Міністерство освіти і науки України

Національний університет «Львівська політехніка»

Кафедра ЕОМ

Екзаменаційна робота

з курсу: «Напрямки дослідження та розвитку системного програмування»

на тему:

**«Оптимізація мікросервісного архітектурного підходу розробки програмних систем за рахунок впровадження асинхронного обміну повідомленнями »**

Виконав: ст. гр. КІСП-21

Мокряк Б.Ю.

Перевірив:

Бочкарьов О.Ю.

Львів - 2019

**УДК [004.65](https://teacode.com/online/udc/00/004.65.html), 004.62**

**Мокряк Б.Ю., Ігнатович А.О.**

Національний університет "Львівська політехніка"

кафедра електронних обчислювальних машин

**Оптимізація мікросервісного архітектурного підходу розробки програмних систем за рахунок впровадження асинхронного обміну повідомленнями**

*© Мокряк Б.Ю., Ігнатович А.О., 2019*

**Розглянуто проблему створення програмних систем з використанням мікросервісної архітектури. Проаналізовані можливі способи оптимізації системи, а саме за рахунок асинхронізації клієнт-сервісної взаємодії. Запропоновано методи заміни синхронного методу комунікації на асинхронний.**

**Ключові слова: мікросервіси, асинхронність, програмні системи.**

**Considered the problem of creating software systems using microservice architecture approach. Analyzed possible ways of system optimization, namely asynchronization of the client-service interaction. Offered methods of replacement synchronous client-service communication to asynchronous.**

**Keywords: microservices, asynchrony, software systems.**

**Вступ.** Проблема проектування 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. В даній статті буде розглянутий альтернативний та більш новий підхід до створення складних легкомасштабованих програмних систем – з використанням мікросервісної архітектури.

**Стан проблеми.** 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 сервісорієнтована архітектура (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 (MSA). Мікросервісну архітектуру можна вважати підмножиною SOA, але все ж таки MSA відрізняється від класичного SOA. Основнa 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:

1. Низька зв’язність між основними компонентами системи та висока зчепленність коду в окремо взятому сервісі
2. 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.
3. Просте масштабування системи. Ми можемо запускати будь-яку кількість сервісів на окремих серверах
4. Відмовостійкість. При виведенні з ладу одного сервісу вся програма ще може вірно працювати
5. Застосування різних мов програмування та технологій
6. Для створення мікросервісів можна використовувати компактні групи розробників, які є відповідальними за власний сервіс.

До мінусів мікросервісної архітектури можна віднести:

1. Відносна складність розробки, прямо пропорційно залежить від кількості обраних мов програмування та фреймворків.

2. Витрачаються додаткові ресурси на пересилання повідомлень між сервісами та на їх серіалізацію та десеріалізацію

3. Проблеми з версіонуванням

4. Відносно складне інтеграційне тестування.

**Постановка задачі.** Дослідити можливість оптимізації мікросервісного архітектурного підходу проектування програмних систем за рахунок впровадження моделі асинхронної клієнт-сервісної взаємодії замість синхронної.

**Розв’язок задачі.** Розглянемо синхронний підхід на прикладі REST. REST - це надзвичайно популярний механізм IPC. Проблема REST полягає в тому, що це синхронний протокол: HTTP-клієнту доводиться чекати, поки сервіс не поверне відповідь. Кожен раз, коли сервіси спілкуються між собою з синхронного протоколу, це знижує доступність додатка.

Розглянемо недоліки синхронного підходу на прикладі сервісу Order. Є інтерфейс REST API для створення замовлень. Для перевірки замовлення він звертається до сервісів Consumer і Restaurant, які теж мають REST API.

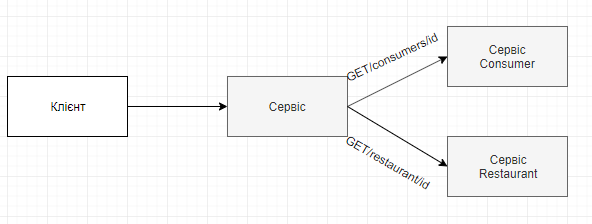


Рис.1 Демонстрація синхронної міжсервісної комунікації

Зі схеми стає зрозуміло – для того, щоб клієнт отримав відповідь на свій запит, керуючий сервіс має доступитися до двох інших – а значить всі сервіси необхідні для формування відповіді повинні бути доступні, і якщо якийсь з них недоступний – клієнт буде змушений чекати. Це і є найбільшим недоліком синхронної клієнт-сервісної комунікації. Тож, розглянемо методи оптимізації комунікації за рахунок впровадження асинхронності.

Перш за все всі комунікаціями між клієнтами та сервісами можливо і слід проектувати асинхронно. Використовуючи попередній приклад, клієнт застосовує для створення замовлень асинхронний стиль взаємодії виду «запит / асинхронний відповідь». Щоб створити замовлення, він відправляє повідомлення із запитом сервісу Order. Потім цей сервіс асинхронно обмінюється повідомленнями з іншими сервісами і в підсумку повертає клієнту відповідь. Клієнт і сервіс спілкуються асинхронно, відправляючи повідомлення через канали. Жоден з учасників цієї взаємодії не блокується в очікуванні відповіді.

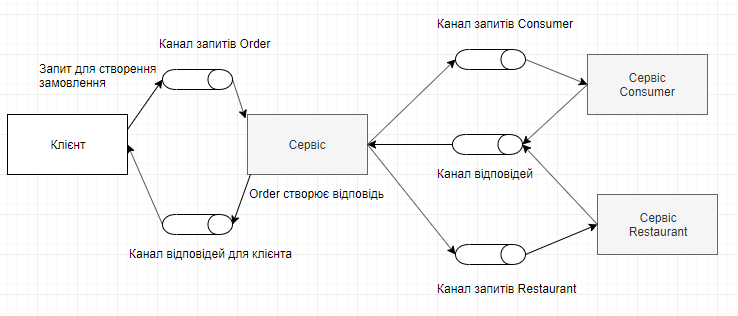


Рис. 2 Комунікація клінта та сервісів за допомогою асихнронних повідомлень

Ще одним із способів мінімізації синхронного взаємодії під час обробки запитів є реплікація даних. Сервіс зберігає копію (репліку) даних, які йому потрібні для обробки запитів. Щоб підтримувати репліку в актуальному стані, він підписується на події, що публікуються сервісами, яким ці дані належать. Наприклад, сервіс Order може зберігати копію даних, що належать сервісам Consumer і Restaurant. Це дозволить йому обробляти запити на створення замовлень, не звертаючись до цих сервісів. Така архітектура показана на рисунку нижче.

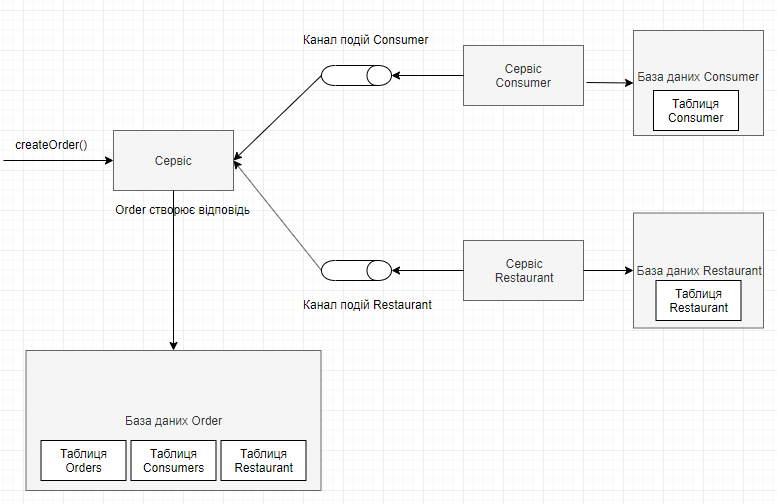


Рис. 3 Сервіс Order автономний, так як має копії даних інших сервісів

Сервіси Consumer і Restaurant публікують події щоразу, коли їх дані змінюються. Сервіс Order підписується на ці події і оновлює свою репліку.

У деяких випадках реплікація даних - це хороше рішення. Один з недоліків цього підходу пов'язаний з тим, що іноді він вимагає копіювання великих обсягів даних, що неефективно. Наприклад, якщо у нас багато замовників, зберігати репліку даних, що належать сервісу Consumer, може виявитися непрактично. Ще один недолік реплікації криється в тому, що вона не вирішує проблему поновлення даних, що належать іншим сервісам.

Ще один спосіб усунення синхронного взаємодії під час обробки запитів полягає в тому, щоб виконувати цю обробку у вигляді наступних етапів.

* Сервіс перевіряє запит тільки за допомогою даних, доступних локально.
* Він оновлює свою базу даних, в тому числі додає повідомлення в таблицю OUTBOX.
* Повертає відповідь своєму клієнтові.

Під час обробки запиту сервіс не звертається синхронно ні до яких інших сервісів. Замість цього він шле їм асинхронні повідомлення. Даний підхід забезпечує слабку зв'язаність сервісів. Як ви побачите в наступному розділі, цей процес часто реалізується у вигляді розповіді.

Сервіс Order мав би діяти наступним чином. Він створює замовлення зі станом PENDING і потім перевіряє його, обмінюючись асинхронними повідомленнями з іншими сервісами. На рис. 3 показано, що відбувається при виклику операції createOrder (). Ланцюжок подій виглядає так.

* Сервіс Order створює замовлення зі станом PENDING.
* Сервіс Order повертає своєму клієнтові відповідь з ID замовлення.
* Сервіс Order шле повідомлення ValidateConsumerInfo сервісу Consumer.
* Сервіс Order шле повідомлення ValidateOrderDetails сервісу Restaurant.
* Сервіс Consumer отримує повідомлення ValidateConsumerInfo, перевіряє, чи може замовник розміщувати замовлення, і відправляє повідомлення ConsumerValidated сервісу Order.
* Сервіс Restaurant отримує повідомлення ValidateOrderDetails, перевіряє коректність елементів меню і здатність ресторану доставити замовлення за заданою адресою і відправляє повідомлення OrderDetailsValidated сервісу Order.
* Сервіс Order отримує повідомлення ConsumerValidated і OrderDetailsValidated і змінює стан замовлення на VALIDATED.

Сервіс Order може отримати повідомлення ConsumerValidated і OrderDetailsValidated в будь-якому порядку. Щоб знати, яке з них він отримав першим, він змінює стан замовлення. Якщо першим прийшло повідомлення ConsumerValidated, стан замовлення змінюється на CONSUMER\_VALIDATED, а якщо OrderDetailsValidated - на ORDER\_DETAILS\_VALIDATED. Отримавши друге повідомлення, сервіс Order привласнює замовлення стан VALIDATED.

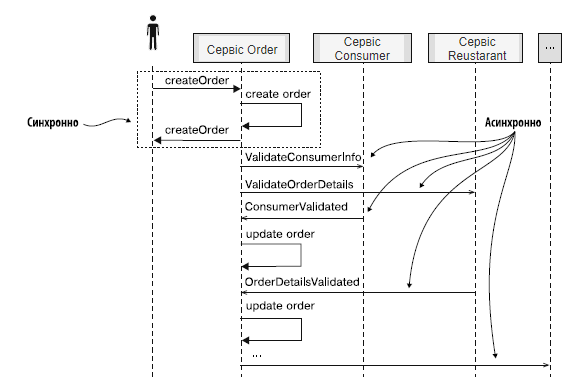


Рис.4 Сервіс створює замовлення, не звертаючись до жодного іншого сервісу. Далі він асинхронно перевіряє замовлення через інші сервіси

**Висновки.** У роботі було запропоновано проаналізовано недоліки синхронного методу комунікації у клієнт-сервісній архітектурі та запропоновано способи впровадження асинхронності для оптимізації клієнт-сервісної взаємодію. Було надано структурні схеми та діаграму діяльності яка демонструє реалізацію запропонованих методів. Ця схеми можуть застосуватися при проектуванні архітектури мікросервісних систем для побудови ефективної взаємодії між ними.

**Література**

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 – 304 с.
3. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions / Gregor Hohpe, Bobby Woolf – Addison-Wesley, 2004 – 736 p.
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 с.
6. Martin Fowler – Microservices – Режим доступу: <http://martinfowler.com/articles/microservices.html>
7. I. Nadareishvili. Microservice Architecture: Aligning Principles, Practices, and Culture / I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen – O’Reilly Media, 2016 – 146 с.
8. Introduction to microservices. – Режим доступу: <https://nginx.com/blog/introduction-to-microservices/>
9. Using an API Gateway. – Режим доступу: [https://nginx.com/blog/building-](https://nginx.com/blog/building-microservices-using-an-api-gateway/)

[microservices-using-an-api-gateway/](https://nginx.com/blog/building-microservices-using-an-api-gateway/)

1. Service Discovery. – Режим доступу[: https://nginx.com/blog/service-discovery-in-a-microservices-architecture/](https://nginx.com/blog/service-discovery-in-a-microservices-architecture/)