ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики
(повне найменування назва факультету)

Кафедра інформаційних систем
(повна назва кафедри)

КУРСОВА РОБОТА

на тему:

Розробка мікросервісної архітектури з допомогою технологій Java

Студентки 3 курсу, групи ПМІ-34,
напряму підготовки Комп'ютерні науки
Ковальчук С. А.
(прізвище та ініціали)

Керівник асист. каф. ІС, канд. фіз.-мат. наук
Стельмащук В. В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна шкала _____
Кількість балів: _____Оцінка: ECTS _____

Львів – 2020

Зміст

ПЕРЕЛІК СКОРОЧЕНЬ

SOA	Service-Oriented Architecture (сервіс-орієнтована архітектура)
MSA	MicroService Architecture (мікросервісна архітектура)
REST	Representational State Transfer, архітектурний стиль
ПЗ	Програмне забезпечення
RPC	Remote Procedure Call, протокол для взаємодії між комп'ютерами
БД	База даних
API	Application Programming Interface, набір інтерфейсів для взаємодії різнотипного програмного забезпечення
HTTP	HyperText Transfer Protocol, протокол передачі даних, на якому побудована мережа Інтернет
HTTPS	HyperText Transfer Protocol Secure, захищений HTTP
ІС	Інформаційна система
SOAP	Simple Object Access Protocol, протокол обміну структурованими повідомленнями в розподілених системах
CI	Continuous Integration
CD	Continuous Delivery
SAML	Security Assertation Markup Language, мова розмітки, відкритий стандарт обміну даними аутентифікації та авторизації
XML	eXtensible Markup Language, розширювана мова розмітки

ВСТУП

Проблема проєктування та створення якісного програмного забезпечення є надзвичайно важливою у сучасному інформаційному світі. З розвитком IT-індустрії було знайдено багато різних підходів та концепцій до побудови складних програмних систем. Показником гарно побудованої програми є, звісно, її архітектура, яка правильно описує предметну область та є формальною моделлю системи. Архітектурою можна вважати набір певних структурних компонентів зв’язаних між собою, які задають поведінку всієї системи. Основною задачею архітектури є управління складністю, елегантне та доцільне відображення предметної області. Довгий час провідне місце займала так звана “монолітна архітектура”. При даному підході вся система являє собою моноліт, який фізично розташовується на єдиній машині, запускається в одному процесі та виконує всі бізнес-операції системи.

Монолітний додаток піддається лише горизонтальному масштабуванню шляхом запуску декількох окремих серверів із кожним окремим монолітом. Але з плином часу знаходилися інші ідеї та підходи, саме таким стала сервіс-орієнтована архітектура (далі SOA), на відміну від монолітної системи, при SOA вся програма являє собою розподілену систему, яка обмінюється повідомленнями за певним протоколом. Вся система складається з набору незалежних сервісів, які фокусуються на власній задачі. SOA націлена на боротьбу з великими монолітними системами. Сама по собі ідея SOA чудова, але питання як правильно та якісно організовувати сервіс-орієнтовану архітектуру залишається відкритим. Основні вузькі місця відносяться до протоколів обміну даними, таких як SOAP, а також неправильні місця розділу системи.

Пізніше було запропоновано новий підхід до організації SOA, так звана мікросервісна архітектура (далі MSA). Мікросервісну архітектуру можна вважати підмножиною SOA, але все ж таки MSA відрізняється від класичного SOA. Основна відмінність — це невелика кількість кодової бази на кожен сервіс, в той час як в SOA не важливий об’єм кодової бази. Також важливим місцем для MSA є

те, що кожен сервіс має мати власний обмежений контекст для цієї предметної області.

Обмежень на кількість існуючих сервісів немає, але кожен сервіс має працювати лише над одною бізнес-задачею. Для обміну інформацією мікросервіси використовують стандартизовані протоколи передачі даних (наприклад, HTTP), як правило кожен сервіс має своє API для спілкування з іншими мікросервісами. Всі сервіси можуть бути написані на абсолютно різних мовах програмування та використовуючи будь-які бібліотеки, також має місце децентралізоване збереження даних, тобто кожен сервіс має свою власну базу даних. Можна виділити основні переваги використання мікросервісної архітектури:

- Низька зв’язність між основними компонентами системи та висока зчепленність коду в окремо взятому сервісі
- Відносно просте розгортання, кожен сервіс розгортається незалежно від інших, на відміну від монолітного додатку, де вся система запускаться в єдиному процесі та динамічно вносити зміни не є можливим.
- Просте масштабування системи. Ми можемо запускати будь-яку кількість сервісів на окремих серверах
- Відмовостійкість. При виведенні з ладу одного сервісу вся програма ще може вірно працювати
- Застосування різних мов програмування та технологій
- Для створення мікросервісів можна використовувати компактні групи розробників, які є відповідальними за власний сервіс.

До мінусів мікросервісної архітектури можна віднести:

- Відносна складність розробки, прямо пропорційно залежить від кількості обраних мов програмування та фреймворків.
- Витрачаються додаткові ресурси на пересилання повідомлень між сервісами та на їх серіалізацію та десеріалізацію
- Проблеми з версіонуванням
- Відносно складне інтеграційне тестування.

Мікросервісну архітектуру доцільно використовувати, якщо всю систему планується розготати в хмарі, оскільки монолітний додаток не можна зручно масшабувати горизонтально.

В цілому, мікросервісна архітектура не є “срібною кулею”, а вирішує лише певний набір задач і залежить від різних обставин. Тому в даній роботі буде розглянуто, які обмеження висуваються до MSA та доцільність використання цієї архітектури.

Метою даної роботи є побудова додатку мовою програмування Java на базі мікросервісів, доцільність використання цієї мови у порівнянні з іншими. Для більш детального ознайомлення слід також розробити тестову програму для демонстрації роботи мікросервісної системи.

1. ПРОБЛЕМИ ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Основні складнощі проєктування

Проєктування архітектури інформаційної системи (далі ІС) є однією із найважливіших етапів створення проєкту. Перш за все, визначимо, що мається на увазі під поняттям “архітектура інформаційної системи”. Існує безліч трактувань даного терміну, однак можемо описати архітектуру ІС, як організацію системи через набір компонент, їх взаємовідносини та зв’язок зі зовнішнім середовищем [1]. Під час проєктування системи, як правило, приймаються відповідні проєктні рішення, після прийняття яких зміна поведінки ІС стає надзвичайно складною.

Основною ціллю програмної архітектури є боротьба з потенційною складністю, яка властива складним програмним системам, корпоративним додаткам. Вдало розроблена архітектура заощадить велику кількість часу та зусиль. Гарно спроєктована архітектурна модель повинна бути гнучкою в місцях, які потенційно мають найчастіше змінюватися чи розширюватись, але статичною в інших. Також правильно побудовану програму легко супроводжувати, тестувати та підтримувати. Можна виділити ключові моменти, які описують правильно побудовану архітектуру [2]:

- Ефективність та працездатність системи. Перш за все, ІС має вирішувати поставлені перед нею функціональні вимоги, при будь-яких обставинах.
- Гнучкість системи. Показник правильно створеної архітектури — здатність швидко та зручно змінювати систему, тому при аналізі предметної області та проєктуванні моделі завжди необхідно оцінювати та знаходити місця, які потенційно будуть змінюватися, щоб в майбутньому не витрачати додатковий час на зміну підсистем, якщо це буде, взагалі, можливим.
- Розширюваність системи. Іншим важливим показником є здатність додавати нові сутності та функції, не змінюючи та не порушуючи її загальної структури та поведінки, причому щоб на додавання нових функцій витрачалось найменш можлива кількість часу.

- Продуктивність. Програма повинна витримувати належне навантаження зі сторони користувачів та відповідати за допустимий проміжок часу.
- Здатність до тестування. Добре протестований код не тільки буде мінімізувати кількість помилок, а буде показником добре сформованої системи, оскільки це означитиме, що наявна низька зв’язність.
- Повторне використання. Систему бажано проєктувати так, щоб її деякі складові можна було використовувати в інших ІС.

Основними показниками неправильно сформованої архітектури є:

- Жорсткість. Дану програму тяжко модифікувати, при зміні одного компоненту, змінюються інші частини системи.
- Крихкість. При внесенні чи зміні нових елементів, інші частини системи виходять з ладу.
- Висока зв’язність. Створену програму важко протестувати, оскільки всі компоненти сильно зв’язані між собою.

1.2 Багатошарова архітектура та SOA

Існують два основних підходи до створення складних інформаційних систем: монолітна багатошарова архітектура (як правило, тришарова) та розподілена сервіс-орієнтована архітектура. Кожен з наведених підходів має свої як переваги, так і недоліки.

Концепція багатошарової моделі є давно відомою та найпопулярнішою на даний час, вона базується на розподілі всієї системи на окремі ключові функціональні частини. Розглянемо класичну тришарову архітектуру, як найпопулярніший приклад багатошарової архітектури, вона включає шар доступу до даних, шар бізнес правил та шар представлення. На рисунку 1.1 зображено основні частини тришарової архітектури.

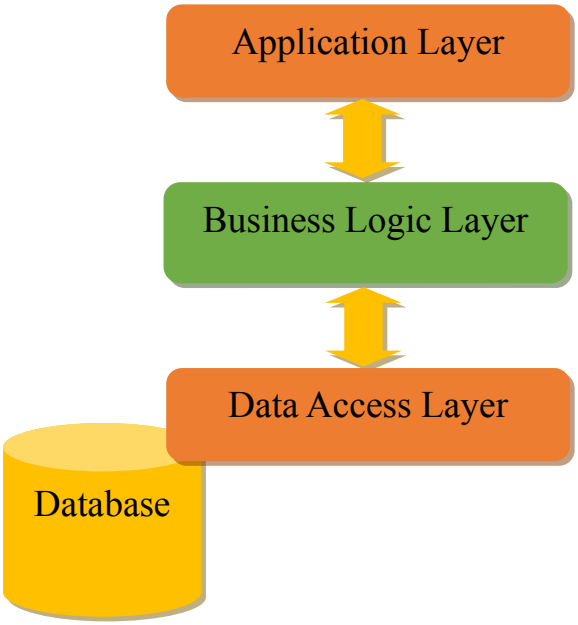

Рисунок 1.1 — Приклад класичної тришарової архітектури

Опишемо основні функції кожного зі трьох модулів. В основі всієї системи лежить шар доступу до даних, який надає зручний інтерфейс для доступу до джерел даних для вищих прикладних рівнів програми.

Наступним модулем є шар бізнес-логіки. Даний рівень містить набір алгоритмів та бізнес-функцій, які задають та описують предметну область, даний шар має бути найбільш гнучким, тому що саме до нього в майбутньому буде застосовуватися найбільше змін. Також використовує API рівня доступу до даних, щоб реалізовувати прикладні методи. Найвище місце займає рівень представлення, який взаємодіє безпосередньо зі користувачем та надає вхідні дані для всієї системи. Іншим підходом по створенню високонавантажених систем є сервіс-орієнтована архітектура (далі SOA). Вона представляє ідею, що програма має складатися з набору сервісів, які взаємодіють один з одним через стандартизовані протоколи та інтерфейси. В свою чергу до сервісів висуваються наступні вимоги [3]:

1. Стандартизовані інтерфейси.
2. Слабка зв’язність.
3. Абстракція
4. Повторне використання
5. Автономність

6. Відсутність стану

Сервіс можна уявляти як окремий функціональний модуль, який може знаходитися на іншому віддаленому комп'ютері, взаємодія між даними модулями відбувається через мережу. В найпростішому випадку існує три основних елементи: постачальник сервісу, споживач сервісу та реєстр сервісів. Схема взаємодії наведена на рисунку 1.2.

Взаємодія між даними елементами виглядає наступним чином: постачальник сервісу реєструє свої сервіси, а споживач звертається до реєстру із запитом. Спілкування відбувається за певним уніфікованим протоколом передачі даних(SOAP, XML).

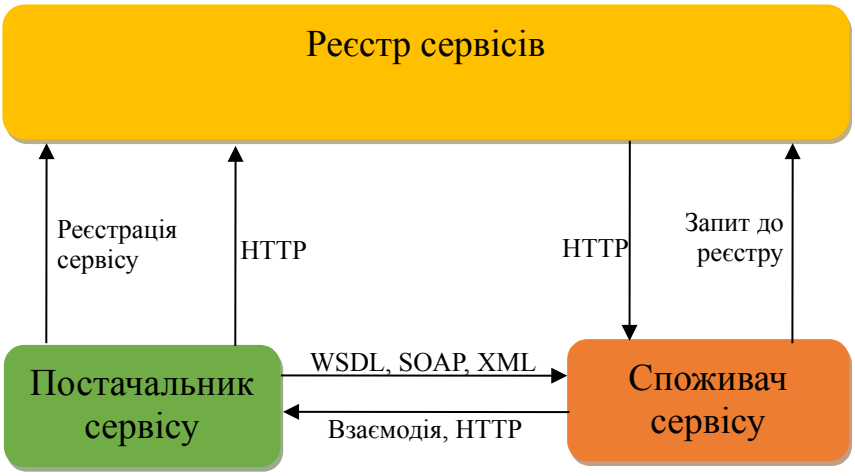

Рисунок 1.2 – Базова схема SOA

Інтерфейси – ключові компоненти SOA. Вони мають бути нейтральними до специфіки реалізації сервісу, які визначаються апаратною платформою, операційною системою чи мовою програмування [4].

1.3 Мікросервісний підхід

Ідея SOA розвинула, так званий, мікросервісний підхід. В цілому, SOA представляє гарні рішення, але не існує чітких правил та настанов як саме правильно досягти успіху в сервіс-орієнтованій архітектурі.

Найбільш вузьке місце в SOA – це складнощі, пов'язані з протоколами обміну даних(таких як SOAP), відсутність методик, які дозволяють оцінити та

встановити ступінь деталізації сервісів, а також місця розділу системи на сервіси. Архітектурний стиль мікросервісів (MicroServices Architecture, MSA) – це підхід, при якому вся система являє собою набір невеликих сервісів, кожен з яких розгортається та працює в окремому процесі та взаємодіє з іншими використовуючи легкі механізми, як правило HTTP. Описані сервіси побудовані навколо бізнес потреб та реалізують свій окремий обмежений предметний контекст з визначеними границями. Всі сервіси можуть бути розроблені на різних мовах програмуванням, використовуючи будь-які фреймворки та засоби зберігання даних.

Мікросервісну архітектуру вважають підмножиною SOA, тому що вона, аналогічно SOA, орієнтована на сервіси та розподілена, використовує уніфіковані протоколи обміну даними. MSA слід розглядати як підхід до реалізації SOA.

Розглянемо ключові положення мікросервісної архітектури. Головним моментом є те, що кожен сервіс має бути невеликим та чітко сфокусованим на своїй задачі в межах певного контексту. Трактувати поняття «невеликий» можна різним чином.

Іншим поняттям, яке характеризує мікросервіси є «сфокусований», тобто сервіс вирішує одну обмежену бізнес-задачу і не більше того. Даний принцип є іншим формулюванням базового принципу єдиної відповідальності (Single Responsibility Principle), якого слід дотримуватися при створенні будь-якого програмного забезпечення [5].

Також кожен мікросервіс має бути слабо зв’язаним та сильно зчепленим. Слабке зв’язування передбачає використання інтерфейсів та інструментів впровадження залежностей (Dependency Injection, DI), які дозволяють вносити зміни в один модуль не змінюючи інший. «Сильно зчепленний» означає, що компонент має всі необхідні методи рішення поставленої задачі. Ідеологія мікросервісів закликає використовувати розумні приймачі та прості канали передачі, замість складних протоколів, типу WS-* або BPEL, слід використовувати неперевантажені протоколи.

Важливим аспектом будь-якої інформаційної системи є організація та управління даними. При мікросервісному підході практикується децентралізоване управління даними. Для стандартного монолітного додатку існує лише одна база даних, яка обслуговує безліч різних компонентів бізнес-логіки системи.

Для мікросервісної архітектури, коли кожен компонент бізнес-логіки являє собою мікросервіс, всі компоненти мають свої власні бази даних, які недоступні іншим мікросервісам. Дані елемента доступні для читання чи запису тільки через відповідний прикладний інтерфейс [6]. Також такий підхід дозволяє використовувати будь-яку кількість різних технологій збереження даних, тобто можливо в одному проєкті використовувати реляційні бази даних, графові бази даних, нереляційні бази даних.

Даний підхід до управління даними має назву Polyglot Persistance. Децентралізація відповідальності за дані серед мікросервісів впливає на те, як ці дані змінюються. Загальний підхід до зміни даних полягає у використанні транзакцій для забезпечення консистентності при зміні даних. Даний підхід характерний для монолітних систем, оскільки забезпечує наступні важливі фактори: узгодженність даних, атомарність та ізольованість, тривалість.

1.3.1 Переваги мікросервісів

Мікросервісна архітектура має багато різноманітних переваг. Деякі з них властиві будь-яким розподіленим системам. Головною ціллю даного архітектурного стилю, як і будь-якої архітектури, це управління складністю. Мікросервісна архітектура вирішує проблему складності для моноліту. Переваги мікросервісів можна розглядати з двох різних аспектів: платформного та програмного. До платформних переваг можна віднести:

- Масштабованість.
- Незалежне розгортання. При використанні мікросервісів можна вносити зміни в окремий модуль та розгортати його незалежно від всіх інших компонентів.
- Гетерогенність технологій. Для реалізації мікросервісів можна обирати будь-яку технологію, яка пасує для поставленої задачі

- Стійкість системи. При виходженні з ладу системи, ми можемо локалізувати причину в рамках конкретного мікросервісу.

Зі сторони програмних переваг можна виділити наступні [2]:

- Модульність. Мікросервісна архітектура дозволяє зберігати модульність та інкапсуляцію, вони забезпечують логічний розподіл системи на модулі за рахунок явного фізичного розділу по серверам. Фізична ізольованість захищає від порушення границь обмежених контекстів.

Також зауважимо, що мікросервіси є легші для розуміння та підтримки, не потрібно вникати одразу в усі подробиці загальної системи. Крім того, можна обирати для кожного сервісу необхідне апаратне забезпечення.

Час запуску та впровадження є значно швидшим, ніж для стандартного тришарового додатку.

MSA надає можливість безперервного розгортання. Також цей підхід надає можливість кожному сервісу масштабуватися незалежним чином. Можливо розгортати таку кількість екземплярів сервісу, яка задовольнить потребу бізнеспотреб. Будь-яка локальна зміна в сервісі може бути легко зроблена розробником і не потребувати комунікацій з командами, які розробляють інші сервіси. В результаті, це надає гнучкості мікросервісам, в порівнянні з монолітом, а також дозволяє легко впровадити CI/CD.

1.3.2 Недоліки мікросервісів

Мікросервісна архітектура не призначена для розв’язку всіх можливих задач та має властиві розподіленим системам недоліки. Найбільше труднощів виникає у питаннях взаємодії мікросервісів, їх інтеграції. Також даний термін з’явився відносно недавно, тому не існує загальноприйнятих специфікацій та рекомендацій для створення якісних додатків.

Наведемо перелік основних складнощів з якими можна зіткнутися при створенні мікросервісного додатку:

- Складність розробки.

- Управління даними. Організація транзакцій є досить складною задачею для розподілених систем, в той час коли для моноліту створення транзакцій є тривіальною задачею, оскільки існує лише єдина база даних.
- Збільшення використання ресурсів. Мікросервісна архітектура вимагає більше ресурсів, ніж монолітна, оскільки кожен мікросервіс необхідно забезпечити власним контейнером з розгорнутим програмним середовищем.
- Збільшення навантаження на мережу. Для взаємодії мікросервіси використовують стандартні протоколи обміну мережею, коли компоненти моноліту спілкуються в рамках єдиного процесу і не вимагають додаткових мережевих викликів [4].
- Тестування системи.
- Моніторинг системи.

Також до недоліків можна віднести складність рефакторингу, особливо якщо рефакторинг вимагає перенесення деякої логіки між сервісами.

Вибір мікросервісної архітектури без попереднього аналізу може привести до безладного проєктування та невдачі всього проєкту.

2. ПОБУДОВА МІКРОСЕРВІСНИХ СИСТЕМ

2.1 Інтеграція мікросервісів

Правильна інтеграція є найважливішим технічним аспектом під час проєктування та реалізації мікросервісів. При належному виконанні мікросервіси збережуть свою автономію, і в той час можна буде вносити в них зміни і випускати їх нові версії незалежно від решти системи. При неправильному виконанні вас чекають серйозні неприємності. Нижче будуть наведені основні проблеми та підходи до їх вирішення. Перш за все розглянемо основні шаблони для створення MSA.

2.1.1 Шаблони проєктування мікросервісів

1. Шаблон «Агрегатор»

Перший та, мабуть, найбільш поширений шаблон проєктування при створенні мікросервісів – «агрегатор».

У найпростішому випадку, агрегатор є звичайною веб-сторінкою, яка викликає безліч сервісів для реалізації функціональності, необхідної додатком. Схема патерну наведена на рисунку 2.1

Оскільки всі сервіси (Service A, Service B і Service C) надаються за допомогою легкого REST-механізму, веб-сторінка може отримати дані й опрацювати їх як потрібно. Якщо необхідне будь-яке додаткове опрацювання, наприклад, застосувати бізнес-логіки до даних, отриманих від окремих сервісів, то для цього у вас може бути CDI-компонент, що перетворює дані таким чином, щоб їх можна було показати на веб-сторінці. Агрегатор може використовуватися і в тих випадках, коли не потрібно нічого відображати, а потрібен лише більш високорівневий мікросервіс, який може використовувати інші сервіси.

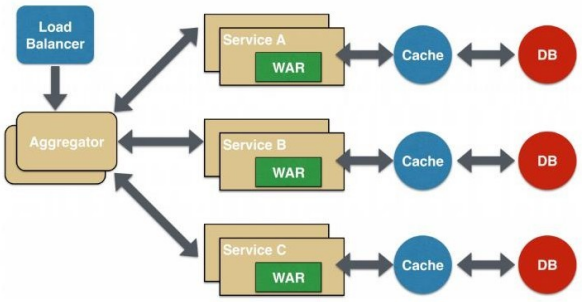

Рисунок 2.1 – Схема шаблону «Агрегатор»

Цей патерн дотримується принципу DRY. Якщо існує безліч сервісів, які повинні звертатися до сервісів A, B і C, то рекомендується абстрагувати цю логіку в мікросервіс і агрегувати її у вигляді окремого сервісу. Перевага абстрагування на цьому рівні полягає в тому, що окремі сервіси, скажімо, A, B і C, можуть розвиватися незалежно, а бізнес-логіку буде, як і раніше, виконувати композитний мікросервіс.

2. Шаблон «Посередник»

Патерн «посередник» при роботі з мікросервісами – це окремий варіант агрегатора. В такому випадку агрегація повинна відбуватися на клієнті, але в залежності від бізнес-вимог при цьому може викликатися додатковий мікросервіс. На рисунку 2.2 наведена схема даного шаблону, як і агрегатор, посередник може незалежно масштабуватися по горизонталі і по вертикалі. Це може знадобитися в ситуації, коли кожен окремий сервіс потрібно не надавати споживачеві, а запускати через інтерфейс.

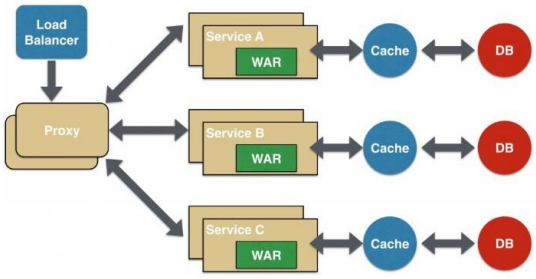

Рисунок 2.2 – Схема шаблону «Посередник»

Посередник може бути формальним, в такому випадку він просто

делегує запит одному з сервісів. Він може бути і розумним, в такому випадку

3. Шаблон «Ланцюг»

Мікросервісний патерн проєктування «Ланцюг» видає єдину консолідовану відповідь на запит. В даному випадку сервіс А отримує запит від клієнта, зв'язується з сервісом В, який, в свою чергу, може зв'язатися з сервісом С.

Архітектура побудови мікросервісів за моделлю «Ланцюг» наведена на рисунку 2.3. Всі ці сервіси, як правило, обмінюються синхронними повідомленнями «запит / відповідь» по протоколу HTTP. Найважливішим моментом є те, що клієнт блокується до тих пір, поки не виконається вся комунікаційна послідовність запитів і відповідей, тобто Service A - Service B і Service B - Service C. Запит від Service B до Service C може виглядати зовсім інакше, ніж від Service A до Service B. Це найбільш важливо у всіх випадках, коли бізнес-цінність декількох сервісів додається.

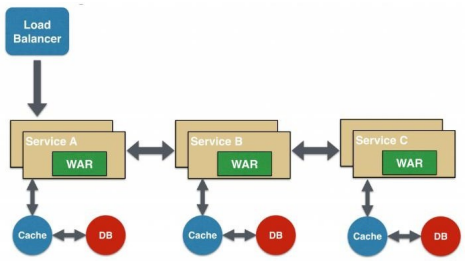

Рисунок 2.3 – Схема шаблону «Ланцюг»

Також важливо зрозуміти, що не можна робити ланцюг занадто довгим. Це критично, оскільки ланцюг синхронний за своєю природою, і чим він довший, тим довше доведеться чекати клієнтові, особливо якщо відгук полягає у виведенні веб-сторінки на екран. Існують способи обійти такий блокуючий механізм запитів і відгуків, і вони розглядаються в наступному шаблоні.

4. Шаблон «Гілка»

Мікросервісний шаблон проєктування «Гілка» розширює шаблон «Агрегатор» і забезпечує одночасну обробку відповідей від двох ланцюгів

мікросервісів, які можуть бути взаємовиключними. Цей патерн також може застосовуватися для виклику різних ланцюгів, або одного і того ж ланцюга - в залежності від потреб. Приклад взаємодії сервісів наведено на рисунку 2.4.

В іншому випадку сервіс А може викликати лише один ланцюг в залежності від того, який запит отримає від клієнта. Такий механізм можна конфігурувати, реалізувавши маршрутизацію кінцевих точок JAX-RS, в такому випадку конфігурація повинна бути динамічною.

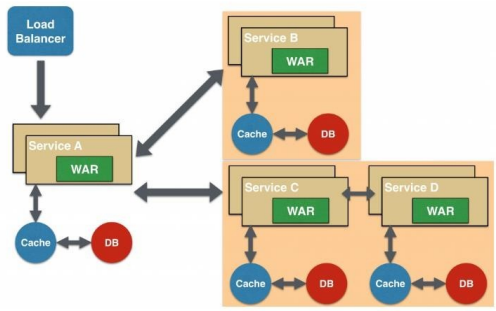

Рисунок 2.4 – Схема шаблону «Гілка»

5. Шаблон «Дані спільного використання»

Один з принципів проєктування мікросервісів — автономність. Це означає, що сервіс повностековий і контролює всі компоненти: інтерфейс, призначений для користувача, проміжне ПЗ, транзакції. В такому випадку сервіс може бути багатомовним і вирішувати кожну задачу за допомогою найбільш відповідних інструментів. Наприклад, якщо при необхідності можна застосувати сховище даних NoSQL, то краще зробити саме так, а не додавати всю цю інформацію в базу даних SQL. Однак, типова проблема, особливо при рефакторингу наявного монолітного додатку, пов'язана з нормалізацією бази даних — так, щоб у кожного мікросервісу був строго визначений обсяг інформації. На рисунку 2.5 продемонстровано базову схему даного патерну. За цим паттерном кілька мікросервісів можуть працювати по ланцюгу і спільно використовувати сховища кеша і бази даних. Це доцільно лише в разі, якщо між двома сервісами існує сильний зв'язок. Деякі можуть вбачати в цьому антипаттерн, але в деяких бізнес- ситуаціях такий шаблон дійсно доречний.

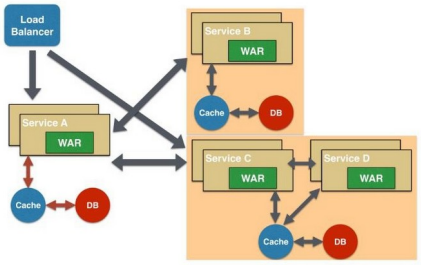

Рисунок 2.5 – Схема шаблону «Дані спільного використання»

Крім того, його можна розглядати як проміжний етап, який потрібно подолати, поки мікросервіси не стануть повністю автономними.

6. Шаблон «Асинхронні повідомлення»

При всій поширеності і зрозумілості підходу REST, у нього є важливе обмеження, а саме: він синхронний і, отже, блокуючий. Забезпечити асинхронність можна, але це робиться по-своєму в кожному додатку. Тому в деяких мікросервісних архітектурах можуть використовуватися черги повідомлень, а не модель REST - запит / відповідь.

На рисунку 2.6 описано як сервіс A може синхронно викликати сервіс C, який потім буде асинхронно зв'язуватися з сервісами B і D за допомогою черги повідомлень. Комунікація між сервісами A та C може бути асинхронною, скажімо, з використанням веб-сокетів; так досягається бажана масштабованість.

Комбінація моделі REST(запит/відповідь) та обміну повідомленнями (видавник / підписник) також можуть використовуватися для досягнення поставлених цілей.

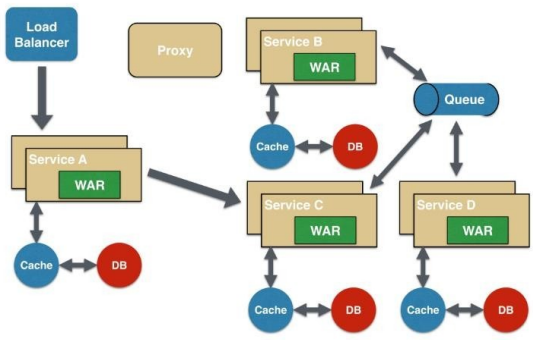

Рисунок 2.6 – Схема шаблону «Асинхронні повідомлення»

2.1.2 Типи комунікації мікросервісів

Існує два основних підходи до організації комунікації мікросервісів, які лежать в основі всіх шаблонів наведених вище: синхронний (REST, RPC) та асинхронний обмін повідомленнями.

2.2. Розбиття моноліту на частини

Майже всі успішні приклади використання мікросервісної архітектури, розпочиналися з моноліту, який з плином часу розростався. І доволі частими є випадки, коли проєкт розпочинався з мікросервісів і не досягав своєї мети.

Тому можна зробити висновок, що краще розпочинати новий проєкт з моноліту, навіть якщо ви знаєте, що ваш проєкт буде достатньо великим для використання мікросервісів. MSA є корисною та ефективною архітектурою, але всі її переваги доцільні лише для великих та складних систем. Для простих систем набагато краще підходить суцільна монолітна архітектура.

Перша причина дотримуватися принципу «спочатку - моноліт» — класичний принцип «Вам це не знадобиться». Коли ви починаєте розробляти новий додаток, необхідно впевнитись, що він буде корисним для користувачів. Кращий спосіб перевірити, чи буде додаток користуватися попитом — це створення його спрощеної версії. Спочатку на першому місці стоїть швидкість розробки, а розробка мікросервісів займає набагато більше часу.

Наступна проблема полягає в тому, що мікросервіси працюють добре, якщо ви досягли чітких, стабільних меж між окремими сервісами — для цього потрібно отримати правильний набір обмежених контекстів. Будь-який рефакторинг функціональності між сервісами складніший аналогічного в моноліті.

Побудувавши монолітний додаток, ви зможете визначити вірні межі, перш ніж використовувати мікросервіси. Це також дасть вам час підготувати

все необхідне для створення сервісів з більш чіткими обмеженими контекстами.

Існують різні шляхи реалізації стратегії «спочатку - моноліт».

- Логічний шлях – це проєктувати моноліт з усією обережністю, звертаючи увагу на модульність в програмному забезпеченні, межі API та спосіб зберігання даних. Якщо зробити це добре, то перехід до мікросервісів буде відносно простим.
- Більш загальний підхід – почати з моноліту і поступово відокремлювати від нього мікросервіси. У цьому випадку значна частина початкового моноліту може залишитися в якості центральної в мікросервісній архітектурі, але основна частина нової розробки буде відбуватися в сервісах, залишаючи моноліт без великих змін.
- Також поширений підхід з повною заміною моноліту.
- Ще один варіант розбиття моноліту – почати з декількох сервісів, які більші за тих, що очікуються в кінці. Використовуйте ці великі сервіси, щоб навчитися працювати в мультисервісному середовищі [6].

Хоча багато переваг надає підхід «спочатку - моноліт», далеко не всі розділяють такий метод. Контраргумент полягає в тому, що починаючи з мікросервісів, ви звикаєте до ритму розробки в такому оточенні.

2.3 Розгортання та масштабування мікросервісів

Монолітні системи розгортаються досить просто, у мікросервісному випадку все не так тривіально. Перш за все розглянемо питання безперервної інтеграції і безперервної доставки.

2.3.1 Основні підходи до розгортання мікросервісів

При використанні CI основною метою є підтримка загального синхронізованого стану, яка досягається шляхом перевірки того, чи щойно введений в експлуатацію код інтегрується з існуючим кодом належним чином.

Для досягнення цієї мети CI-сервер визначає переданий код і здійснює ряд перевірок, переконуючись в тому, що код скомпільований та тести з ним проходять без збоїв. В якості частини даного процесу часто створюється артефакт (або артефакти), які використовуються для подальшої перевірки, наприклад розгортання працюючого сервісу з метою запуску для нього ряду тестів. Бажано створювати ці артефакти тільки один раз і використовувати їх для всіх розгортань тієї чи іншої версії коду.

Щоб допустити повторне використання цих артефактів, ми розміщуємо їх в особливе сховище, яке надається CI-інструментарієм, або знаходиться в окремій системі. Розмірковуючи про мікросервіси та безперервну інтеграцію, треба думати про те як засіб CI створює відображення на окремо взяті мікросервіси.

В найпростішому випадку, все розміщується в одному великому сховищі, де знаходиться весь код(рис. 2.7).

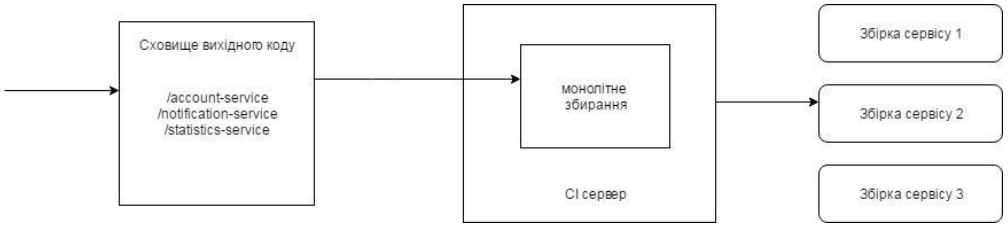

Рисунок 2.7 – Збирання за спільним сховищем коду

Взагалі, це шаблон, якого слід уникати, але на ранній стадії проєкту, особливо якщо над проєктом працює одна команда, його короткочасне використання має певний сенс.

У даної моделі є безліч недоліків. При внесенні навіть невеликих змін в будь-який сервіс, будуть зібрані та перевірені всі інші сервіси. На це може піти багато часу, оскільки необхідно буде чекати проходження тестів там, де це не є потрібним. Різновидом вище наведеного підходу є наявність єдиного дерева для всього вихідного коду і декількох CI-збірок, які відображаються на частини цього дерева. Приклад даного збирання проєкту наведено на рисунку 2.8

Даний підхід є кращим ніж попередній, але все однак він передбачає перевірку всього вихідного коду, хоча збирання буде здійснюватис я для кожного сервісу окремо.

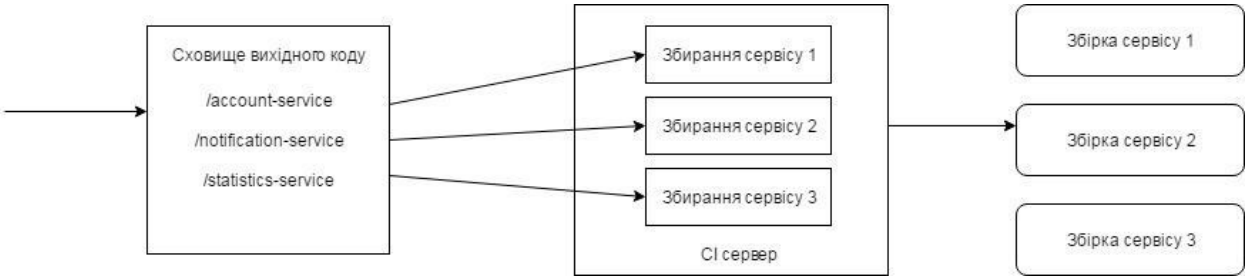

Рисунок 2.8 – Збирання на базі деревоподібної структури

Найбільш практичним є підхід зображений на рисунку 2.9, коли у кожного мікросервісу існує своє окреме сховище для вихідного коду, яке відображається на його власну CI-збірку. При внесенні змін запускається та тестується тільки необхідна збірка.

Але при цьому виникає додаткова складність при внесенні змін, які зачіпають відразу декілька сховищ.

Концепція конвеєрного збирання надає відмінний спосіб відстеження ходу обробки програми у міру прояснення ситуації на кожній стадії, допомагаючи з'ясувати якість програми. Ми здійснюємо збірку нашого артефакту, і цей артефакт використовується на всьому протязі конвеєра.

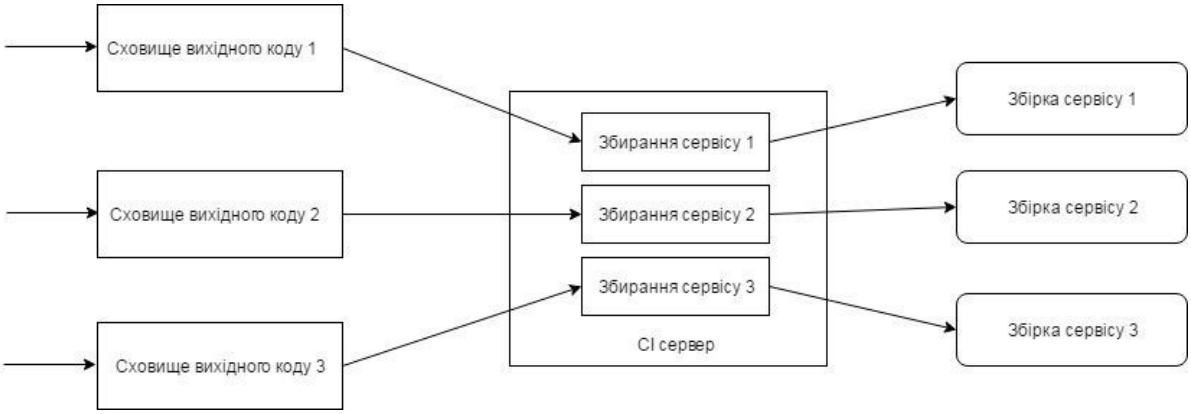

Рисунок 2.9 – Збирання при наявності репозиторію для кожного сервісу

2.3.2 Масштабування мікросервісів

Володіти системою здатною автоматично масштабуватись та відповідно реагувати на збільшення навантаження або відмову деяких вузлів є пріоритетною вимогою, але для деяких випадків це виявиться не актуальним і ресурсозатратним.

Коли розглядаються питання про необхідність і способи масштабування системи, що дозволяє краще впоратися з навантаженням або збоями, висуваються наступні вимоги:

- Час відгуку / затримки.
- Доступність.
- Збереження даних.

Важливою частиною створення відмовостійкої системи, особливо коли функціональні можливості розподіляються серед декількох мікросервісів, які можуть перебувати як в робочому, так і в неробочому стані, є забезпечення її спроможності безпечно знижувати рівень функціональності. При роботі з єдиним монолітним додатком нам не доводиться приймати безліч рішень. Працездатність системи залежить від роботи двійкового коду. Але при використанні архітектури мікросервісів потрібно розглядати набагато складніші ситуації. Чим більше один сервіс залежить від залучення інших сервісів, тим більше успішна робота одного сервісу впливає на виконання завдань іншими сервісами. Використання технологій інтеграції, що дозволяють переводити нижчий сервер в режим автономної роботи, може знизити ймовірність впливу простоїв, як планових, так і позапланових збоїв на вищі сервіси.

При проведенні ідемпотентних операцій результат після першого застосування не змінюється, навіть якщо операція послідовно виконується ще кілька разів. Якщо операції є ідемпотентними, ми можемо повторювати виклик кілька разів без негативного впливу. Це нам дуже знадобиться, якщо необхідно повторно відтворити повідомлення, коли немає впевненості, що вони оброблені. Це є досить поширеним способом відновлення після помилок. В основному масштабування систем виконується з двох причин.

По-перше, для того, щоб легше було впоратися зі збоями: якщо ми переживаємо за відмову будь-якого компонента, то допомогти зможе наявність такого ж додаткового компонента.

По-друге, для підвищення продуктивності, що дозволяє або впоратися з більш високим навантаженням, або знизити час відгуку, або досягти обох результатів.

Розглянемо ряд найбільш поширених технологій масштабування, якими можна буде скористатися, і подумаємо про їх застосування до архітектури мікросервісів.

1. Нарощування потужностей

Від нарощування потужностей деякі операції можуть тільки виграти. Більш об'ємний корпус з більш швидким центральним процесором і більш ефективної підсистемою введення-виведення часто здатні зменшити затримки і підвищити пропускну здатність, дозволяючи виконувати більший обсяг робіт за менший час. Але такий різновид масштабування, яку часто називають вертикальним масштабуванням, може бути занадто витратним: іноді один великий сервер може

коштувати набагато більше, ніж два невеликих сервера нижчої потужності.

2. Розподіл робочих навантажень

Наявність єдиного мікросервіса на кожному хості, безумовно, краще моделі, яка передбачає наявність на хості відразу декількох мікросервісів. Але спочатку, з метою зниження вартості обладнання або спрощення управління хостом, багато хто приймає рішення про співіснування кількох мікросервісів на

одній фізичній машині. Оскільки мікросервіси запускаються в незалежних процесах, які обмінюються даними по мережі, завдання подальшого їх переміщення на власні хости з метою підвищення пропускної спроможності і масштабування не представляє особливої складності.

3. Балансування навантаження

Коли сервісу потрібна відмовостійкість, вам знадобляться способи обходу критичних місць збоїв. Для мікросервісу, який надає синхронну кінцеву точку по HTTP, найбільш простим способом вирішення цього завдання (рис. 2.10) буде використання декількох хостів із запущеними на них екземплярами мікросервіс у, що знаходяться за балансувальником навантаження. Споживачі мікросервісу не знають, чи пов'язані вони з одним його екземпляром або з сотнею таких екземплярів.

Рисунок 2.10 – Схема розподілу навантаження

4. Системи на основі виконавців

Застосування балансувальника не є єдиним способом поділу навантаження серед кількох екземплярів сервісу та зменшення їх крихкості. Дана модель також добре працює при пікових навантаженнях, де в міру зростання потреб можуть запускатися додаткові екземпляри для відповідності вхідного навантаження. Поки сама черга робіт буде зберігати стійкість, ця модель може використовувати масштабування для підвищення як пропускної здатності робіт, так і відмовостійкості, оскільки стає простіше впоратися з впливом відмовив (або відсутнього) виконавця. Робота займе більше часу, але нічого при цьому не втратиться [7].

3. ПРИКЛАД СТВОРЕННЯ ТЕСТОВОЇ СИСТЕМИ НА БАЗІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

В даному розділі буде розглянуто приклад побудови мікросервісної архітектури для обміну валют. Моє завдання полягало у розробці мікросервісної архітектури за допомогою технологій Java.

Java – одна з найпопулярніших та найпотужніших мов програмування сучасності. Програми на Java транслюються в байт-код, який виконується віртуальною машиною Java(JVM) – програмою, яка опрацьовує байтовий код та передає інструкції обладнанню як інтерпретатор. Перевага подібного способу виконання програм є повна незалежність байт-коду від операційної системи та апаратури, що дозволяє виконувати Java-додатки на будь-якому пристрої, для якого існує відповідна віртуальна машина.

Java без пребільшення, є найпопулярнішою мовою для побудови мікросервісів. Багато відомих компаній створюють свої системи саме на цій мові, прикладом є Netflix, Amazon.

Spring framework – найпотужніша Java-бібліотека, яка дозволяє створювати програмні системи будь-якої складності. Виник як альтернатива J2EE платформи. Для побудови мікросервісів найважливіші модулі Spring це Spring Boot та Spring Cloud.

Spring Boot – складова екосистеми бібліотеки Spring. Spring Boot дозволяє легко створювати повноцінні, ефективні Spring-додатки. Для налаштування програмної системи необхідно відносно мало конфігурацій, в порівнянні з Spring MVC.

Основні особливості Spring Boot:

- Створення повноцінних Spring додатків;
- Вбудований сервер Tomcat або Jetty;
- Автоматична конфігурація Spring framework;
- Забезпечує можливостями моніторингу стану системи;

- Не вимагає написання конфігураційних файлів на XML [8].

Spring Boot є основою для більш складного фреймворку Spring Cloud, який призначений для створення розподілених систем та має широкий інструментарій для розробки.

З основних переваг використання Spring Cloud виділимо наступні:

- наявність реєстру сервісів та системи виявлення сервісів, маршрутизації, міжсервісних викликів;
- балансування навантаження;
- наявність автоматичного вимикача;
- розподілений обмін повідомленнями.

Spring Cloud надає декларативний підхід до створення програмного забезпечення.

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application  {
    public static void main(String[] args) {
        SpringApplication.run(Application.class,  args);
    }
}
```

В даному прикладі використовується лише єдина анотація *@EnableDiscoveryClient* для введення в систему механізму виявлення сервісів. Також Spring Cloud має чудову інтеграцію з Netflix OSS, та надає дані інструменти «з коробки»:

```
@SpringBootApplication
@EnableEurikaServer
public class ServiceRegistrationServer {
    public static void main(String[]
    args)  {
        System.setProperty("spring-
config.yml", "registration-server");

SpringApplication.run(ServiceRegistrationServer.class,
args);
```

```
    }
}
```

Вище наведено приклад використання компоненту Netflix OSS через Spring

Cloud. Eurika – це сервер, який виконує функцію реєстрації сервісів.

Для включення балансувальника навантаження також необхідно докласти мінімально зусиль, а саме включити анотацію *@LoadBalanced* над відповідним Spring-компонентом.

```
@Autowired
@LoadBalanced
protected RestTemplate restTemplate;
```

Також Spring Cloud надає горизонтально масштабоване сховище конфігурацій для розподілених систем. Як джерело даних на даний момент підтримується Git, Subversion та прості файли, що зберігаються локально. За замовчуванням Spring Cloud Config віддає файли, які відповідають імені викликаючого Spring додатку.

Spring Cloud надає зручні анотації та автоконфігурації для забезпечення аутентифікації, створення токенів OAuth2 для доступу до ресурсів бекенду.

Spring Cloud спрощує підключення до сервісів та отримання можливостей середовища в хмарних платформах, таких як Cloud Foundry і Heroku. Особлива підтримка Spring-додатків через Java і XML-конфігурації робить підключення до хмарних сервісів тривіальним завданням. Ви можете використовувати існуючі хмарні коннектори або написати власний для вашої хмарної платформи. "З коробки" підтримуються найбільш популярні сервіси (реляційні СУБД, MongoDB, Redis, Rabbit), але також можливе розширення для ваших сервісів. Жоден з сервісів не вимагає зміни самого Spring Cloud, досить просто додати необхідну вам jar- бібліотеку в область видимості classpath [8].

Для інтеграції сервісів я використала інструменти компанії Netflix, які знаходяться у вільному доступі, а саме: Ribbon, Eureka.

Моя розробка полягає у створенні декількох мікросервісів, які

взаємодіють між собою за допомогою сервісу імен Eureka та реалізації балансування на стороні клієнта за допомогою Ribbon.

Всю систему можна умовно розподілити на 2 основні частини: набір допоміжних сервісів, таких як точка для єдиного входу, автоматичний вимикач, а також незалежні самостійні мікросервіси. Всього в системі три мікросервіси: Forex Service, CurrencyConversion Service, Eureka Service. Система побудована за шаблоном агрегатор.

Forex Service надає системі можливість отримати значення курсів для різних валют.

CurrencyConversion Service може конвертувати безліч валют в іншу валюту. Він використовує Forex Service для отримання поточних значень обміну валюти. Тому Exchange Service є споживачем послуг.

Eureka Service забезпечує взаємодію між іншими сервісами без потреби щоразу змінювати конфігурацію CurrencyConversion Service для появи кожного нового екземпляру Forex Service.

Основні компоненти системи наведені на рис 3.1

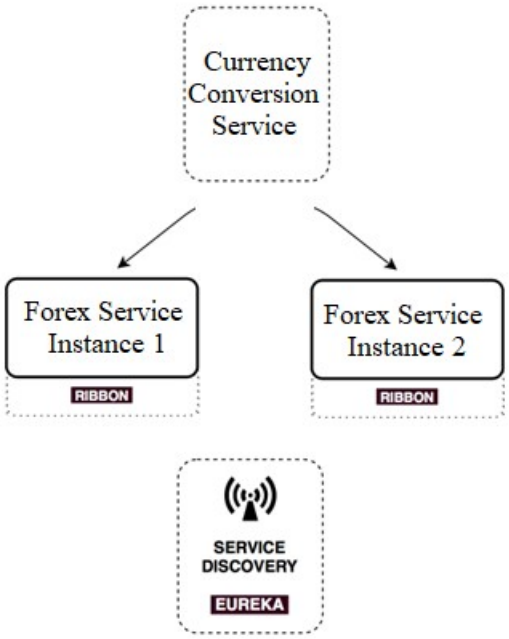

Рисунок 3.1 – Загальна архітектура додатку

Далі буде детально розглянуто кожен із складових компонентів системи.

3.1 Forex Service

Даний мікросервіс є вагомою частиною всієї системи та надає системі можливість отримувати значення курсів, необхідні при конвертації валют. Можемо припустити, що він взаємодіє з Forex Exchange і надає поточну вартість при конвертації валют. Приклад запиту та відповіді:

```
GET to
http://localhost:8000/currency-exchange/from/UAH/to/USD

{ id: 10002, from: "UAH", to: "USD", conversionMultiple:
25, port: 8000 }
```

Відповіддю на запит є конвертація з UAH до USD. У відповіді отримуємо значення 75.

3.2 CurrencyConversion Service

Даний сервіс може конвертувати множину валют у іншу валюту. Він використовує сервіс Forex для отримання поточних значень обміну валюти. Приклад запиту та відповіді для сервісу:

```
GET to
http://localhost:8100/currency-converter/from/UAH/to/USD/qu
antity/10000

{ id: 10002, from: " UAH ", to: " USD", conversionMultiple:
25, quantity: 250000, totalCalculatedAmount: 10000, port:
8000 }
```

Цей запит дозволяє дізнатися суму при конвертації 10000 гривень у долари. Значення – 10000 USD.

3.3 Балансування навантажень за допомогою Ribbon

При реалізації Forex Service та CurrencyConversion Service ми створюємо взаємодію між ними за допомогою наступного коду:

```
@FeignClient(name="forex-service" url="localhost:8000")

    public interface CurrencyExchangeServiceProxy
```

```
{

  @GetMapping("/currency-exchange/from/{from}/to/{to}")

  public CurrencyConversionBean retrieveExchangeValue
  (@PathVariable("from") String from,
  @PathVariable("to") String to);

}
```

Але у такій ситуації при запуску нових екземплярів Forex Service у нас немає можливості розподілити навантаження між ними.

Щоб вирішити цю проблему, я реалізувала балансування навантажень на клієнтській стороні за допомогою Ribbon (це балансувальник навантаження на стороні клієнта, який дає великий контроль над поведінкою клієнтів HTTP та TCP) за допомогою написання коду у відповідних частинах проєкту:

- у залежностях:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-ribbon</
artifactId>
</dependency>
```

- в частині CurrencyExchangeServiceProxy:

```
@FeignClient(name="forex-service")
@RibbonClient(name="forex-service")
public interface CurrencyExchangeServiceProxy {
```

- в властивостях аплікації:

```
forex-
```

```
service.ribbon.listOfServers=localhost:8000,localhost:8001
```

3.4 Інфраструктурний сервіс Netflix Eureka

Service Discovery є одним з ключових принципів мікросервісної архітектури. Задача виявлення сервісів була вирішена використанням Netflix Eureka. Eureka – це сервер для реєстрації всіх сервісів, які знаходяться в системі. Для запуску серверу я створила простий Spring Boot додаток з анотацією @EnableEurekaServer:

```
@SpringBootApplication
@EnableEurikaServer
public class
SpringBootMicroserviceEurekaNamingServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootMicroserviceEurek
        aNamingServerApplication.class, args);
    }
}
```

Опісля я налаштувала ім’я та порт для сервісу:

```
spring.application.name=netflix-eureka-naming-server
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

При правильному налаштуванні серверу, відкрилося вікно браузера з панеллю управління, приклад якої наведено на рисунку 3.2

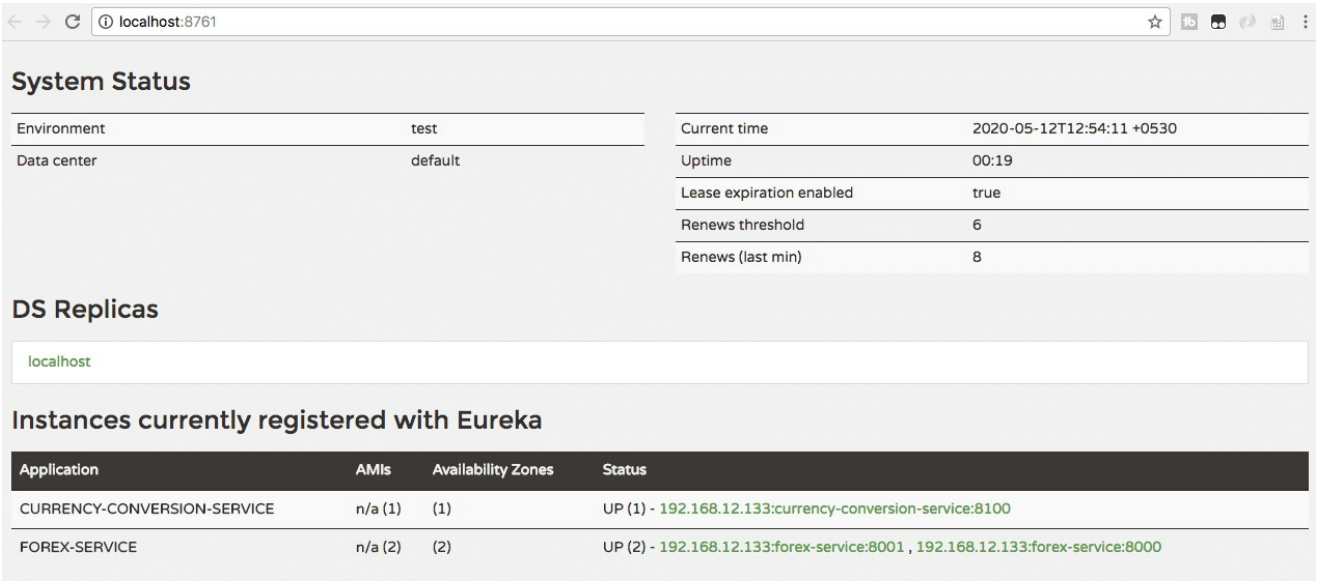

Рисунок 3.2 – Панель сервісу Netflix Eureka

Дана панель демонструє всі сервіси, які зареєстровані в системі.

ВИСНОВКИ

В даній роботі було досліджено основні концепції побудови додатку на базі мікросервісної архітектури. Розглянуто основні особливості, переваги та недоліки даного підходу до побудови програмних систем, порівняно з класичним монолітним рішенням. Монолітна архітектура дуже добре розв'язує свої задачі, але із зростанням складності вона вже не може якісно вирішувати свої функції, тому розвинулися сервіс-орієнтовані архітектури, прикладом якої є мікросервісна архітектура.

У ході роботи мною було детально розглянуто основні принципи проєктування та розгортання даних систем, основні шаблони інтеграції мікросервісів, технології комунікації. Також було досліджено ключові компоненти для побудови мікросервісних систем, які утворюють інфраструктурний рівень та надають необхідну гнучкість всій системі. До даних компонентів відносяться: сервіс єдиного входу, сервіс відкриття, балансувальних навантаження, автоматичний вимикач.

В тестовому прикладі було продемонстровано працездатність вищенаведених концепцій. Для побудови тестової архітектури я використала Spring Framework, як найпотужнішу Java-бібліотеку для створення додатків різної складності, а також інструменти для забезпечення інтеграції та внутрішні бібліотеки, які надають змогу розгортати мережеві розподілені додатки: шаблон «Агрегатор», як зручний патерн для побудови WEB-аплікації, та шаблон мікросервісної архітектури Service Discovery. Дана система піддається горизонтальному масштабуванню шляхом збільшення вузлів в системі та володіє необхідною відмовостійкістю.

Список використаних джерел

1. М. Фаулер. Архитектура корпоративных программных приложений /М. Фаулер. – Издательский дом Вильямс, 2006 – 544 с.
2. Ньюмен С. Создание микросервисов / Ньюмен С. – СПб.:Питер, 2016 – 304с.
3. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions / Gregor Hohpe, Bobby Woolf – Addison-Wesley,2004 -736c.
4. Chris Richardson. From Design to Deployment / Chris Richardson, Floyd Smith, 2016. – 74 p.
5. E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software /E. Evans – Addison-Wesley, 2003 – 560 p.
6. Martin Fowler – Microservices – Режим доступу: http://martinfowler.com/articles/microservices.html
7. Nadareishvili. Microservice Architecture: Aligning Principles, Practices, and Culture / I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen – O'Reilly Media, 2016 – 146 p.
8. Офіційний сайт Spring framework. – Режим доступу: https://spring.io