

User name:  
**Volodymyr Matiiievskyi**

Check ID:  
**1013523876**

Check date:  
**17.01.2023 22:41:39 EET**

Check type:  
**Doc vs Internet**

Report date:  
**17.01.2023 22:46:21 EET**

User ID:  
**100010994**

File name: **кваліф\_робота\_магістра\_ІПЗ\_Переяславська\_С\_О\_перевірка**

Page count: **62** Word count: **9161** Character count: **72265** File size: **1.61 MB** File ID: **1013287632**

## 2.67% Matches

Highest match: **0.52%** with Internet source (<https://arabicprogrammer.com/article/23373294833>)

2.67% Internet sources 37

Page 64

No Library search was conducted

## 0.14% Quotes

Quotes 2

Page 65

Exclusion of references is off

## 0.13% Exclusions

Some exclusions were automatic (exclusion filters: matched word count less than **10 words** and **0%**)

0.13% Internet exclusions 21

Page 66

No Library exclusions

## Modifind

Text modifications detected. Find more details in the online report.

Replaced characters 1

## ВСТУП

Одною з провідних завдань сучасної ІТ-компанії полягає в оптимізації своїх внутрішніх процесів та технологій з метою зниження витрат та підвищення затребуваності ІТ-продуктів. Управління вимогами, змінами, інцидентами та проблемами — усі ці процеси є об'єктами для оптимізації, але практика показала, що цього не достатньо. Застосування сервіс-орієнтованої архітектури (SOA) при побудові ІТ-рішення може дати набагато більший ефект, оскільки переваги SOA не так у короткочасному скороченні витрат, як у посиленні адаптивності інформаційних систем і всієї компанії в цілому.

Сервіс-орієнтована архітектура (SOA) — це шаблон архітектурного проектування, заснований на окремих частинах програмного забезпечення, що надає функціональні можливості програми як послуги для інших програм через протокол. SOA стає ефективним рішенням в вирішенні проблем, пов'язаних зі швидкими змінами в бізнес-середовищі та, як наслідок, зміною вимог до програмних продуктів. Таким чином, сервіс-орієнтована архітектура допомагає досягти балансу між вартістю та швидкістю.

Технології проектування та розробки сервіс-орієнтованої архітектури приділяється багато уваги в вітчизняних та закордонних дослідженнях. Так, еволюційний розвиток архітектури програмних систем після визначення «Big ball of mud»-додатків вивчали М. Malhotra, J. Carnell, в працях А. Zalewski, М. Szlenk, S. Kijas досліджується еволюція моделей SOA. Дослідженню самої сервіс-орієнтованої архітектури та її переваг присвячені роботи С. G. Ezekwe, Р. Okwu, F. E. Onuodu. Багато уваги приділяється мікросервісної архітектурі, як різновиду SOA (S. Pittet, J. Carnell, С. Richardson). Аналіз робіт доводить актуальність технологій, пов'язаних з сервіс-орієнтованою архітектурою та доцільність теми дослідження.

**Мета роботи** - дослідження сучасних підходів та технологій розробки сервіс-орієнтованих застосувань на платформі Java.

**Об'єкт дослідження** – технологій розробки сервіс-орієнтованих застосувань на прикладі мікросервісів.

**Предмет дослідження** - програмний додаток, спрямований на реалізацію основних концепцій мікросервісної архітектури, застосування сучасних патернів проектування мікросервісів.

Для досягнення даної мети повинні бути вирішені наступні **завдання**:

1. Дослідити сучасні підходи до розробки сервіс-орієнтованих застосувань.
2. Проаналізувати особливості мікросервісної архітектури, визначити переваги та недоліки.
3. Дослідити технології ефективної розробки сервіс-орієнтованих Java-застосувань (на прикладі мікросервісів)
4. Розробити програмний додаток, спрямований на реалізацію основних концепцій мікросервісної архітектури.

**Методи дослідження:**

*Теоретичні методи:* аналіз науково-технічних джерел з проблем дослідження. *Емпіричні методи:* порівняльний аналіз сучасних підходів до розробки сервіс-орієнтованих застосувань. *Експериментальні методи:* тестування розробленого додатку.

**Практичне завдання.** Досліджено сучасні технології розробки сервіс-орієнтованих застосувань на платформі Java. Проаналізовано підходи до проектування та розробки мікросервісів. Розроблено додаток, спрямований на реалізацію основних концепцій мікросервісної архітектури, застосування сучасних патернів проектування мікросервісів на базі фреймворку Spring Boot та мови програмування Java.

**Апробація роботи.** Матеріали роботи доповідалися на 22 міжнародній науково-практичній конференції «Modern aspects of modernization of science: status, problems, development trends», яка проходила 7 липня 2022 р. в м. Любляна (Словенія). За результатами конференції надрукована стаття за

темою «Approaches to architectural solutions of enterprise software based on services and microservices» [23].

**Структура дипломної роботи.** Робота складається з пояснювальної записки, списку використаних джерел, додатків. Обсяг роботи становить 70 сторінок, обсяг використаної літератури - 40 джерел.

В пояснювальній записці перший розділ містить дослідження підходів до розробки сервіс-орієнтованих застосувань. Розглянуто еволюцію SOA. Окрема увага приділяється мікросервісній архітектурі, як різновиду SOA. Проаналізовано патерни проектування, міжпроцесна комунікація в мікросервісах та технології розробки, керування, масштабування та розгортання MSA-додатку.

У другому розділі розроблена архітектура додатку на базі мікросервісів та розглянута програмна реалізація основних концепцій мікросервісної архітектури, застосування сучасних патернів проектування мікросервісів.

Додаток містить в собі код програми.

## РОЗДІЛ 1

### СУЧАСНІ ПІДХОДИ ДО РОЗРОБКИ СЕРВІС-ОРІЄНТОВАНИХ ЗАСТОСУВАНЬ

#### 1.1. Еволюція сервіс-орієнтованої архітектури

Розвиток інформаційних технологій призвело до значного ускладнення програмного забезпечення (ПЗ), яке втрачало гнучкість і керованість. В цьому випадку ігнорування застосування архітектурних підходів призводить до появи «Big ball of mud»-додатків з нерозпізнаною архітектурою та погано структурованим кодом, що дуже ускладнює його опрацювання.

За роки розвитку ПЗ-розробникам вдалося напрацювати надійні підходи, щоб усунути недоліки проектування без архітектури.

Malhotra M. [17] розглядає наступні підходи:

- багат шарова архітектура (Layered Architecture).
- багаторівнева архітектура (Tiered architecture).
- сервіс-орієнтована архітектура (Service Oriented Architecture—SOA).
- мікросервісна архітектура (Microservice Architecture).

На наш погляд, цей перелік треба доповнити **МОНОЛІТНОЮ архітектурою**, яка має зв'язок з перерахованими вище в контексті структури та еволюції розвитку.

Монолітна архітектура (МА) є односистемним додатком (рис. 1.1) лише з одним магістральним кодом та інфраструктурою. Програмний додаток може мати модульну структуру, яка включає кілька служб на одному рівні, але «не є незалежними від програми, до якої вони належать» [1].



Рис. 1.1. Структура монолітної архітектури [24]

Монолітне програмне забезпечення спроектоване так, щоб бути автономним, в якому компоненти або функції програми тісно пов'язані між собою. У шаблоні моноліту все, від інтерфейсу користувача, бізнес-кодів і викликів бази даних, включено в одну кодову базу. Всі проблеми програм містяться в одному великому розгортанні.

У монолітній архітектурі кожен компонент та пов'язані з ним компоненти повинні бути присутніми для виконання або компіляції коду та для запуску програмного забезпечення. Монолітні програми є однорівневими, що означає об'єднання декількох компонентів в одну велику програму. Отже, вони, як правило, мають великі кодові бази, управління якими з часом може стати громіздким.

Монолітна архітектура має свої переваги, тому багато програм досі створюються з використанням цієї парадигми розробки. По-перше, монолітні програми можуть мати кращу продуктивність, ніж модульні програми. Їх також може бути легше тестувати і налагоджувати.

5

Єдина кодова база також полегшує ведення журналів, управління конфігурацією, моніторинг продуктивності додатків та інші завдання розробки. Тим не менш, монолітний підхід зазвичай краще підходить для простих та легких додатків. Для більш складних програм із частими очікуваними змінами коду або зростаючими вимогами до масштабованості цей підхід не підходить.

Як правило, монолітні архітектури мають недоліки, які можуть затримувати розробку та розгортання додатків. Ці недоліки стають особливо суттєвими зі збільшенням складності продукту чи зі збільшенням розміру команди розробників.

Кодову базу монолітних додатків може бути важко зрозуміти, оскільки вона може бути великою, що може утруднити для нових розробників зміну коду відповідно до бізнес та технічних вимог, що можуть змінюватися. У міру того, як вимоги розвиваються або ускладнюються, стає складно правильно реалізувати зміни, які не погіршують якість коду і не впливають на роботу програми в цілому.

Після кожного оновлення монолітної програми розробники повинні компілювати всю кодову базу та повторно розгортати всю програму, а не лише оновлену частину. Це ускладнює безперервне або регулярне розгортання, що потім впливає на гнучкість програми та команди.

*Багатошарова архітектура* працює за принципом поділу відповідальності. ПЗ розділено на шари (шар уявлення (Presentation Layer), шар бізнес-логіки (Business Logic Layer), шар передачі даних (Data Link Layer)), що лежать один на одному, і кожен з них виконує певні функції (рис.1.2). Дані та елементи управління проходять через кожен шар у дизайні та передаються від одного до іншого. Ця система також підвищує рівень абстракції та певною мірою навіть стабільність ПЗ.

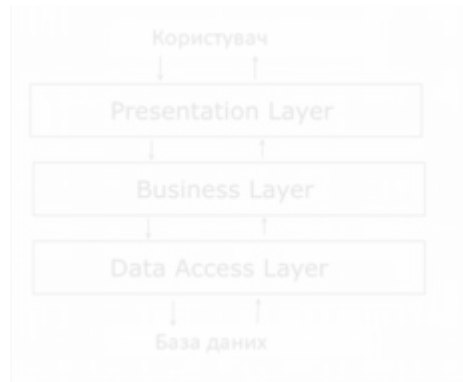


Рис.1.2. Багатошарова архітектура

Серед переваг цієї архітектури є простіша реалізація порівняно з іншими підходами, збільшення абстракції завдяки розподілу відповідальності між рівнями. Серед недоліків виокремлюють відсутність великої масштабованості, тому що ПЗ, створене з таким підходом, матиме монолітну структуру, що ускладнює внесення модифікацій.

Одним з поширених типів архітектури корпоративного програмного забезпечення є **багаторівнева** або *n-рівнева архітектура*. Цей архітектурний підхід поділяє комплекс на рівні за принципом взаємодії "клієнт-сервер". Додатки з цією архітектурою поділяються на кілька рівнів, кожен зі своїми обов'язками і функціями, такими як інтерфейс користувача, служби, дані, тестування тощо (рис.1.3).

У великих корпоративних системах *n-рівневі* програми мають безліч переваг [6, 17], у тому числі: *n-рівневі* програми дозволяють чітко розділити завдання і розглядати такі елементи, як інтерфейс користувача, дані та бізнес-логіку окремо. Такі архітектури мають високу масштабованість як по горизонталі, так і по вертикалі. Реалізація *n-рівневої* системи, як правило, коштує дорожче, але забезпечує високу продуктивність. Тому вона зазвичай застосовується у великих та комплексних програмних рішеннях.



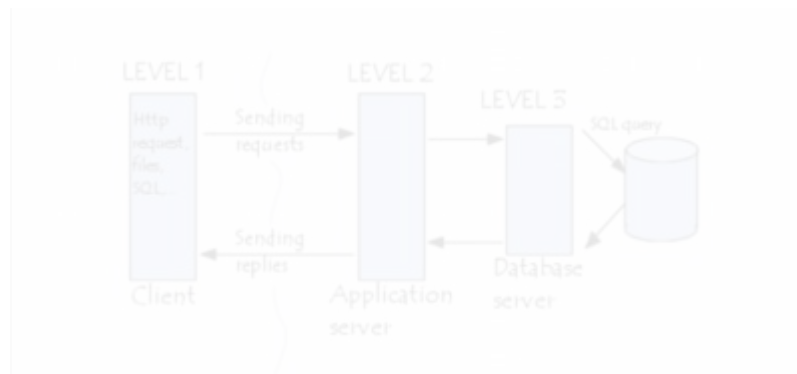


Рис.1.3. Багаторівнева архітектура

Але n-рівневі програми мають і недоліки: після внесення змін до коду доводиться зупиняти та повторно запускати всю програму; повідомлення, як правило, курсують вгору і вниз через рівні, що може бути неефективним; рефакторинг великого багаторівневого застосування після розгортання може виявитися складним завданням.

*Сервіс-орієнтована архітектура* — це шаблон архітектурного проектування, заснований на окремих частинах програмного забезпечення, що надає функціональні можливості програми як послуги для інших програм через протокол [7].

Основним припущенням, що лежить в основі сервіс-орієнтованої архітектури є концепція розробки нової функціональності (тобто нових послуг) шляхом композиції слабо пов'язаних (тому легко модифікованих), незалежних послуг (сервісів) (рис.1.4.). Отже, композиція сервісу має стати основним способом розробки нових функціональних можливостей.

Сервісно-орієнтована архітектура, по суті, сукупність сервісів, які взаємодіють між собою. Кожен сервіс надає бізнес-можливості, і сервіси можуть взаємодіяти один з одним на різних платформах і мовах. Розробники застосовують SOA для багаторазового використання сервісів у різних системах чи об'єднання кількох незалежних сервісів до виконання складних завдань.



Рис.1.4. Сервіс-орієнтована архітектура

У наведеному вище контексті ми розуміємо сервіс-орієнтовану систему як набір бізнес-процесів, що складається з послуг (незалежно від їх внутрішнього чи зовнішнього походження). Сервіс-орієнтована архітектура (SOA) – це розподілена архітектурна структура, яка надає рішення на основі набору взаємодіючих сервісів

Повна структура SOA-моделей має багат шарову структуру, яка складається з наступних послідовних рівнів [36]:

- цілі бізнес-мотивації, потреби тощо;
- моделі бізнес-процесів;
- склад послуг;
- моделі послуг (виражені моделями, які просуваються такими підходами, як SOMA [3], SOMF [4], використовується лише для розробки та розвитку послуг);

- низькорівневі, детальні технічні моделі (як правило, UML) і виконуваний код.

SOA з'явилася як архітектурний підхід, який підвищує продуктивність надання послуг існуючих традиційних систем, зберігаючи при цьому їх найважливіші функції [15]. Ця архітектура привернула увагу завдяки низці переваг, які вона пропонує, як-то: слабо пов'язані сервіси, незалежність від платформи, гнучкість тощо.

На думку авторів [11], основною перевагою SOA є можливість одночасного використання та легкого взаємного обміну даними між програмами різних виробників без додаткового програмування чи внесення змін до послуг. Ці послуги також можна використовувати багаторазово, що призводить до зниження витрат на розробку та обслуговування та забезпечує більшу цінність після розробки та випробування послуги. Наявність багаторазових послуг, які можна легко використовувати, також сприяє швидкому виходу на ринок.

Тут повторне використання бізнес-процесів важливіше, ніж повторне використання технології, оскільки SOA визначає ті самі бізнес-діяльності та об'єднує їх у групу як послугу. Отже, SOA зменшить дублювання додатків шляхом зменшення реплікації процесів. Крім того, використовуючи невеликі незалежні взаємопов'язані сервіси замість складних монолітів, отримуємо низку стійких переваг, які були виявлені в загальних ІТ та хмарних контекстах:

- збільшення повторного використання коду, зниження складності; значно швидша гнучка розробка та цикли тестування завдяки меншій кількості залежностей у коді;
- безперервна автоматизована та інтегрована розробка, тестування, розгортання, експлуатація та технічне обслуговування (наприклад, DevOps або DevSecOps), а також менша кількість необхідної підтримки внаслідок меншої кількості помилок.

Також, сервісна орієнтація зберігає переваги розробки на основі компонентів (самоопис, інкапсуляція, динамічне розгортання та завантаження), але є зрушення в парадигмі від віддаленого виклику методів для об'єктів до передачі повідомлень між службами. Це сприяє взаємодії та забезпечує переваги адаптації, оскільки повідомлення можуть надсилатися від однієї служби до іншої без урахування того, як була реалізована служба, яка обробляє ці повідомлення. повідомлень, а також поведінкові контракти для визначення прийнятних моделей обміну повідомленнями та політики для визначення семантики служби.

Але SOA стикається з деякими проблемами та обмеженнями, пов'язаними з вибором і виявленням сервісів, спільною роботою служб, композицією сервісів, міжпроцесним зв'язком між сервісами тощо [36]. Так, спільне використання баз даних у SOA може сприяти створенню своєрідного тісного зв'язку даних між сервісами та іншими компонентами системи, що призводить до появи небажаних ситуацій. Інша проблема полягає в тому, що сервіси в SOA, як правило, мають «великомодульну структуру», що може ускладнювати їх повторне використання [3].

Сервісна орієнтація (SO) є природною еволюцією поточних моделей розвитку. 80-ті роки XX століття побачили об'єктно-орієнтовані моделі; потім у 90-х з'явилася модель розробки на основі компонентів; останні роки маємо сервісну орієнтацію (SO).

Еволюція — це керований бізнесом процес, у якому система змінюється у відповідь на потреби бізнесу, що постійно виникають або змінюються [36]. Етап еволюції — це перехід між двома послідовними версіями моделей SOA. Це перетворення досягається шляхом поширення змін через багатшарову модель. Ця концепція відображена у моделі еволюції SOA (рис. 1.5), яку розробили ряд дослідників [36].

На думку авторів, еволюція складається з «Evolution Step», який складається з набору «Evolution decisions», що відображають відповідні потреби бізнесу. Еволюційні рішення складаються з двох різних наборів

рішень, що представляють зміни в моделях: «рішення щодо бізнес-процесів» і «комплект послуг». Перші представляють зміни, внесені в бізнес-процеси у відповідь на бізнес-потреби (бізнес-мотивація), а другі — зміни в складі послуг, зроблені для пристосування складу послуг до змін, внесених у бізнес

процеси.

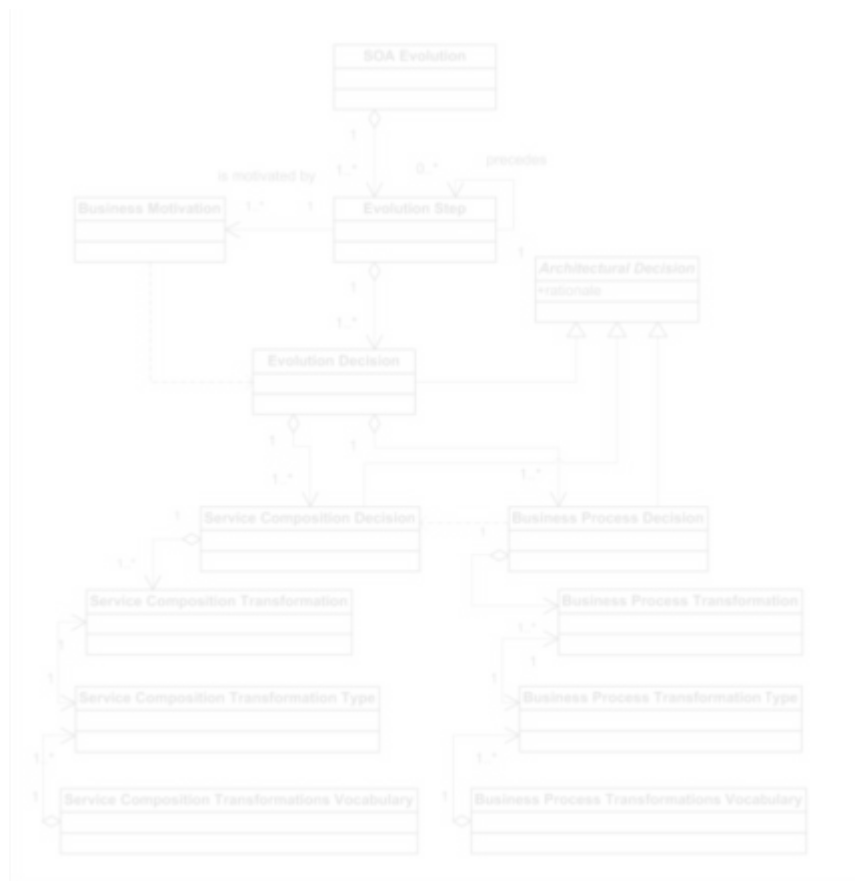


Рис.1.5. Модель еволюції SOA за Zalewski A., Szlenk M., Kijas S. [36]

13

Природно, існує багато способів, якими нові або змінені бізнес-потреби можуть впливати на бізнес-процеси, і те саме стосується впливу змін бізнес-процесів на склад послуг. Це своєрідне творче інженерне завдання, яке неможливо ні повністю формалізувати, ні автоматизувати.

Таким чином, в теперішній час еволюція системи SOA полягає в модифікації набору бізнес-процесів, наприклад, додаванні нових процесів, видаленні або зміні існуючих.

## 1.2. Мікросервісна архітектура

Подальшим розвитком SOA є архітектурний стиль мікросервісів (MSA) - це сучасний підхід до розробки програмного забезпечення, основним принципом якого є створення програмного проекту шляхом поділу його бізнес-компонентів на невеликі служби (сервіси), які можуть бути розгорнуті і працювати незалежно один від одного [25]. Кожен з сервісів працює в своєму власному процесі. Взаємодія між службами відбувається за допомогою чітко визначених інтерфейсів з використанням стандартних протоколів. Вони спілкуються один з одним за допомогою нейтральних до мови інтерфейсів прикладного програмування (API), таких як Representational State Transfer (REST). Мікросервіси також мають обмежений контекст. Їм не потрібно нічого знати про базову реалізацію чи архітектуру інших мікросервісів.

Варто зазначити, що мікросервіси, в деякому сенсі, є наступним кроком в еволюції сервіс-орієнтованої архітектури (SOA), тому що підтримують багато концепцій, що й SOA. Перш за все, обидва підходи володіють загальними рисами, що притаманні розподіленим архітектурам, вони пропонують значні переваги в порівнянні з монолітними і багаторівневими архітектурами завдяки кращій масштабованості, кращому поділу й контролю

над розробкою, тестуванням і розгортанням. Компоненти в цих архітектурах більш автономні, що спрощує обслуговування та контроль над змінами. Це, в свою чергу, призводить до створення більш надійних додатків.

Тим не менш, існують певні відмінності, які дозволяють виокремити мікросервіси як самостійний напрямок в розвитку архітектури розподілених систем. На думку Марка Річардса (Mark Richards), основна різниця між цими підходами полягає в тому, що мікросервісна архітектура побудована за принципом «якомога менше спільно використовуваних елементів», а SOA, навпаки, використовує принцип «як можна більше спільно використовуваних елементів», в якому основний акцент зроблений на абстрагуванні і повторному використанні бізнес-логіки [27, с. 22]. У MSA функціональні можливості служби мають тенденцію бути дуже малими, іноді реалізовані лише через один або два модулі; в SOA послуги, як правило, включають набагато більше бізнес-функціональних можливостей, іноді реалізованих як повні підсистеми [31].

Не дивлячись на те, що мікросервіси і SOA покладаються на «сервіс» як основний компонент архітектури, вони сильно розрізняються за характеристиками сервісів (служб). Концептуальні відмінності полягають у обсязі відповідальності, покладеної на окрему службу. У SOA служба може відповідати за обробку широкого спектру функцій і доменів даних, в той час як мікросервіс управляє одним доменом даних і одним набором відповідних функцій або однією функцією в цьому домені (рис.1.6).





Рис.1.6. Порівняння архітектур SOA та MSA

Управління даними в MSA відбувається децентралізовано [16]. Крім децентралізації рішень про концептуальні моделі, мікросервіси також децентралізують рішення про зберігання даних. У той час як монолітні програми віддають перевагу єдиній логічній базі даних для постійних даних. Мікросервісна архітектура передбачає, щоб кожен сервіс керує своєю власною базою даних, або різними екземплярами однієї й тієї ж технології баз даних, або абсолютно різними системами баз даних (підхід Polyglot Persistence).



Рис. 1.7. Технології застосування БД в монолітній та мікросервісній архітектурі [16]

Зазначені властивості надають переваги мікросервісній архітектурі при розробці високомасштабованих корпоративних проєктів. Цей підхід дозволяє вирішити деякі проблеми, що пов'язані з SOA, а також проблеми, виявлені з великими монолітними додатками [25, с. 9].

На думку дослідників [33, 37, 40,], застосування MSA є доцільним для обслуговування та обробки великих даних. Першим фактором успіху архітектури великих даних, які користуються мікросервісами, є підтримка цілісності даних. Ще одним фактором успіху є технологічна незалежність окремих компонентів [33]. Дослідники [40] для досягнення необхідної модульності та масштабування пропонують розгортання служб на багатьох стандартних апаратних серверах. Висока масштабованість досягається за рахунок усунення обмеження централізованої бази даних та використання натомість реплікованих сіток даних у пам'яті.

Результати опитування, проведеного IBM Market Development & Insights (2021) [20], свідчать, що користувачі бачать покращення від

застосування мікросервісної архітектури у корпоративних застосувань для свого бізнесу. Серед переваг, які вони відчували, були найважливіші: висока гнучкість для збільшення або зменшення ресурсів програми, покращений захист даних, швидкі темпи виходу на ринок та швидке реагування на зміни, які виникають під час розробки та провадження додатку тощо, невисокі ризики під час розгортання проекту тощо.

Але респонденти виокремили низку проблем, з якими вони стикнулися під час застосування цієї архітектури у бізнесі. Тим не менш, лише відносно невеликий відсоток ( $\leq 25\%$ ) респондентів (розробники, керівники розробників та ІТ-керівники) назвали будь-яку з них серйозною проблемою [20]. Відсутність належних навичок і знань стосовно мікросервісів, а також занадто багато застарілих систем і відсутність корпоративної підтримки становлять найпопулярніші причини, чому компанії не інтегрували мікросервіси у свої програми [21].

Таким чином, з урахуванням думки багатьох вчених [19, 21, 30], можемо сформулювати переваги мікросервісної архітектури в порівнянні з іншими архітектурними рішеннями.

По-перше, MSA вирішує проблему складності. Вона розкладає монолітну програму на набір послуг. Хоча загальний обсяг функціональних можливостей не змінився, додаток в випадку MSA розбито на керовані блоки або служби.

Кожна служба має чітко визначені межі у формі API, керованого викликом віддаленої процедури (RPC) або керованого повідомленням. Шаблон архітектури мікросервісів забезпечує рівень модульності, якого на практиці надзвичайно важко досягти за допомогою монолітної бази коду. Отже, окремі служби набагато швидше розвиваються, їх набагато легше зрозуміти та підтримувати.

По-друге, ця архітектура дозволяє кожній службі розроблятися незалежно командою, яка зосереджена на цій службі. Вони можуть вільно вибирати будь-які технології, які є доцільними, за умови, що служба виконує

договір API. Ця свобода означає, що розробники більше не зобов'язані використовувати, можливо, застарілі технології, які існували на початку нового проекту. При написанні нового сервісу вони мають можливість використовувати поточні технології. Крім того, оскільки сервіси відносно невеликі, стає більш доцільним переписати старий сервіс, використовуючи поточну технологію.

По-третє, шаблон архітектури мікросервісів дозволяє розгорнути кожен мікросервіс незалежно. Розробникам ніколи не потрібно координувати розгортання змін, які є локальними для їх служби. Такі зміни можна застосовувати одразу після перевірки. Шаблон архітектури мікросервісів робить можливим безперервне розгортання.

Крім того, шаблон архітектури мікросервісів дозволяє незалежно масштабувати кожну службу. Ви можете розгорнути лише ту кількість екземплярів кожної служби, яка задовольняє обмеження щодо її потужності та доступності. Крім того, ви можете використовувати апаратне забезпечення, яке найкраще відповідає вимогам служби до ресурсів. Наприклад, ви можете розгорнути службу обробки зображень із інтенсивним використанням ЦП на екземплярах EC2 Compute Optimized і розгорнути службу бази даних у пам'яті на екземплярах, оптимізованих для EC2 Memory [30].

### **1.3. Технології ефективної розробки сервіс-орієнтовних Java-застосувань (на прикладі мікросервісів)**

#### **1.3.1. Патерни проектування MSA**

Під час проектування мікросервісів треба враховувати певні закономірності та принципи, на яких будується ця архітектура. Відповідно до

19

[5, 12] можна узагальнити основні принципи, що використовуються для проектування мікросервісної архітектури:

- незалежні та автономні послуги;
- масштабованість;
- децентралізація;
- відмовостійкі послуги;
- ізоляція від збоїв;
- балансування навантаження у реальному часі;
- доступність;
- безперервна доставка завдяки інтеграції DevOps;
- повна інтеграція API та безперервний моніторинг;
- автоматична підготовка.

Вивчення загальних закономірностей у вирішенні проблем розробки додатків з MSA, призвело до появи патернів мікросервісної розробки (Microservices Patterns) або шаблонів проектування мікросервісів. Архітектурні шаблони та шаблони проектування зазвичай використовуються в життєвому циклі розробки програмного забезпечення (software development life cycle SDLC). Шаблони проектування програмного забезпечення — це загальні та багаторазові рішення для типових проблем у проектуванні програмного забезпечення в контексті проектування програмної системи [38]. Основна мета - надати перевірені часом рішення для таких завдань, як розробка мікросервісної архітектури, організація взаємодії мікросервісів один з одним, клієнтськими додатками, базами даних, забезпечення їх стійкості до відмов.

В наш час існує безліч шаблонів, пов'язаних із шаблоном мікросервісів. Найбільш повну класифікацію, на наш погляд, наведено в роботі С. Richardson [29] (додаток А). Автор виокремлює наступні групи Microservices Patterns:

- Патерни декомпозиції (*Decompose by business capability, Decompose by subdomain*);

20

- Патерн [Database per Service](#) описує, як кожна служба має власну базу даних, що забезпечує слабкий зв'язок;
- [API Gateway pattern](#) визначає, як клієнти отримують доступ до служб у мікросервісній архітектурі;
- Патерни [Client-side Discovery](#) та [Server-side Discovery](#) використовуються для маршрутизації запитів клієнта до доступного екземпляра служби в архітектурі мікрослужби;
- Шаблони обміну повідомленнями (*Messaging*) та віддаленого виклику процедур (*Remote Procedure Invocation patterns*) – це два різні способи взаємодії служб.
- Шаблони «Один сервіс на хост» (*Single Service per Host*) та «Кілька сервісів на хост» (*Multiple Services per Host*) – це дві різні стратегії розгортання.
- Шаблони наскрізної відповідальності: [Microservice chassis pattern](#) та [Externalized configuration](#)
- Шаблони тестування: [Service Component Test](#), [Service Integration Contract Test](#);
- Автоматичний вимикач ([Circuit Breaker](#));
- Маркер доступу ([Access Token](#));
- Патерни типу «Спостерігач»:
  - Агрегація журналів ([Log aggregation](#));
  - Метрики програми ([Application metrics](#));
  - Ведення журналу аудиту ([Audit logging](#));
  - Розподілене трасування ([Distributed tracing](#));
  - Відстеження винятків ([Exception tracking](#));
  - API перевірки працездатності ([Health check API](#));
  - Журнал розгортань та змін ([Log deployments and changes](#));
- Патерни інтерфейсу користувача (UI):

- Композиція фрагмента сторінки на стороні сервера ([Server-side page fragment composition](#));
- Композиція інтерфейсу користувача на стороні клієнта ([Client-side UI composition](#))

На наш погляд, є ще декілька важливих патернів проектування мікросервісів, які можна додати до попередньої системи класифікації. Це *патерни комунікації з мікросервісами*. До цих патернів віднесемо *Aggregator*, *chain* и *Branch* [26].

*Шаблон дизайну мікросервісу Aggregator*. Агрегатор може являти собою звичайну веб-сторінку або прикладну програму, яка викликає різні служби для досягнення корисності, необхідної програмі. Наприклад, агрегатор збирає весь запит із зовнішнього джерела, передає його відповідному мікросервісу та повертає результат.

*Chained Microservice Design Pattern*. Як впливає з назви, у цьому випадку створюється єдина консолідована відповідь на запит, а клієнт блокується, доки не завершиться повний ланцюжок запитів/відповідей. Тобто цей шаблон має єдину точку входу - сервіс А, та інші послуги залежать від сервісу А, і будуть викликані ланцюжком.

*Branch Microservice Design Pattern* розширює два попередніх шаблона та дозволяє синхронну обробку реакцій від абсолютно непов'язаних ланцюжків мікросервісів. Виходячи з потреб бізнесу, цей шаблон також можна використовувати для виклику різних ланцюжків або одного ланцюжка.

Аналіз досліджень [2] доводить, що не існує єдиного шаблону мікросервісів, який би був кращим за інші. Навпаки, кожен шаблон проектування працює краще в різних сценаріях.

### 1.3.2. Міжпроцесна комунікація в мікросервісах

Оскільки додатки на основі мікросервісів є розподіленими, однією з ключових проблем при розробці MSA-додатків є вибір механізму, за

22

допомогою якого служби зможуть спілкуватися один з одним. IPC (Interprocess communication) є однією з важливих проблем архітектури мікросервісів, яка не існує в монолітній системі. У монолітних системах компоненти можуть викликати один одного на рівні мови, тоді як у мікросервісах кожен компонент працює на власному процесі на, можливо, іншій машині, ніж інші сервіси, і саме тут IPC відіграє вирішальну роль у системі на основі мікросервісів. Вибір механізму IPC є критичним архітектурним рішенням, оскільки він може вплинути на продуктивність і доступність програми [32].

Існує кілька підходів до реалізації міжпроцесного зв'язку (IPC) у мікросервісах, і кожен має різні переваги та компроміси. Відповідно до [32], методи IPC були класифіковані на синхронні та асинхронні.

**Синхронний тип.** Ця форма спілкування часто розглядається як стиль взаємодії запит/відповідь. У цьому режимі один мікросервіс надсилає запит іншому сервісу та своєчасно чекає, поки той сервіс обробить результат і надішле відповідь. У цьому стилі типово, що запитувач блокує свою операцію, очікуючи відповіді від віддаленого сервера.

REST API на основі HTTP і gRPC є двома найпоширенішими типами синхронного зв'язку під час створення мікросервісів [28].

**REST** (Representational State Transfer) на основі HTTP - це архітектурний стиль, який зазвичай використовується для розробки інтерфейсів прикладного програмування (API) для сучасних веб-сервісів.

REST API є одним із найпоширеніших методів обміну даними між двома системами незалежно від архітектури програмного забезпечення. У цьому методі клієнтські програми використовують API через протокол HTTP для зв'язку з постачальником веб-служб. У системі, яка використовує REST API для зв'язку IPC, кожна служба зазвичай має свій власний веб-сервер, який працює на певному порту, наприклад 8080 або 443, і кожна служба надає набір кінцевих точок для забезпечення взаємодії з іншими мікросервісами та обмін інформацією між ними (рис. 1.8).

23





Рис.1.8. REST API безпосередньо прослуховує запити та відповідає на них [32]

В архітектурі мікросервісів API сервісу є контрактом між сервісами та їх споживачами. В REST API кожна кінцева точка має інтерфейс, який визначає набір операцій, які інші служби можуть викликати та отримувати відповідь. Кожен API служби має список операцій із назвою, обов'язковими параметрами, а також значеннями, що повертаються.

REST API в основному складається з двох компонентів: ресурсу та дієслова. Ресурс зазвичай представляє окремий бізнес-об'єкт або сукупність об'єктів, доступ до яких можна отримати за допомогою уніфікованого ідентифікатора ресурсу (URI), тоді як дієслово посилається на метод HTTP, за допомогою якого можна виконати операцію CRUD.

Наприклад, виклик HTTP GET до кінцевої точки `/customer/{customerId}` може повернути дані певного клієнта, тоді як виклик HTTP POST до `/customer/{customerId}` може використовуватися для оновлення інформації про цього конкретного клієнта.

**gRPC** - це сучасний протокол на основі віддаленого виклику процедури (RPC), розроблений і опублікований Google для розробки клієнта та сервера. RPC - це механізм, який використовується в багатьох розподілених програмах для полегшення взаємодії між процесами. RPC був вперше реалізований і розглядався як протокол, який забезпечує обмін повідомленнями між двома процесами з характеристиками низького перевантаження, простоти та прозорості [32]. За замовчуванням, коли клієнт надсилає запит на сервер, він зупиняє процес і чекає на повернення результатів. На відміну від REST API, який використовує HTTP/1.1 як транспортний протокол за замовчуванням,

gRPC працює через HTTP/2.0, що дає gRPC кілька переваг, пов'язаних із продуктивністю та безпекою [9].

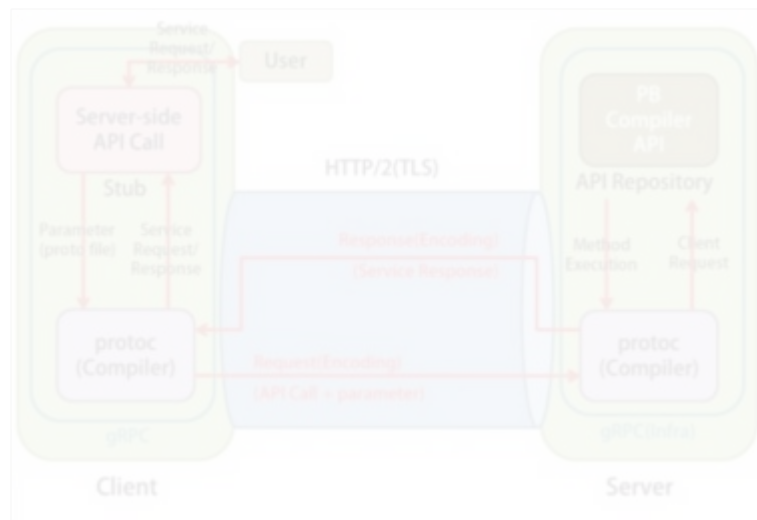


Рис.1.9. Процес роботи між клієнтом і сервером у gRPC [9]

Рисунок 1.9 демонструє життєвий цикл запиту/відповіді gRPC. Клієнт має локальний об'єкт, відомий як «заглушка», який реалізує методи для виклику сервера gRPC. Як тільки клієнт запускає метод заглушки з його необхідним параметром, запит буде надіслано на сервер. Отримавши запит, сервер декодує запит і починає обробку запиту. Нарешті, результат кодується сервером перед тим, як він надсилає назад клієнту[9].

**Асинхронна форма зв'язку** може бути реалізована в мікросервісах, коли служби обмінюються повідомленнями один з одним за допомогою шаблону обміну повідомленнями. У цій формі IPC мікросервіси мають брокера повідомлень, який діє як посередник між сервісами для координації запитів і відповідей [28].

Одна з фундаментальних відмінностей асинхронного зв'язку порівняно з синхронним режимом полягає в тому, що в асинхронному зв'язку клієнт більше не здійснює прямий виклик на сервер і не очікує негайної відповіді.

Замість цього клієнт публікує запит до брокера повідомлень. Одна чи декілька служб візьмуть запит від брокера й оброблять його далі, перш ніж повернути результат брокеру. Іншими словами, в асинхронній формі зв'язку взаємодія між мікросервісами здійснюється за допомогою служби-посередника, відомої як брокер повідомлень.

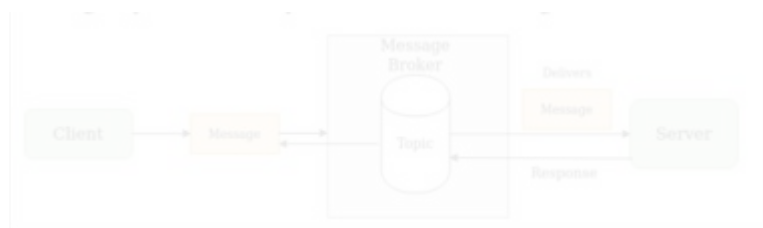


Рис.1.10. Асинхронне спілкування за допомогою шаблону Pub/Sub [32]

У цьому режимі, оскільки зв'язок є асинхронним, клієнт не блокує його роботу під час очікування відповіді.

Від вибору того чи іншого типу IPC залежить ефективність та продуктивність системи. Так, дослідження [32] доводять, що синхронна форма зв'язку може запропонувати вищу пропускну здатність, ніж асинхронний метод у ситуації, коли навантаження на систему відносно низька. І навпаки, експеримент [32] доводить значну різницю між синхронною та асинхронною формами зв'язку як у пропускній спроможності, так і в затримці, коли кількість паралельних запитів збільшується. В даному випадку асинхронний тип виявився більш ефективним, коли система перебуває під високим навантаженням.

Крім того, в роботі [32] доведене ефективність асинхронного типу взаємодії під час масштабування системи та збільшення пропускну здатності. Навпаки, у синхронній формі зв'язку, оскільки сервіси прив'язані один до одного, єдиним способом масштабування є збільшення потужності всіх залучених сервісів, що ускладнює масштабування та стає менш ефективним.

### 1.3.3. Фреймворки Java, що підтримують мікросервісну архітектуру

Фреймворк (framework) - програмна платформа, що визначає структуру програмної системи; програмне забезпечення, що полегшує розробку та об'єднання різних компонентів великого програмного проекту.

Існує кілька фреймворків для розробки архітектури мікросервісів із використанням мови програмування Java. Java-фреймворки для написання мікросервісів діляться на кілька категорій, від малих до великих:

**Microframeworks** - прості, з навмисно обмеженим набором можливостей, наприклад: Spark, Javalin, Micronaut та інші.

**MicroProfile** – open-source community specification для Enterprise Java Microservices. На даний час він включає 13 специфікацій (рис. 1.11). В даний час існують такі реалізації MicroProfile від різних постачальників: KumuluzEE, Open Liberty, Wildfly, Payara Micro, Helidon, Quarkus.

**Full Stack** – повнофункціональні, такі як Spring Boot, Dropwizard.



Рис.1.11. Специфікації MicroProfile [18]

Розглянемо деякі з них.

**Spring Boot.** Spring Boot є одним з найпопулярнішим фреймворкім розробки сервіс-орієнтованих Java-додатків. Він дає можливість створення на Java мікросервісів в архітектурному стилі REST (Representational State Transfer - передача репрезентативного стану). Spring boot легко інтегрується з іншими популярними фреймворками за допомогою інверсії керування.

Основні характеристики цього фреймворку [6, с. 35]:

- має вбудований веб-сервер (Tomcat, Jetty або Undertow)
- зумовлена конфігурація для швидкого початку роботи над проектом;
- автоматичне налаштування можливостей Spring (коли це **МОЖЛИВО**);
- широкий спектр можливостей, готових до використання у бізнес-проектах (таких як метрики, безпека, перевірка статусу, збереження конфігурації зовні тощо).

Використання Spring Boot дає наступні переваги під час розробки додатків на базі MSA:

- скорочує час розробки та збільшує ефективність та продуктивність роботи над проектом;
- пропонує вбудований HTTP-сервер для запуску веб додатків;
- дозволяє позбутися великої кількості шаблонного коду;
- спрощує інтеграцію з екосистемою Spring (включаючи Spring Data, Spring Security, Spring Cloud та ін);
- надає колекцію різних плагінів для розробки.

Spring Boot доцільно застосовувати при певних потребах [14]:

- корпоративні та перевірені в галузі мікросервіси на основі Java;
- необхідна готова інтеграція з багатьма сторонніми бібліотеками/залежностями.
- доступність розробників та підтримка спільноти є основними критеріями вибору.

**Dropwizard.** Фреймворк Dropwizard використовується для розробки зручних, високопродуктивних та Restful веб-сервісів. Без додаткових налаштувань підтримує інструменти конфігурації, метрики програми, протоколювання та роботи. Крім того, до складу фреймворка входять Jetty для HTTP, Jersey для REST, Jackson для JSON transformation, Hibernate для DB Access тощо.

Бібліотека метрик [Metrics](#) дає можливість отримати інформацію щодо поведінки коду у виробничому середовищі. Завдяки модулям для поширених бібліотек, таких як Jetty, Logback, Log4j, Apache HttpClient, Ehcache, JDBI, Jersey, та механізмам створення звітів, таким як Graphite, Metrics забезпечує повну видимість стека.

**Restlet.** Restlet Framework допомагає розробникам Java створювати кращі веб-API, які відповідають стилю архітектури REST. Повністю відкритий код, його можна безкоштовно завантажити та використовувати згідно з умовами ліцензії на програмне забезпечення Apache.

Завдяки потужним можливостям маршрутизації та фільтрації Restlet Framework, уніфікованому Java API клієнта та сервера розробники можуть створювати безпечні та масштабовані RESTful web API мікросервісної архітектури. Він надає досить потужні можливості маршрутизації та фільтрації, а також пропонує численні розширення. Фреймворк доступний для всіх основних платформ (Java SE/EE, Google AppEngine, OSGi, GWT, Android) і пропонує численні розширення, які відповідають потребам усіх розробників.

**Micronaut** - full-stack фреймворк на основі JVM для побудови модульних, легко тестованих мікросервісних та безсерверних додатків з багатомовною підтримкою Java, Kotlin і Groovy. Створює повнофункціональні мікросервіси, включаючи використання залежностей, автоконфігурацію, виявлення служб, маршрутизацію HTTP та клієнт HTTP. Micronaut прагне уникнути недоліків фреймворків Spring, Spring Boot, забезпечуючи швидший час запуску, зменшення обсягу пам'яті, мінімальне

29

використання рефлексії та юніт-тестування. Для ефективного створення мікросервісів до фреймворку вже включено HTTP-сервер Netty і контейнер залежностей, що створюється під час компіляції.

Microanaut доцільно застосовувати, якщо потрібно забезпечити [14]:

- модель реактивного програмування (з безсерверною підтримкою)
- ключовий критерій
- програми, керовані подіями/повідомленнями, програми командного рядка, HTTP-сервери та мікросервіси з прискореним запуском;
- легкі та мінімальні накладні витрати

**Helidon** - колекція бібліотек Java для написання мікросервісів. Helidon поставляється у двох варіантах і належить до двох категорій: Microframeworks та MicroProfile. Архітектура Helidon зображена на рис.1.12.

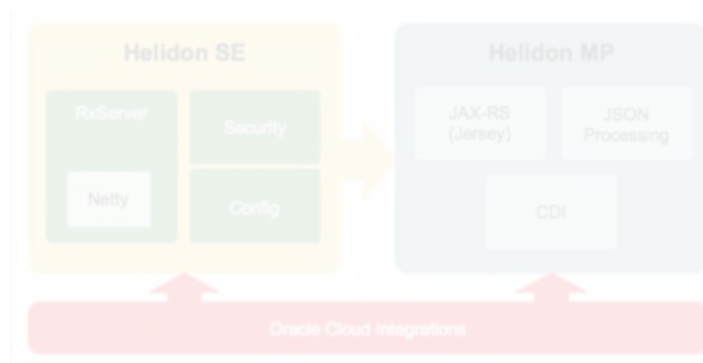


Рис. 1.12. Архітектура Helidon [22]

Helidon SE – простий, функціональний, легковажний мікрофреймворк, розроблений у сучасному reactive-стилі. Helidon SE пропонує три основні API для створення мікросервісу: веб-сервер, конфігурація та безпека для створення програм на основі мікросервісів. Сервер програм не потрібний.

Helidon MP – реалізація Eclipse Microprofile, що надає стиль розробки Java EE/Jakarta EE. Helidon MP підтримує специфікацію MicroProfile 1.1 для створення програм на основі мікросервісів. Простий у використанні, з

інструментальними можливостями, реактивним веб-сервером, стійкий до відмов.

Helidon підходить для наступних рішень [14]:

- архітектура, що підтримує як функціональну, так і реактивну функціональність: Helidon SE для мінімального досвіду розробки без анотацій та ін'єкцій залежностей, а Helidon MP для досвіду розробки мікропрофілів Jakarta EE.
- сучасна архітектура з хмарною підтримкою мікросервісів.

#### 1.3.4. Технології керування, масштабування та розгортання MSA-додатку

Автоматизація є ключовим аспектом процесу керування, масштабування та розгортання застосування на базі мікросервісної архітектури. Без неї практично неможливо досягти успіху у використанні парадигми мікросервісів.

На даний час існує багато технологій, спрямованих на вирішення задач ефективної розробки та розгортання мікросервісних додатків. Однією з популярних технологій є контейнеризація.

Контейнери пропонують віртуальне середовище, в яке упаковуються процес програми, метадані та файлова система, тобто все, що необхідне програмі для роботи. На відміну від віртуальних машин, контейнери не вимагають власної операційної системи — це лише обгортки навколо процесів, які безпосередньо взаємодіють з ядром.





Рис. 1.13. Порівняння архітектури, що працює на віртуальній машині та за допомогою контейнерів

Оскільки індивідуальні контейнери для додатків не мають всіх атрибутів повноцінної операційної системи, вони менші і легші за обсягом, ніж звичайні віртуальні машини. Завдяки цьому вони запускаються та відключаються швидше, і тим самим ідеально підходять для невеликих та легких мікросервісів.

Найпопулярніша підсистема контейнеризації – Docker. Docker [35] - це платформа для розробки, розгортання, запуску та спільного використання контейнерних програм. Він пропонує рішення, яке дозволяє відокремити програмне забезпечення від обчислювальної інфраструктури для швидшої доставки. Docker надає можливість об'єднувати залежності програми та виконувати їх в ізольованому середовищі, яке називається контейнером. Розділення дозволяє запускати багато контейнерів одночасно на даному хості безпечним способом.

Контейнери Docker легші, ніж віртуальні машини. Вони віртуалізують ОС хост-машини і зазвичай мають розмір лише десятки мегабіт. Навпаки, віртуальна машина займає десятки гігабіт, оскільки містить повну копію ОС, програми та її залежностей. Крім того, Docker дозволяє використовувати

модульний підхід для розробки додатків. Замість того, щоб запускати велику програму в одному контейнері, вона розділена між багатьма контейнерами відповідно до функціональності. Таким чином зміни можна застосувати до модуля без перебудови всієї програми (рис.1.14).

Як пояснюється на офіційному веб-сайті Docker [8], Docker базується на архітектурі клієнт-сервер. Docker-клієнт підключається до віддаленого Docker-демона або працює в тій же системі. Клієнт розмовляє з демоном Docker, який відповідає за керування контейнерами Docker. Обидва модуля обмінюються даними за допомогою REST API через сокети UNIX або мережеві інтерфейси.



Рис.1.14. Контейнеризація мікросервісів

На думку [34], контейнер Docker ідеально підходить для програм, розгорнутих у мікросервісній архітектурі, завдяки тому, що він дозволяє спільно використовувати операційну систему та бібліотеки підтримки, а

також тому, що контейнеризація є більш легкою, оперативною і масштабованою, ніж віртуалізація на основі гіпервізора.

Контейнери надають можливість переміщувати програми в різних середовищах. Вони гарантують, що програма працюватиме на будь-якій платформі однаково та використовуватиме переваги базового середовища. Кількість контейнерів може змінюватися, як тільки вона почне масштабуватися. В цей час виникає потреба в інструменті *оркестровки* для підтримки програми, керування контейнерами, організації розгортання оновлень і заміни несправних контейнерів.

Інструмент оркестрації контейнера забезпечує механізм для автоматизації розгортання контейнера, масштабування та роботи в мережі. Найпоширенішими прикладами цих інструментів є Kubernetes і Docker Swarm [35].

Kubernetes [13] — це інструмент оркестровки з відкритим кодом для розгортання, масштабування та керування контейнерними програмами. Kubernetes допомагає ефективно керувати групою контейнерів, що працюють на фізичних або віртуальних машинах. Kubernetes пропонує ряд різних переваг, перш за все в тому, що він поєднує в собі всі необхідні інструменти - оркестрацію, виявлення сервісів і балансування навантаження.

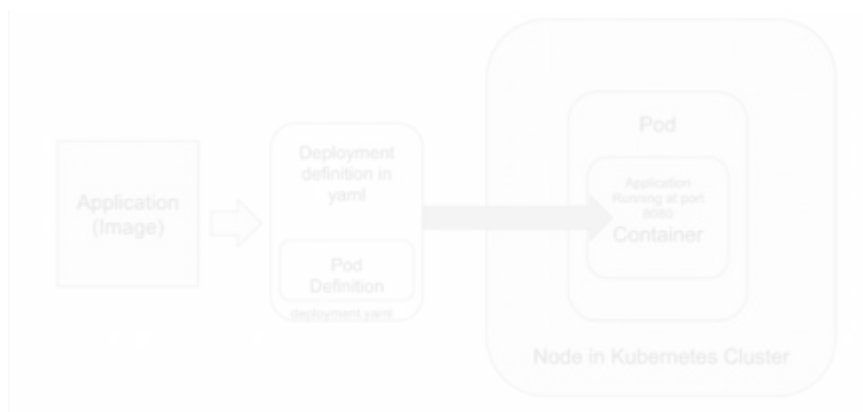


Рис.1.15. Розгортання додатку в кластері Kubernetes

Kubernetes надає наступне [13]:

*Моніторинг сервісів та розподіл навантаження.* Kubernetes може виявити контейнер, використовуючи ім'я DNS або власну IP-адресу. Якщо трафік у контейнері високий, Kubernetes може збалансувати навантаження та розподілити мережний трафік, щоб розгортання було стабільним.

*Оркестрація сховища.* Kubernetes дозволяє автоматично змонтувати систему зберігання за вибором, таку як локальне сховище, провайдери загальнодоступної хмари та багато іншого.

*Автоматичне розгортання та відкати.* Використовуючи Kubernetes, можна описати бажаний стан розгорнутих контейнерів і змінити фактичний стан на бажаний. Наприклад, можна автоматизувати Kubernetes на створення нових контейнерів для розгортання, видалення існуючих контейнерів та розподілу всіх їх ресурсів у новий контейнер.

*Автоматичне розподілення навантаження.* Користувач надає Kubernetes кластер вузлів, який він може використовувати для запуску контейнерних завдань. Користувач вказує Kubernetes, скільки ЦП та пам'яті (ОЗУ) потрібно кожному контейнеру. Kubernetes може розмістити контейнери на вузлах так, щоб найбільш ефективно використовувати ресурси.

**Самоконтроль.** Kubernetes перезапускає контейнери, що відмовили, замінює і завершує роботу контейнерів, які не проходять певну користувачем перевірку працездатності, і не показує їх клієнтам, поки вони не будуть готові до обслуговування.

*Керування конфіденційною інформацією та конфігурацією.* Kubernetes може зберігати та керувати конфіденційною інформацією, такою як паролі, OAuth-токени та ключі SSH. Користувач можете розгортати та оновлювати конфіденційну інформацію та конфігурацію програми без змін образів контейнерів та не розкриваючи конфіденційну інформацію у конфігурації стека.

Таким чином, застосування технологій контейнеризації та оркестровці в реалізації Docker та Kubernetes дають змогу зробити керування,

35

масштабування та розгортання MSA-додатку ефективним як з боку ресурсів,  
так і з боку процесів.

## Висновки до розділу 1

36

У першому розділі кваліфікаційної роботи було розглянуто сучасні підходи до розробки сервіс-орієнтованих застосувань. А саме, розглянуто еволюцію сервіс-орієнтованої архітектури. Було визначено наступні еволюційні підходи: багат шарова архітектура, багаторівнева архітектура, монолітна архітектура, сервіс-орієнтована архітектура, мікросервісна архітектура. В теперішній час еволюція системи SOA полягає в модифікації набору бізнес-процесів, наприклад, додаванні нових процесів, видаленні або зміні існуючих.

В дослідженні особлива увага приділена мікросервісній архітектурі, що є подальшим розвитком в еволюції сервіс-орієнтованої архітектури. Визначені спільні риси та відмінності SOA та MSA, а також переваги мікросервісної архітектури.

Окремим питанням розглянуто технології ефективної розробки сервіс-орієнтованих Java -застосувань (на прикладі мікросервісів). Досліджено класифікаційні підходи до сучасних патернів проектування мікросервісів, міжпроцесна комунікація в мікросервісах. Проаналізовані синхронна (REST API на основі HTTP і gRPC) та асинхронна (зв'язок за допомогою брокера повідомлень) форми зав'язків в MSA. Встановлено, що асинхронний тип виявився більш ефективним, коли система перебуває під високим навантаженням.

Також було розглянуто фреймворки Java, що підтримують мікросервісну архітектуру, та виокремлено переваги Spring Boot, як одного з найпопулярніших фреймворків з розробки сервіс-орієнтованих Java-додатків.

Як технологію автоматизації процесів керування, масштабування та розгортання MSA-додатку було розглянуто Docker та Kubernetes.

## РОЗДІЛ 2

### ПРОГРАМНА РЕАЛІЗАЦІЯ МІКРОСЕРВІСНОГО JAVA-ДОДАТКУ

37

## 2.1. Характеристика мікросервісного додатку

Додаток, що розробляється в кваліфікаційній роботі, спрямований на реалізацію основних концепцій мікросервісної архітектури, застосування сучасних патернів проектування мікросервісів на базі фреймворку Spring Boot.

Архітектура проекту подана на рис.2.1.



Рис.2.1. Архітектура мікросервісного додатку

На рис. 2.2. зображена структура мікросервісного додатку в середовищі розробки IntelliJ IDEA. Проект має наступні модулі.

*Project-microservice* – це контейнер, який містить всі модулі проекту. Назви модулів, що містяться в проекті, вказані в файлі settings.gradle (рис.2.3).

Рис.2.2. Структура Java-проекту

Рис.2.3. Файл settings.gradle проекту project-microservice

Модуль *EurekaServerApplication* – це реєстратор мікросервісів. Порт сервера – 8761.

Модуль *ApiGatewayApplication* – модуль шлюза, який виконує функції маршрутизації запитів до мікросервісів, єдиної точки входу до сервісів, балансувальника навантаження. Порт – 8765.

Проект складається з трьох мікросервісів, які є окремими модулями, кожен з яких виконує власні завдання:



- *EurekaClientApplication* – повертає ID певного інстанса мікросервісу, який було обрано балансувальником навантаження Load Balancer для опрацювання запиту;
- *EurekaClient2Application* – обробка запиту методом POST (база даних H2);
- *EurekaClient3Application* - конвертація випадкового набору json-даних в JPA-об'єкт (база даних PostgreSQL).

Мікросервіс *EurekaClientApplication* реалізує три інстанси для демонстрації роботи Load Balancer: *EurekaClientApplication-inst1*, *EurekaClientApplication-inst2*, *EurekaClientApplication-inst3*. (рис.2.4)

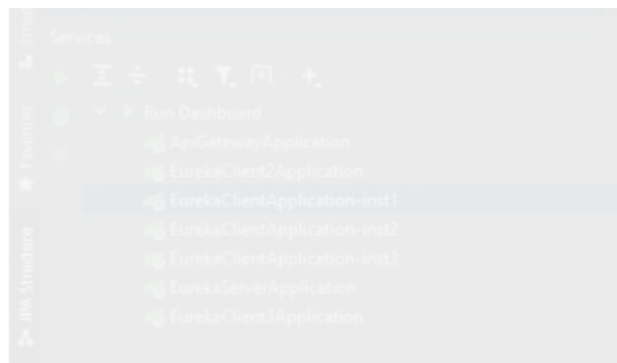


Рис.2.4. Служби проекту

Мікросервіси *EurekaClient2Application* та *EurekaClient3Application* в своїй реалізації застосовують СУБД реляційного типу (H2, PostgreSQL). З метою уніфікації та спрощення доступу до різних типів сховищ збереження (баз даних) застосовується специфікація Java Persistent API (JPA) (рис.2.5).

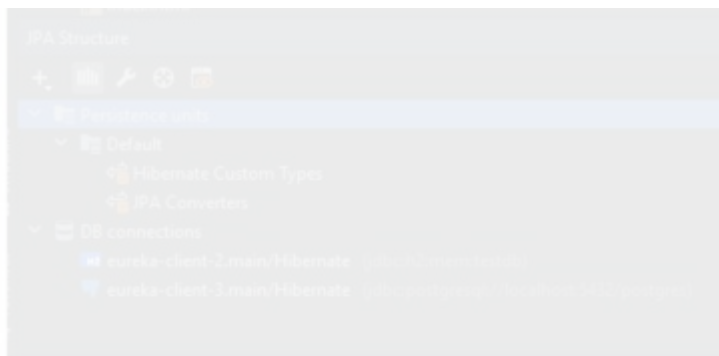


Рис.2.5. JPA Структура проекту

СУБД H2 – це легковажна реляційна база даних Java з відкритим кодом. Вона має вбудований та серверний режими; дискові чи оперативні бази даних. В даному випадку, база даних H2 налаштована для роботи як база даних пам'яті, що означає, що дані не будуть зберігатися на диску. H2 має консольний браузерний додаток. Офіційний сайт проекту: <https://www.h2database.com/html/main.html>

PostgreSQL - вільна об'єктно-реляційна система управління базами даних (СУБД). PostgreSQL підтримується всім основних операційних систем - Windows, Linux, MacOS. Офіційний сайт проекту: <https://www.postgresql.org/>.

Під час розробки застосовувалися наступні *патерни проектування*:

- *Service discovery* - патерн виявлення сервісів. Виявлення служб – один із ключових принципів архітектури на основі мікросервісів. Для реалізації цього патерну застосовується [Netflix Eureka](#). Netflix Eureka – це сервер та клієнт служби виявлення мікросервісів. Це сервісний реєстр. Він надає REST API для керування реєстрацією екземплярів служби та для запиту доступних екземплярів.
- *API Gateway* - один із основних патернів, навколо якого будується мікросервісна архітектура. API Gateway – це спосіб

поділу клієнтського програмного інтерфейсу від вашої внутрішньої реалізації. API Gateway приймає всі виклики API від клієнтів, а потім направляє їх у відповідний мікросервіс з маршрутизацією запитів, композицією та перетворенням протоколів. Зазвичай він обробляє запит, викликаючи кілька мікросервісів і об'єднуючи результати визначення кращого шляху. Тобто API Gateway забезпечує функціональність балансувальника навантаження Load Balancer. Балансування навантаження здійснює рівномірний розподіл мережевого трафіку, щоб уникнути збою через перевантаження ресурсів. Це також допомагає правильно та вчасно обробляти запити користувачів [39].

Для реалізації мікросервісного додатку застосовувалися наступні програмні інструменти:

- середовище розробки - IntelliJ IDEA 2021.2.3 (Ultimate Edition);
- мова програмування Java 16;
- збиральник проекту – Gradle;
- фреймворк Spring Boot, версія 2.7.6;

Для реалізації окремих модулів застосовувалися наступні зовнішні залежності:

- *EurekaServerApplication*: з набору бібліотек Spring Cloud версії 2021.0.5. - Eureka Server (залежність `springframework.cloud:spring-cloud-starter-netflix-eureka-server`);
- *ApiGatewayApplication* - Spring Cloud Gateway (залежність `org.springframework.cloud:spring-cloud-starter-gateway`), Spring Reactive Web – для асинхронного виконання запитів (залежність `org.springframework.cloud:spring-cloud-starter-gateway`);
- мікросервіси: з набору бібліотек Spring Cloud версії 2021.0.5. - Eureka Client (залежність `org.springframework.cloud:spring-cloud-starter-netflix-eureka-client`);

- *EurekaClient2Application* - Spring Data JPA (залежність `org.springframework.boot:spring-boot-starter-data-jpa`); H2 Database Engine (`runtimeOnly 'com.h2database:h2:2.1.214'`);
- *EurekaClient3Application* - Spring Data JPA (залежність `org.springframework.boot:spring-boot-starter-data-jpa`); БД PostgreSQL (`runtimeOnly 'org.postgresql:postgresql'`), бібліотека Jackson (залежність `group: 'com.fasterxml.jackson.core', name: 'jackson-core', version: '2.13.4'`), а також ресурс Mockaroo (<https://www.mockaroo.com/>)

## 2.2. Реалізація основних модулів проекту

### 2.2.1. Налаштування серверу Eureka

Eureka Server — це програма, яка містить інформацію про всі клієнтські сервісні програми. Кожен мікросервіс реєструється на сервері Eureka, і Eureka знає всі клієнтські програми, що працюють на кожному порту та IP-адресі.

Налаштування серверу Eureka відбувається в модулі *EurekaServerApplication*, який виступає в ролі реєстратора мікросервісів. Реалізація модуля передбачає налагодження файлу `application.properties`:

```
server.port=8761
spring.application.name=eserver
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
logging.level.com.netflix.eureka=OFF
logging.level.com.netflix.discovery=OFF
logging.pattern.console=%C{1.} [%-5level] %d{HH:mm:ss} - %msg%n
```

43

В цьому файлі вказано порт сервера, ім'я, вимикаємо налагодження eureka.client, шлях доступу мікросервісів до сервера, вимкнення деяких логів, встановлення формату виведення логів.

Налаштування файлу EurekaServerApplication.java передбачає встановлення анотацій. Анотація @EnableEurekaServer використовується для того, щоб програма Spring Boot діяла як сервер Eureka.

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {

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

### 2.2.2. Модулі мікросервісів

Проект складається з трьох мікросервісів:

1. Мікросервіс *EurekaClientApplication*. Цей модуль є клієнтським додатком Eureka. Файл application.properties для цього модуля має наступний вигляд:

```
server.port=0
spring.application.name=eclient
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
logging.pattern.console=%C{1.} [%-5level] %d{HH:mm:ss} - %msg%n
eureka.instance.instance-id=${spring.application.name}:${random.value}
```

Інструкція «server.port=0» дозволяє надавати автоматично вільні порти мікросервісу випадковим чином.

Також, треба вказати анотацію `@EnableEurekaClient` в файлі `EurekaClientApplication.java`. Анотація `@EnableEurekaClient` повідомляє платформі, що цей сервіс є екземпляром певного мікросервісу і просить зареєструвати його на сервері Eureka. Також Eureka Client запитує реєстр служб у Eureka Server, щоб визначити запущені екземпляри мікросервісів.

```
@SpringBootApplication
@EnableEurekaClient
public class EurekaClientApplication {

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

Файл `ControllerRest.java` є фактично Rest-сервісом. REST – це архітектурний стиль взаємодії додатків у мережі.

Контекст програми створить та зареєструє екземпляр класу, оскільки вказана інструкція `@RestController`. Крім цього інструкція підказує, що значення, що повертаються методами, є тілом відповідей на запити. Ця інструкція замінює собою пару `@Controller` + `@ResponseBody`. Анотація `@RequestMapping` вказує на маршрутизацію запитів.

Виклик метода `test()` буде відбуватися за допомогою HTTP-методу GET. Цей метод поверне ID певного екземпляра (інстанса) мікросервіса, який був обраний балансувальником навантаження API GateWay.

```
@RestController
@RequestMapping("/main")
public class ControllerRest {

    @Value("${eureka.instance.instance-id}")
```

45

```
private String id_eureka_client;
```

```
@GetMapping ("/test")
```

```
public String test() {
```

```
    return id_eureka_client;
```

```
}
```

```
}
```

2. Мікросервіс *EurekaClient2Application*. Цей модуль, як і попередній, є клієнтським додатком Eureka. Тому в головному класі *EurekaClient2Application* вказана відповідна анотація - `@EnableEurekaClient`.

Файл `application.properties` для цього модуля окрім налаштувань, які ідентичні з модулем *EurekaClientApplication*, має додаткові конструкції, що забезпечують роботу з консоллю бази даних H2 та JPA:

```
spring.h2.console.enabled=true
```

```
spring.h2.console.path=/h2console
```

Також файл містить налаштування роботи з драйвером базою даних H2, яка буде розміщуватися в оперативній пам'яті:

```
spring.datasource.url=jdbc:h2:mem:test
```

```
spring.datasource.driverClassName=org.h2.Driver
```

```
spring.datasource.username=sa
```

```
spring.datasource.password=sa
```

Останні конструкції файлу налаштовують роботу з JPA. Перші дві -забезпечують автоматичне створення/оновлення таблиці в базі даних, використовуючи нашу сутність:

```
spring.jpa.show-sql=true
```

```
spring.jpa.hibernate.ddl-auto=update
```

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
```

```
@Entity
```

```

@Table (name = "PERSONS")
public class Person {

    @Id
    @GeneratedValue
    @Column (name = "id", nullable = false)
    private Long id;

    @Column(name = "name")
    private String name;

    .....
}

```

Основне поняття Spring Data JPA - це репозиторій. Це інтерфейси, які використовують JPA Entity для взаємодії з базою даних для забезпечення основних операцій з пошуку, збереження, видалення даних (CRUD операції). Тому створено репозиторій PersonRep.java, який наслідує один з інтерфейсів Spring Data – CrudRepository:

```

@Repository
public interface PersonRep extends CrudRepository<Person, Long> {
}

```

Логіка додатку розміщена в контролері ControllerRest2.java:

```

@RestController
@RequestMapping (value = "/rest2", produces = "application/json")
public class ControllerRest2 {

    @Autowired
    private PersonRep personRep;

    // @RequestMapping("create-person")

```

47



```

@PostMapping("/create-person")
public Person createPerson (@RequestParam (value = "name",
required = true) String name)

{
    Person per=new Person();
    per.setName(name);
    return personRep.save(per);
}
.....
}

```

Анотація `@Autowired` відзначає об'єкт `personRep`, що вимагає автозаповнення ін'єкцією залежності Spring.

Запит до методу `createPerson()` здійснюється за допомогою POST-методу.

3. Мікросервіс *EurekaClient3Application*. Цей модуль є клієнтським додатком Eureka. Тому в головному класі *EurekaClient2Application*, аналогічно двом попереднім модулям, вказана відповідна анотація - `@EnableEurekaClient`.

Файл `application.properties` для цього модуля окрім налаштувань, які ідентичні з модулем *EurekaClientApplication*, має додаткові конструкції, що забезпечують роботу з консоллю бази даних PostgreSQL та JPA. Треба звернути увагу на те, що останні два мікросервіси працюють з різними базами даних, відповідно до концепції мікросервісної архітектури.

В наведених нижче конструкціях прописано драйвер для PostgreSQL, URL нашої бази даних з вказаним портом, встановлено логін та пароль для неї, прописано діалект для Hibernate.

```

Hibernate.spring.datasource.driverClassName =org.postgresql.Driver
spring.jpa.database=POSTGRESQL
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres

```

48

```

spring.datasource.username=postgres
spring.datasource.password=sa
spring.jpa.show-sql = false
spring.jpa.hibernate.ddl-auto = create
spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy
spring.jpa.properties.hibernate.id.new_generator_mappings=true
spring.jpa.properties.hibernate.temp.use_jdbc_metadata_defaults = false
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQL9Dialect

```

Згідно налаштувань, база даних буде ініціалізуватися за допомогою JPA, а точніше, за допомогою Hibernate, що є інструментом реалізації цієї специфікації. За допомогою конструкції *spring.jpa.hibernate.ddl-auto = create* встановлює явну ініціалізацію. Тобто після запуску додатку буде створена фізична таблиця база даних відповідно до розробленої моделі Entity.

Створена модель сутності бази даних Person.java, яка має наступний вигляд:

```

@Entity
@Table(schema="json", name = "person")
public class Person {

    @Id
    @GeneratedValue
    private Long pk;
    private long id;
    private String firstName;
    private String lastName;
    private String email;
    private String gender;
    private String country;
    private String ipAddress;
}

```

```

        public Person() {
        }
        ....
    }

```

Відповідно до цієї моделі буде створена фізична база даних postgres з таблицею person. Для подальшої роботи з БД в контексті JPA розроблено репозиторій, тобто клас-інтерфейс, який розширює CrudRepository:

```

@Repository
public interface PersonRepository extends CrudRepository<Person,
Long> {
}

```

Логіка додатку розміщена в контролері PersonController.java:

```

@RestController
@RequestMapping ( "/rest3")
public class PersonController {

    // Логер, що фіксує події у програмі та записує все,
    //що там відбувається.

    private    final    static    Logger    logger    =
        LoggerFactory.getLogger(PersonController.class);

    private PersonRepository personRepository;

    @Autowired
    public PersonController(PersonRepository personRepository) {
        this.personRepository = personRepository;
    }

    @RequestMapping("json")
    public void json() {

```

50

```

File jsonFile = null;
try {
    jsonFile = ResourceUtils.getFile("classpath:people.json");
} catch (FileNotFoundException e) {
    e.printStackTrace();
}

ObjectMapper objectMapper = new ObjectMapper();

try {
    List<Person> people = objectMapper.readValue(jsonFile, new
TypeReference<List<Person>>() {
    });

    personRepository.saveAll(people);
    logger.info("All records saved.");
} catch (IOException e) {
    e.printStackTrace();
}
}

```

Ключовим в даному файлі є метод `json()`. В цьому методі отримується дані у форматі `json` з файлу `people.json`, який був попередньо створений за допомогою ресурсу `Mockaroo` (сервіс, що створює випадкові дані у форматі `json`). За допомогою `ObjectMapper` з бібліотеки `Jackson`, `json`-дані перетворюються в `java`-об'єкт.

### 2.2.3. Налаштування маршрутизації мікросервісів

При створенні шлюзу ApiGatewayApplication основна робота відбувається в налагодженні файлу application.properties, який має наступний вигляд:

```
server.port=8765
spring.application.name=apigateway

eureka.client.service-url.defaultZone=http://localhost:8761/eureka
logging.pattern.console=%C{1.} [%-5level] %d{HH:mm:ss} - %msg%n
spring.cloud.gateway.discovery.locator.enabled=true
spring.cloud.gateway.discovery.locator.lower-case-service-id=true

spring.cloud.gateway.routes[0].id=test
spring.cloud.gateway.routes[0].uri=lb://eclient
spring.cloud.gateway.routes[0].predicates[0]=Path=/main/test
spring.cloud.gateway.routes[0].predicates[1]=Method=GET

spring.cloud.gateway.routes[1].id=create-person
spring.cloud.gateway.routes[1].uri=lb://eclient2
spring.cloud.gateway.routes[1].predicates[0]=Path=/rest2/create-person
spring.cloud.gateway.routes[1].predicates[1]=Method=POST

spring.cloud.gateway.routes[2].id=json
spring.cloud.gateway.routes[2].uri=lb://eclient3
spring.cloud.gateway.routes[2].predicates[0]=Path=/rest3/json
```

Файл містить налаштування роутингу запитів: у секції spring.cloud.gateway.routes ми задаємо конфігурацію маршруту до мікросервісу ресурсів.

### 2.3. Аналіз отриманих результатів

Розроблений Java-додаток реалізує концепцію мікросервісної архітектури, побудованої на основі патернів проектування Service Discovery та API GateWay, на базі фреймворку Spring Boot.

Проаналізуємо отримані результати. Для більш легкого тестування мікросервісного додатку розроблено клієнтську веб-сторінку (рис.2.6.)



Рис.2.6. Веб-сторінка тестування мікросервісного додатку

Для реалізації концепції Service Discovery, патерну виявлення та реєстрації мікросервісів, в додатку застосовано стек Netflix Eureka з екосистеми Spring Cloud. Запустимо всі модулі нашого додатку. Перейдемо по посиланню <http://localhost:8761> в дашборд Eureka Server (рис. 2.7).



Рис. 2.7. Дашборд Eureka Server

Отримані результати демонструють екземпляри, наразі зареєстровані в Eureka: ми бачимо по одному інстансу мікросервісів EurekaClient2Application (eclient2) та EurekaClient3Application (eclient3), три інстанси мікросервісу EurekaClientApplication (eclient), модуль ApiGatewayApplication (apigateway) на порту 8765. Таким чином, ми бачмо, що всі екземпляри запущених модулів зареєстровані.

Проаналізуємо результати роботи мікросервісу EurekaClientApplication. Під час роботи додатку створено та запущено три екземпляри мікросервісу. Rest-сервіс повинен повертати ID певного інстанса мікросервіса, який було обрано балансувальником навантаження Load Balancer для опрацювання запиту. Всі три інстанси працюють на різних портах, які було визначено випадковим чином (серед доступних портів). Але завдяки API GateWay можемо звертатися до мікросервісу за посиланням <http://localhost:8765/main/test>. Load Balancer самостійно обирає певний інстанс мікросервісу відповідно навантаженню для посилання запиту (рис.2.8).

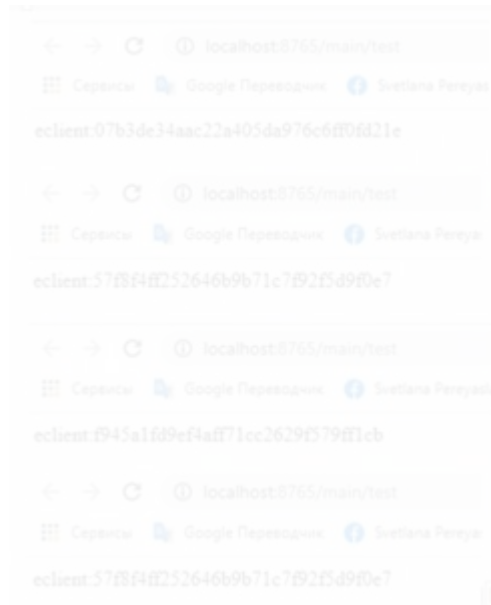


Рис.2.8. Приклад тестування мікросервісу EurekaClientApplication

Як видно з наведеного зображення, на запит повертається відповідний ID певного інстансу, при цьому, його значення змінюється. Це демонструє роботу Load Balancer на базі API Gateway.

Проаналізуємо результати роботи мікросервісу EurekaClient2Application. Його завдання - обробка запиту методом POST (база даних H2). Тестувати роботу мікросервісу будемо за допомогою консолі бази даних H2 (рис.2.9.)



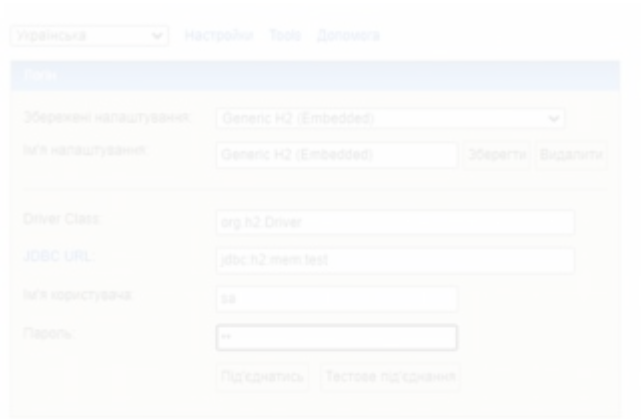


Рис.2.9. Консоль бази даних H2

Перед початком відправлення запиту перевіримо вміст таблиці persons (рис.2.10).

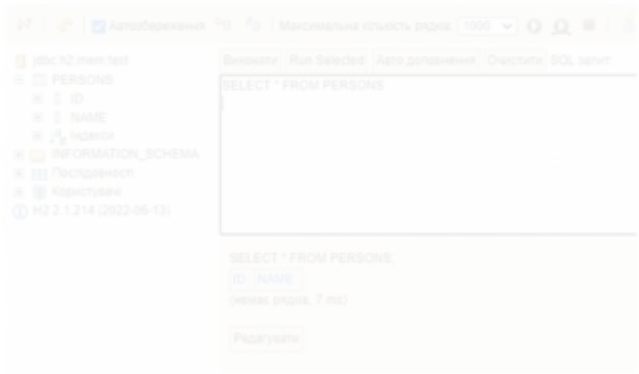


Рис.2.10. Початковий стан таблиці persons

Сформуємо запит методом POST за допомогою форми веб-сторінки (рис.2.11)



Рис.2.11. Форма для створення POST-запиту

Результатом виконання запиту є дані у форматі json (рис.2.12)

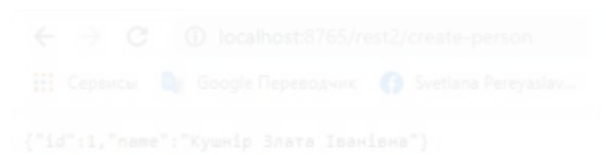


Рис.2.12. Результат виконання POST-запиту

В консолі бази даних H2 також можна подивитися отриманий результат (рис. 2.13). Таким чином, ми бачимо, що мікросервіс виконує поставлені завдання.



Рис.2.13. Консоль H2 з сформованим записом в таблиці persons

Проаналізуємо результати роботи мікросервісу EurekaClient3Application, завданням якого є конвертація випадкового набору json-даних в JPA-об'єкт (база даних PostgreSQL). В-першу чергу створимо файл даних в форматі json за допомогою ресурсу Moskaoo (рис. 2.14)



Рис.2.14. Створення json-файлу за допомогою ресурсу Mockaroo

В результаті ми отримали файл people.json, який містить 1000 записів заданої структури, сформованих випадковим чином.

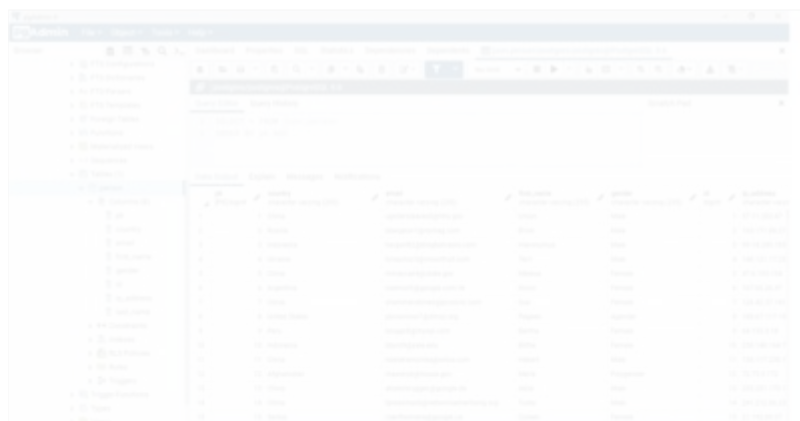
Наступний шаг – встановлення та налагодження бази даних PostgreSQL. Як результат – ми отримаємо можливість скористатися панеллю адміністратора PostgreSQL (рис.2.15).



Рис.2.15. Панель адміністратора PostgreSQL

Під час направлення запиту `http://localhost:8765/rest3/json` методу `json()`, де описується основна логіка з конвертації випадкового набору json-даних в JPA-об'єкт (база даних PostgreSQL), відбувається створення таблиці `person`,

структура якої повністю співпадає з моделлю, яка була задана в Entity-класі Person, а також заповнення цієї таблиці даними, які були конвертовані з файлу people.json (рис.2.16).



id	name	email	phone	address	date_of_birth
1	John	john.doe@example.com	+380123456789	123 Main St, Kyiv	1990-01-01
2	Jane	jane.doe@example.com	+380123456789	123 Main St, Kyiv	1990-01-01
3	John	john.doe@example.com	+380123456789	123 Main St, Kyiv	1990-01-01
4	Jane	jane.doe@example.com	+380123456789	123 Main St, Kyiv	1990-01-01
5	John	john.doe@example.com	+380123456789	123 Main St, Kyiv	1990-01-01
6	Jane	jane.doe@example.com	+380123456789	123 Main St, Kyiv	1990-01-01
7	John	john.doe@example.com	+380123456789	123 Main St, Kyiv	1990-01-01
8	Jane	jane.doe@example.com	+380123456789	123 Main St, Kyiv	1990-01-01
9	John	john.doe@example.com	+380123456789	123 Main St, Kyiv	1990-01-01
10	Jane	jane.doe@example.com	+380123456789	123 Main St, Kyiv	1990-01-01

Рис.2.16. Структура та вміст таблиці person

Вдале виконання запиту підтверджується повідомленням, яке ми отримуємо від логера проекту (рис.2.17).

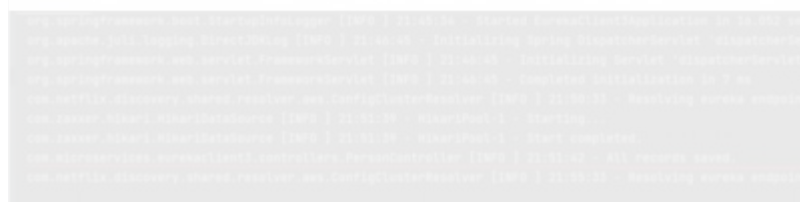


Рис.2.17. Повідомлення про конвертацію та збереження всіх записів

Таким чином, ми можемо зробити висновок. Що розроблений проект, який підтримує мікросервісну архітектуру, відповідає всім заявленим вимогам та повністю реалізує концепцію, яка була розроблена на етапі проектування. Мікросервіси, які входять до складу проекту, виконують поставлені завдання. В проекті реалізовані функції реєстрації та маршрутизації мікросервісів та розподілення навантаження між ними.

## Висновки до розділу 2

Під час виконання кваліфікаційної роботи розроблено Java-додаток, спрямований на реалізацію основних концепцій мікросервісної архітектури, застосування сучасних патернів проектування мікросервісів Service Discovery та API GateWay на базі фреймворку Spring Boot.

А саме, розроблена архітектура проекту який складається з декілька модулів: модуля реєстратору мікросервісів, модуля шлюза, який виконує функції маршрутизації запитів до мікросервісів, єдиної точки входу до сервісів, балансувальника навантаження та трьох мікросервісів.

Для реалізації розробленої архітектури застосовувалися бібліотеки екосистеми Spring Cloud: Spring Cloud Gateway та Eureka (Server, Client). Для реалізації специфікації Java Persistence API застосовувалося Spring Data JPA.

Таким чином, ми можемо зробити висновок, що розроблений проект, відповідає всім заявленим вимогам та повністю реалізує концепцію, яка була розроблена на етапі проектування. Мікросервіси, які входять до складу проекту, виконують поставлені завдання.

Подальший розвиток дослідження мікросервісів, як одного з напрямків сервіс-орієнтованої архітектури, передбачає вивчення технологій міграції в хмарні платформи для збільшення ефективності роботи та зменшення навантаження на фізичні ресурси.

## ВИСНОВКИ

У кваліфікаційній роботі проведено дослідження сучасних підходів та технологій розробки сервіс-орієнтованих застосувань на платформі Java. В першому розділі було розглянуто теоретичні підходи та еволюцію сервіс-орієнтованої архітектури. Встановлено, що за еволюційний період було багато архітектурних підходів, які застосовувалися до появи SOA. Але і в теперішній час еволюція системи SOA має місце, й полягає в модифікації набору бізнес-процесів, наприклад, додаванні нових процесів, видаленні або зміні існуючих.

В дослідженні особлива увага приділена мікросервісній архітектурі, що є подальшим розвитком в еволюції сервіс-орієнтованої архітектури. Визначені спільні риси та відмінності SOA та MSA, а також переваги мікросервісної архітектури, розглянуто технології ефективного розробки сервіс-орієнтованих Java -застосувань на прикладі мікросервісів (патерни проектування, синхронна та асинхронна форми зав'язків в MSA). Встановлено, що асинхронний тип виявився більш ефективним, коли система перебуває під високим навантаженням.

Також було розглянуто фреймворки Java, що підтримують мікросервісну архітектуру, та виокремлено переваги Spring Boot, як одного з найпопулярніших фреймворків з розробки сервіс-орієнтованих Java-додатків.

Під час виконання другого розділу кваліфікаційної роботи розроблено Java-додаток, спрямований на реалізацію основних концепцій мікросервісної архітектури, застосування сучасних патернів проектування мікросервісів Service Discovery та API GateWay на базі фреймворку Spring Boot. Для реалізації розробленої архітектури застосовувалися бібліотеки екосистеми Spring Cloud: Spring Cloud Gateway та Eureka (Server, Client). Для реалізації специфікації Java Persistence API застосовувалося Spring Data JPA.

Розроблений проект відповідає всім заявленим вимогам та повністю реалізує концепцію, яка була розроблена на етапі проектування. Мікросервіси, які входять до складу проекту, виконують поставлені завдання.

Подальший розвиток дослідження мікросервісів, як одного з напрямків сервіс-орієнтованої архітектури, передбачає вивчення технологій міграції в хмарні платформи для збільшення ефективності роботи та зменшення навантаження на фізичні ресурси.

## Matches

Internet sources

37

1	<a href="https://arabicprogrammer.com/article/23373294833">https://arabicprogrammer.com/article/23373294833</a>	10 Sources	0.52%
2	<a href="https://juejin.cn/post/6844903757721894920">https://juejin.cn/post/6844903757721894920</a>	4 Sources	0.47%
3	<a href="https://www.paigeniedringhaus.com/blog/how-to-use-netflixs-eureka-and-spring-cloud-for-service-registry">https://www.paigeniedringhaus.com/blog/how-to-use-netflixs-eureka-and-spring-cloud-for-service-registry</a>		0.4%
4	<a href="https://www.cnblogs.com/idoljames/p/11623388.html">https://www.cnblogs.com/idoljames/p/11623388.html</a>	2 Sources	0.35%
5	<a href="https://stackoverflow.com/questions/53078306/populate-a-database-with-testcontainers-in-a-springboot-integration-test">https://stackoverflow.com/questions/53078306/populate-a-database-with-testcontainers-in-a-springboot-integration-test</a>		0.34%
6	<a href="http://yoonbumtae.com/?p=2555">http://yoonbumtae.com/?p=2555</a>		0.32%
7	<a href="https://pastebin.com/yMNF1yL">https://pastebin.com/yMNF1yL</a>	2 Sources	0.32%
8	<a href="https://stackoverflow.com/questions/47823564/postgresql-failure-in-spring-boot">https://stackoverflow.com/questions/47823564/postgresql-failure-in-spring-boot</a>		0.31%
9	<a href="https://iditect.com/article/spring-mysql-dialectspring-boot-using-mysql-and-jpa-in-spring-boot.html">https://iditect.com/article/spring-mysql-dialectspring-boot-using-mysql-and-jpa-in-spring-boot.html</a>	2 Sources	0.27%
10	<a href="https://ukrfire.com/what-is-kubernetes">https://ukrfire.com/what-is-kubernetes</a>		0.24%
11	<a href="https://www.itdaan.com/blog/2018/04/04/6addc7ea8d73d58018e2d23261a85185.html">https://www.itdaan.com/blog/2018/04/04/6addc7ea8d73d58018e2d23261a85185.html</a>	2 Sources	0.21%
12	<a href="https://www.geeksforgeeks.org/spring-hibernate-configuration-and-create-a-table-in-database">https://www.geeksforgeeks.org/spring-hibernate-configuration-and-create-a-table-in-database</a>		0.21%
13	<a href="https://www.appsdeveloperblog.com/spring-cloud-api-gateway-tutorial">https://www.appsdeveloperblog.com/spring-cloud-api-gateway-tutorial</a>		0.17%
14	<a href="https://developer.aliyun.com/article/763114">https://developer.aliyun.com/article/763114</a>	2 Sources	0.13%
15	<a href="https://www.appsdeveloperblog.com/spring-cloud-api-gateway-automatic-mapping-of-routes">https://www.appsdeveloperblog.com/spring-cloud-api-gateway-automatic-mapping-of-routes</a>		0.13%
16	<a href="https://qna.habr.com/q/934569">https://qna.habr.com/q/934569</a>		0.12%
17	<a href="https://hevodata.com/learn/java-rest-api">https://hevodata.com/learn/java-rest-api</a>		0.12%
18	<a href="https://www.logicbig.com/tutorials/spring-framework/spring-orm/spring-jpa-dao.html">https://www.logicbig.com/tutorials/spring-framework/spring-orm/spring-jpa-dao.html</a>		0.12%
19	<a href="https://zhuanlan.zhihu.com/p/126368941">https://zhuanlan.zhihu.com/p/126368941</a>	2 Sources	0.11%



## Quotes

Quotes

2

1 «не є незалежними від програми, до якої вони належать»

2 «рішення щодо бізнес-процесів»

## Exclusions

## Internet exclusions

21

[illegible]