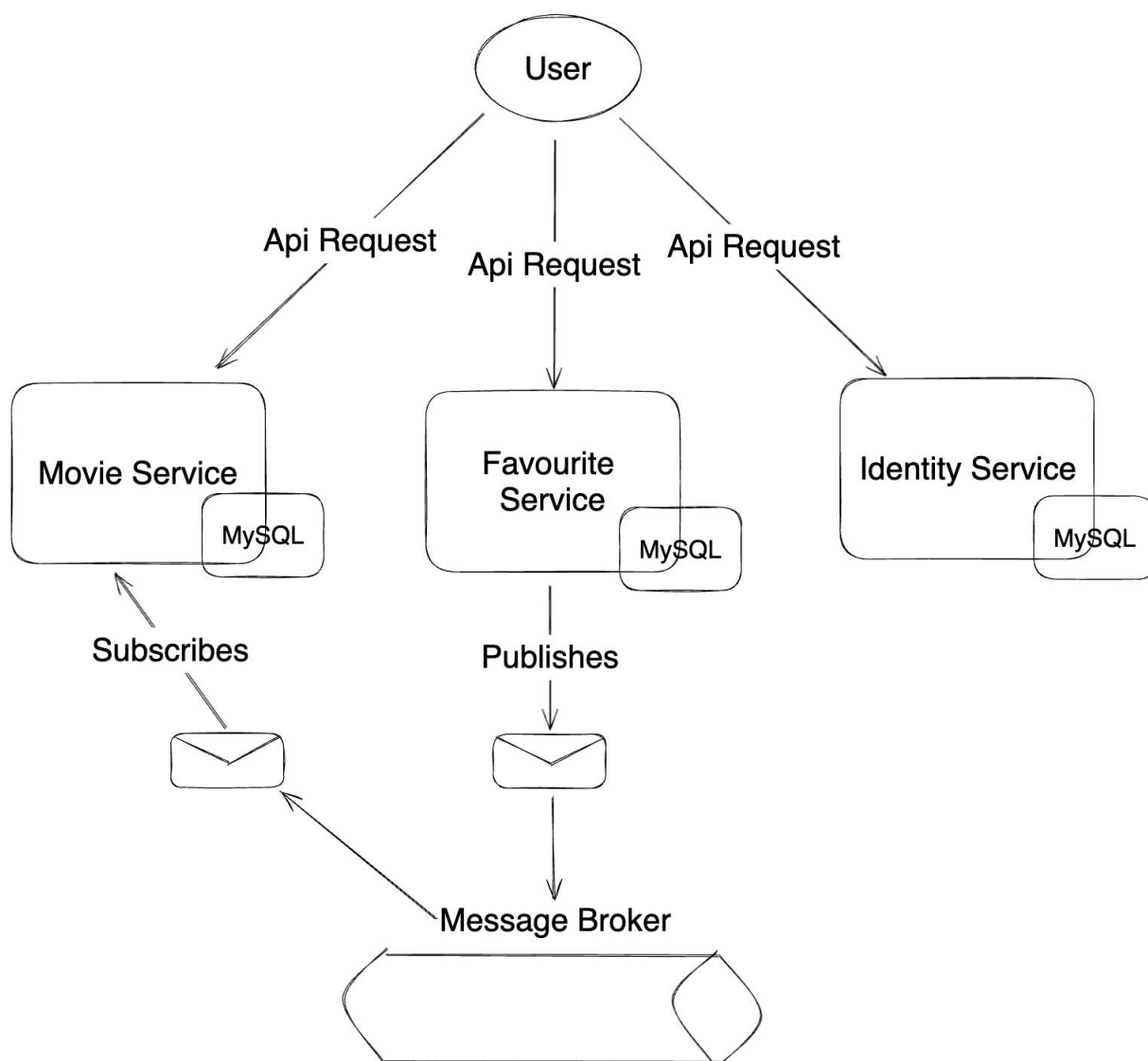


Рис. 3.7. Архітектура прототипу

Як відомо, імплементація мікросервісів не потребує використання одного і того ж стеку технологій для всіх сервісів, але в прототипі, для усіх сервісів, була обрана мова програмування PHP та фреймворк Laravel [23].

В якості Message Broker було обрано Redis, як storage-сервіс з великою пропускною здатністю. Основною причиною використання Redis як message broker є його висока швидкодія та можливості в роботі з повідомленнями та чергами (queues). Redis відомий своєю здатністю до швидкого зберігання та витягання даних в пам'яті, що робить його ефективним в ролі message broker для асинхронних операцій та обробки черг.

Так як імплементація прототипу не вимагає присутності інтерфейсу користувача, доступ до системи реалізований за допомогою API ендпоінтів.

Лістинг ендпоінтів системи прототипу

Сервіс	Метод	Ендпоінт	Параметри	Опис
identity	POST	/api/registration	email, password	Реєстрація нового користувача
identity	POST	/api/get-access-token	email, password	Генерація токену доступу
identity	GET	/api/user/{userId}	userId	Дані користувача
movie	GET	/api/details/{id}	id - ідентифікатор кінострічки	Дані кінострічки
movie	GET	/api/list?page={page}	page - поточна сторінка пагінації	Лістинг кінострічок
favourite	POST	/api/add-to-favourites/{movieId}	movieId	Додати кінострічку до списку обраних поточного користувача
favourite	GET	/api/favourites	-	Список обраних кінострічок поточного користувача

Таким чином усі бізнес вимоги були покриті зазначеними API ендпоінтами. Для протости було прийнято рішення дозволити звернення до конкретних сервісів напряму, так як це лише прототип. Але варто зазначити що

в реальних системах необхідно подумати про безпеку та закрити мікросервіси від зовнішнього світу хоча б за деяким API Gateway який займається внутрішнім роутингом.

Тестування реалізований проекту проходило на локальній машині. Усі сервіси є запущеними за допомогою вмонтованого в фреймворк Laravel веб-серверу.

```
vladyslavsikailo@VladSikailo-MacBook-Pro:~/projects/architecture-project/services/identity$ a serve
```

INFO Server running on [<http://127.0.0.1:8000>].

Press Ctrl+C to stop the server

Рис. 3.8. Запуск Identity мікросервісу

```
vladyslavsikailo@VladSikailo-MacBook-Pro:~/projects/architecture-project/services/movie$ a serve
```

WARN Failed to listen on 127.0.0.1:8000 (reason: Address already in use).

INFO Server running on [<http://127.0.0.1:8001>].

Press Ctrl+C to stop the server

Рис. 3.9. Запуск Movie мікросервісу

```
vladyslavsikailo@VladSikailo-MacBook-Pro:~/projects/architecture-project/services/favourite$ a serve
```

WARN Failed to listen on 127.0.0.1:8000 (reason: Address already in use).

WARN Failed to listen on 127.0.0.1:8001 (reason: Address already in use).

INFO Server running on [<http://127.0.0.1:8002>].

Press Ctrl+C to stop the server

Рис. 3.10. Запуск Favourite мікросервісу

Усі інші додаткові сервіси такі як база даних MySQL для (кожного мікросервісу) та Redis (один на весь проект) завантажуються за допомогою технології docker, а саме docker-compose:

```
vladyslavsikailo@VladSikailo-MacBook-Pro:~/projects/architecture-project$ docker-compose up -d
```

[+] Running 5/5

:: Network architecture-project_architecture-project	Created
:: Container redis	Started
:: Container architecture-project-db-identity-1	Started
:: Container architecture-project-db-movie-1	Started
:: Container architecture-project-db-favourites-1	Started

Рис. 3.11. Запуск MySQL та Redis

Безпосередньо спілкування з сервісами відбувається за допомогою застосунку Postman.

Перш за все, для тестування, необхідно створити користувача:

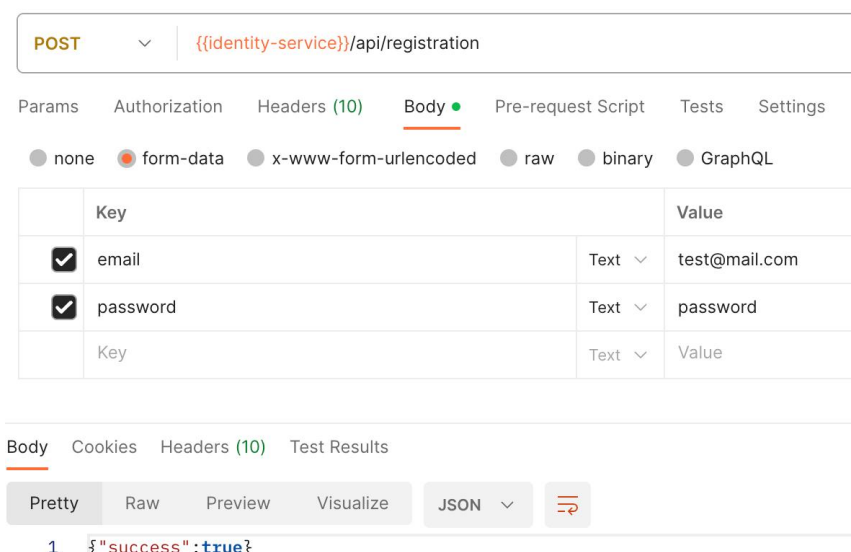


Рис. 3.12. Створення користувача в системі

Як ми бачимо з відповіді - користувач створений успішно. Наступним етапом є генерація ключа доступу, який необхідний для авторизації користувача в системі так як кожен мікросервіс займається авторизацією користувача перед обробкою запиту.

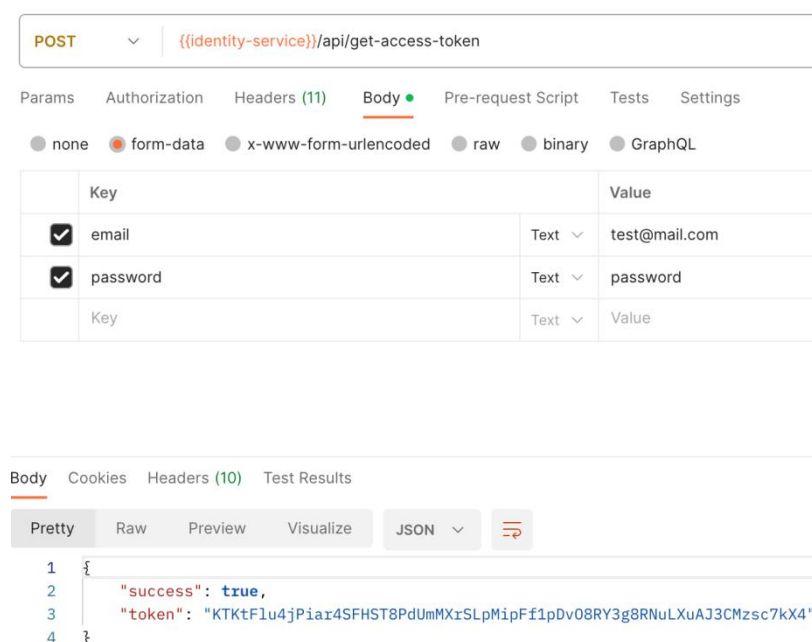


Рис. 3.13. Генерація ключу доступу

Як ми бачимо з відповіді - ключ доступу успішно згенерований. Наступний етап, в тестових цілях, є перегляд лістингу кінострічок:

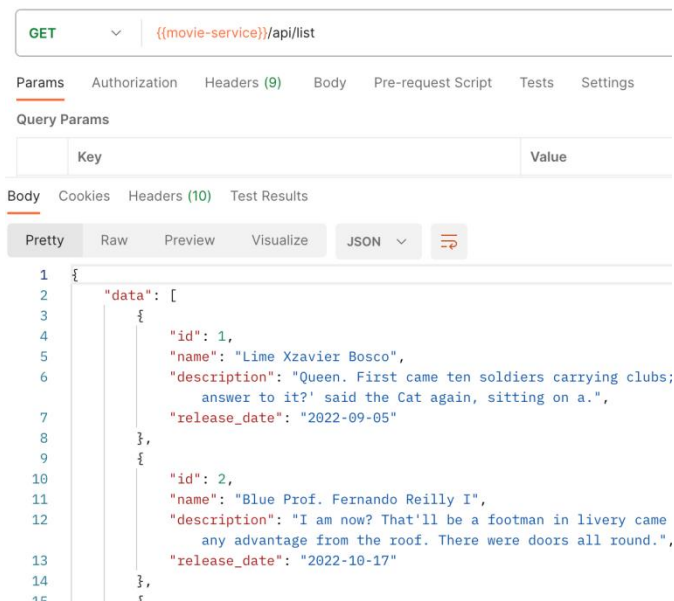


Рис. 3.14. Перегляд лістингу кінострічок

Як ми можемо бачити, лістинг містить певний набір даних про кожну кінострічку. Також, є змога проходити по лістингу з використанням пагінації - це необхідно для уникнення перевантаження сервісу (наприклад якщо треба буде віддати деталі про декілька тисяч кінострічок одночасно). Наступний тестовий крок - перегляд детальної інформації про кінострічку:

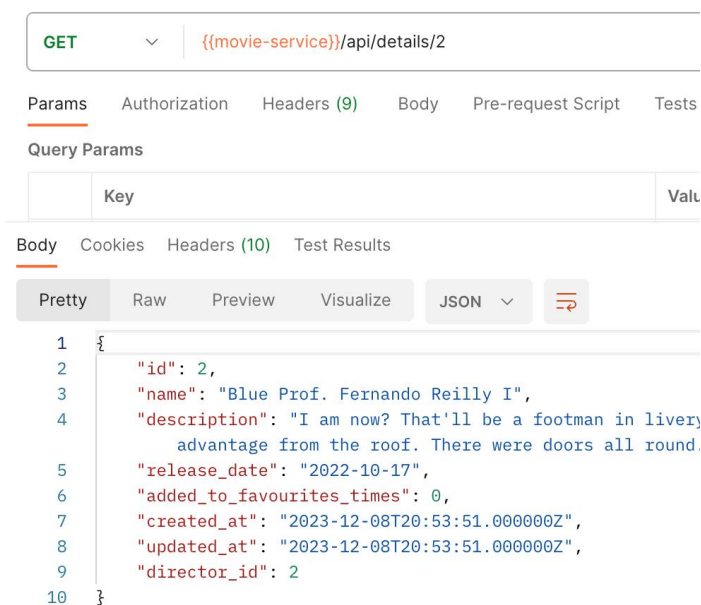


Рис. 3.15. Перегляд деталей кінострічки

З відповіді ми бачимо що деталі містять більшу кількість інформації про кінострічку ніж лістинг. До прикладу поле “added_to_favourites_times” містить кількість додавань до обраного конкретно цієї кінострічки. Наступним тестовим кроком є додавання кінострічки до списку обраного:



Рис. 3.16. Додавання кінострічки до списку обраних

І можемо переглянути список обраних кінострічок поточного користувача:

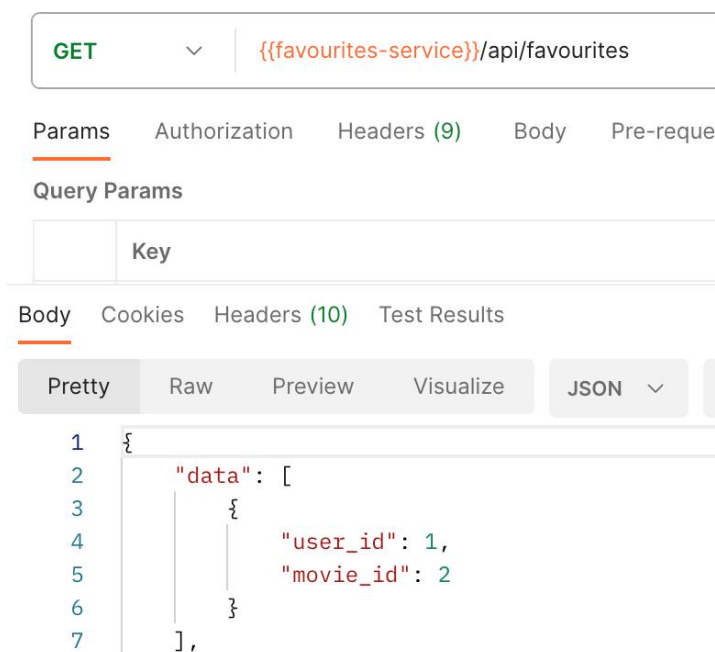


Рис. 3.17. Список обраних поточного користувача

Так як було імплементовано Event-Driven Architecture стиль, подія про додавання до списку обраного була створена та запущена в чергу. Для обробки черги ми використовуємо консюмери з фреймворку Laravel:

```
vladyslavsikailo@VladSikailo-MacBook-Pro:~/projects/architecture-project/services/movie$ a queue:work --queue=favourites -v
```

```
INFO Processing jobs from the [favourites] queue.
```

```
2023-12-13 19:32:49 App\Jobs\AddedToFavourite EGS65HU3UjJAF95nQjRoCg9o5mAfVD16 ..... RUNNING
```

```
2023-12-13 19:32:49 App\Jobs\AddedToFavourite EGS65HU3UjJAF95nQjRoCg9o5mAfVD16 ..... 45.15ms DONE
```

Рис. 3.18. Процесінг повідомлення з черги

Після процесінгу ми можемо побачити що кількість додання до обраного збільшилась у кінострічці яку ми додали:

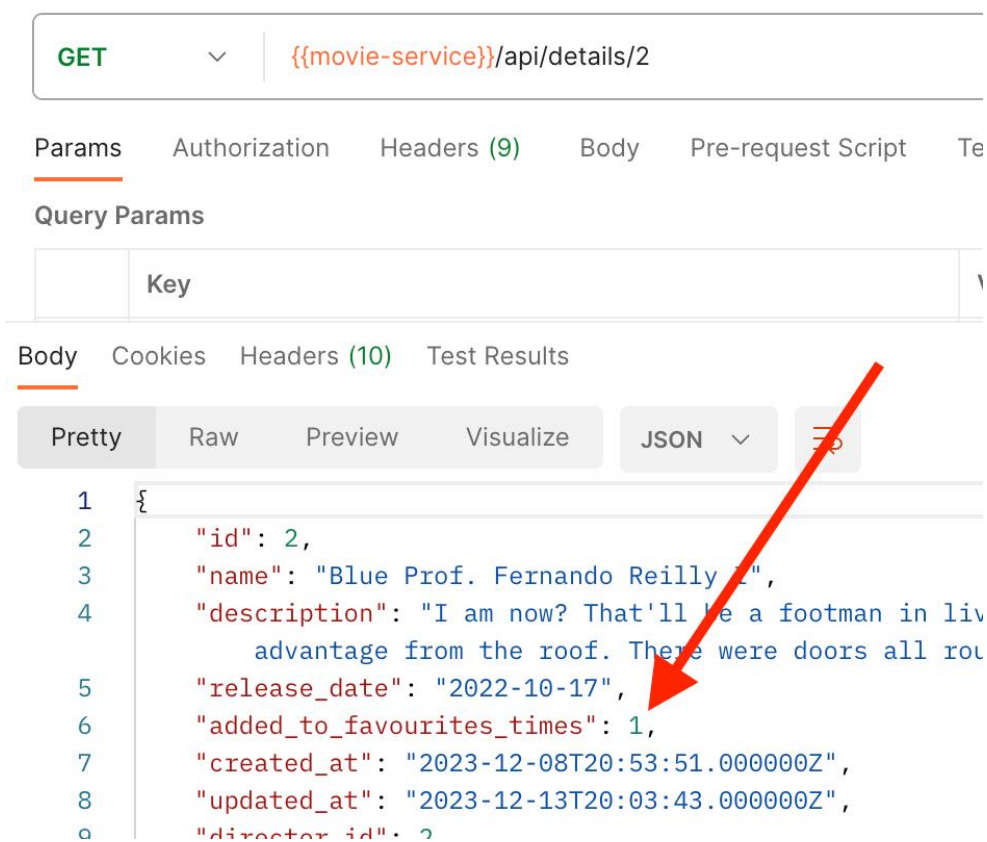


Рис. 3.19. Збільшена кількість доданого в обране

Як результат ми можемо побачити що Event-Driven Architecture працює для цього бізнес правила. Також необхідно зазначити, що для повної імплементатії цієї можливості необхідно також створити події та процесори подій на видалення зі списку обраного. Але ми маємо лише прототип, тому такої часткової імпелементації цілком достатньо для демонстрації можливостей цього архітектурного стилю.

Висновки до третього розділу

У розділі, присвяченому методам практичного впровадження архітектурних стилів, було розглянуто ключові підходи та рекомендації, спрямовані на успішну розробку та імплементацію програмного забезпечення. Використання архітектурних стилів у практиці виявилось необхідним для покращення якості та ефективності систем.

Проведено аналіз та надано приклади дизайну різних великих та масштабних систем. Ці приклади відображають реальні сценарії впровадження архітектурних стилів у різноманітних областях, підкреслюючи їх важливість та вплив на архітектурні рішення.

Детально описано імплементацію прототипу застосунку з використанням мішаного архітектурного стилю. Цей підхід дозволив поєднати переваги різних стилів, створюючи гнучку та оптимізовану систему. Розглянуті деталі імплементації сприяють розумінню практичного застосування архітектурних концепцій у реальних проектах.

Усе вищезазначене підкреслює важливість та актуальність вивчення архітектурних стилів для створення високоефективних, масштабованих та надійних програмних систем. Отримані знання та методи практичного застосування є необхідними для розробників та архітекторів у сучасному світі програмної інженерії.

ВИСНОВКИ

У ході проведеного дослідження було ретельно розглянуто та обґрунтовано актуальність впровадження гнучкої архітектури в сфері веб-застосунків. Проаналізовано проблематику дизайну та архітектури програмного забезпечення, а також висвітлено важливість вибору відповідних архітектурних стилів для побудови ефективних та масштабованих систем.

В розділі, присвяченому аналізу існуючих архітектурних стилів, детально висвітлено поняття чистого коду як основи для конструювання архітектури системи. Також проведено докладний огляд різних архітектурних стилів та їх топологій, включаючи оцінку їх характеристик.

У третьому розділі здійснено практичне впровадження стилів архітектури та дизайну систем. Описано методи практичного впровадження архітектурних стилів та приведено приклади дизайну систем. Детально розглянуто застосунок-прототип з використанням мішаного архітектурного стилю, який об'єднує Microservices, Layered Architecture та Event-Driven Architecture.

Загальні результати дослідження свідчать про те, що вибір та правильна імплементація архітектурних стилів є критичним для успішного розвитку та функціонування веб-застосунків. Впроваджені методи та розглянуті приклади надають фундаментальні знання для розробників та архітекторів, що дозволяють підвищити якість та швидкість розробки, а також забезпечити високу ефективність системи.

Висновки щодо практичної цінності результатів підкреслюють важливість вибору архітектурних стилів у веб-розробці та їх вплив на різні аспекти функціонування системи. Зазначається також необхідність подальших наукових досліджень у напрямку вдосконалення програмних систем та розширення практичних можливостей використання архітектурних стилів у масштабних веб-застосунках.

ЛІТЕРАТУРА

1. Мартін Р. Чиста архітектура / Роберт Мартін., 2019. – 368 с.
2. Richards M. Fundamentals of Software Architecture / M. Richards, F. Neal., 2020. – 396 с.
3. Fowler M. Patterns of Enterprise Application Architecture / Martin Fowler., 2002. – 560 с.
4. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software / Eric Evans., 2003. – 560 с.
5. Vernon V. Implementing Domain-Driven Design / Vaughn Vernon., 2013. – 614 с.
6. Khononov V. Learning Domain-Driven Design / Vlad Khononov., 2022.
7. Booch G. Object-Oriented Analysis and Design with Applications / Grady Booch., 2007. – 720 с. – (3).
8. Keeling M. Design It!: From Programmer to Software Architect / Michael Keeling., – 360 с. – (2017).
9. Newman S. Building Microservices: Designing Fine-Grained Systems / Sam Newman., 2015. – 278 с.
10. Hohpe G. The Software Architect Elevator / Gregor Hohpe., 2020.
11. Ford N. Software Architecture: The Hard Parts / N. Ford, M. Richards., 2022.
12. Gorton I. Foundations of Scalable Systems / Ian Gorton., 2022. – 339 с.
13. Ford N. Building Evolutionary Architectures / N. Ford, R. Parsons., 2022. – 265 с. – (2).
14. Newman S. Monolith to Microservices / Sam Newman., 2020. – 272 с.
15. Mitra R. Microservices Up & Running / R. Mitra, I. Nadareishvili., 2021. – 319 с.
16. Bellemare A. Building Event-Driven Microservices / Adam Bellemare., 2020. – 324 с.
17. Сікайло В.О., Кравченко С.М. Необхідність проектування гнучкого дизайну для монолітної архітектури. Тези доповідей V Всеукраїнської науково-технічної конференції: «Комп'ютерні технології: інновації, проблеми,

- рішення» 01-02 грудня 2022 року. Житомир: “Житомирська політехніка”, 2022. С.53-54.
18. Сікайло В.О., Кравченко С.М. Важливість написання юніт-тестів для програмного коду. Тези доповідей III Всеукраїнської науково-практичної інтернет-конференції: «Сучасні комп’ютерні та інформаційні системи і технології» 12-19 грудня 2022 року. Запоріжжя: “Таврійський державний агротехнологічний університет”, 2022. С.451-452.
 19. Richards M. Developer to Architect [Електронний ресурс] / Mark Richards – Режим доступу до ресурсу: <https://developertoarchitect.com/>.
 20. Fowler M. Personal Blog [Електронний ресурс] / Martin Fowler – Режим доступу до ресурсу: <https://martinfowler.com/>.
 21. Event Sourcing pattern [Електронний ресурс] – Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing>
 22. Martin R. The Clean Code Blog [Електронний ресурс] / Robert Martin – Режим доступу до ресурсу: <https://blog.cleancoder.com/>.
 23. Otwell T. Laravel - The PHP Framework For Web Artisans [Електронний ресурс] / Taylor Otwell – Режим доступу до ресурсу: <https://laravel.com/docs/10.x/readme>.
 24. Olugbenga E. Netflix System Design- Backend Architecture [Електронний ресурс] / Elegberun Olugbenga – Режим доступу до ресурсу: <https://dev.to/gbengelebs/netflix-system-design-backend-architecture-10i3>.
 25. Instagram Engineering: Search Architecture [Електронний ресурс] – Режим доступу до ресурсу: <https://instagram-engineering.com/search-architecture-eeb34a936d3a>.
 26. Foo D. Software Architecture and Design Trend 2023 [Електронний ресурс] / Daniel Foo – Режим доступу до ресурсу: <https://danielfoo.medium.com/software-architecture-and-design-trend-2023-f55ecfbcfcc0>.
 27. What is an Event-Driven Architecture? [Електронний ресурс] – Режим доступу до ресурсу: <https://aws.amazon.com/event-driven-architecture/>.

28. Mooney T. A Year with Redis Event Sourcing [Электронный ресурс] / Tom Mooney – Режим доступа до ресурсу: <https://medium.com/lcom-techblog/a-year-with-redis-event-sourcing-lessons-learned-6736068e17cc>.