УДК 339.1

Кузоров Євгеній Олегович
Аспірант кафедри прикладних інформаційних систем, *orcid.org/0000-0001-5848-8537*
Київський національний університет ім. Тараса Шевченка, Київ

ПОБУДОВА МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ДЛЯ ВЕЛИКИХ ІНФОРМАЦІЙНИХ СИСТЕМ

Анотація. *Розглянуто переваги мікросервісної архітектури. Стрімкий розвиток технологій і численна кількість користувачів глобальної мережі Інтернет змушують розробляти і впроваджувати нові методики щодо побудови складних інформаційних систем. Розглянуто можливість мікросервісної архітектури оптимізувати проектування надвеликих прикладних систем, у тому числі командну роботу розробників, а також комунікації між ними, зменшити ціну розробки та впровадження і супроводження програмних комплексів, з гнучкою властивістю до масштабованості і модифікації, альтернативою монолітній у випадку використання клієнт-серверного підходу. Розглянуто методи побудови мікросервісних систем та розбиття складних монолітних систем на мікросервіси. Показано, як мікросервісний підхід допомагає знизити вартість підтримки великих програмних комплексів.*

Ключові слова: архітектура програмного забезпечення; мікросервісна архітектура; мікросервіс; автоматизація інфраструктури

Вступ

«Мікросервіси» – це архітектурний стиль, за якого застосунок створюють як сукупність сервісів, кожен з яких функціонує у своєму власному процесі та комунікує з рештою процесів, використовуючи такий механізм, як HTTP. Подібні сервіси будують завдяки потребам користувача і розгортають незалежно з використанням повністю автоматизованого середовища. Вони можуть бути написані з використанням різних мов програмування високого рівня і технологій зберігання даних.

Зростаюча складність систем, навантаження на окремі її частини, а також необхідність відповідати вимогам бізнесу потребує нових, гнучкіших підходів до проектування програмного забезпечення та інфраструктури для його розгортання.

Мікросервісна архітектура підходить для процесів безперервного постачання. Мікросервісну архітектуру спрямовано на створення одного чи кількох застосунків, на відміну від сервіс-орієнтованої архітектури (архітектурний шаблон програмного забезпечення, модульний підхід до розроблення програмного забезпечення, оснований на використанні розподілених компонентів, забезпечених стандартними інтерфейсами для взаємодії застосунків за визначеними протоколами). Відомо низку проектів, що використовують мікросервісну архітектуру. Їх порівнюють з потужним застосунком, що побудований як єдине ціле. Застосунки на рівні підприємств базовано з трьох основних підходів: дружній інтерфейс (HTML сторінки, скрипти мовою Java), база даних (БД) (як правило, реляційна) та сервер. Серверна частина обробляє HTTP запити, оновлює дані в БД, заповнює HTML сторінки, які потім мають бути відправлені браузеру клієнта. Зміни в системі потребують розгортання нової версії серверної частини програми. Бізнес-логіку з оброблення запитів виконують в єдиному місці, при цьому можна скористатися наявними можливостями мови програмування для поділу прикладного застосунку на класи, функції та області дії імен. Можна масштабувати великі застосунки горизонтально через запуск кількох фізичних серверів за балансувальником навантаження.

Багато досліджень проведено щодо мікросервісної архітектури, але проблема коректного виділення сервісів та розподілу бізнес-логіки, водночас із стандартизацією транспорту та протоколу взаємодії між ними все ще залишається відкритою проблемою [6].

Монолітні застосунки можуть бути успішними, але все більше користувачів мають проблеми через зростаючу складність розгортання та підтримку застосунків, розроблених як «єдине ціле», особливо з урахуванням того, що застосунки розгортають із використанням хмарних рішень. Із розгортанням за часом, стає важче зберігати модульну структуру, змінювати логіку одного модуля, що, у свою чергу, впливає на програмний код інших модулів, тому масштабувати доводиться застосунок у цілому.

Ці незручності призвели до виникнення мікросервісної архітектури: побудови застосунків у

вигляді набору сервісів, тобто архітектурний стиль, за якого єдиний застосунок будують як сукупність невеликих сервісів, кожен з яких функціонує у своєму власному процесі та комунікує з рештою.

Мета статті

Основна мета полягає в тому, щоб проаналізувати основні ідеї і принципи мікросервісної архітектури, виділити проблеми та особливості проектування таких систем, оцінити їх перспективи розвитку у контексті підвищення складності та навантаження на програмні комплекси.

Виклад основного матеріалу

Розподілення через сервіси

Мікросервісна архітектура використовує бібліотеки, але її основний спосіб поділу – розподіл на сервіси. Визначають бібліотеки як компоненти, які підключають до програми і викликають її в тому ж процесі, в той час як сервіси, це компоненти, що виконують окремо і комунікують між собою через веб-запити або виклики віддалених процедур.

Головна причина використання сервісів замість бібліотек – це незалежне розгортання. Якщо розробляємо застосунок, підтримуваний кількома бібліотеками, які працюють в одному процесі, будь-яка зміна в цих бібліотеках призводить до повторного розгортання всієї програми. Але якщо застосунок розподілено на кілька сервісів, то зміни, що зачіпають будь-який з них, вимагатимуть розгортання тільки цієї частини. Це дозволяє зменшити витрати на розгортання та краще розподілити потужності між тими частинами програмного комплексу, які потребують цього в конкретний момент часу. Також такий підхід до розроблення дозволяє автоматизувати масштабування мікросервісів залежно від добових або сезонних піків навантаження та зменшити використання персоналу для подібних налаштувань.

Організація навколо потреб бізнесу

Коли великі застосунки розподіляють на частини, менеджмент фокусують на технологіях, що приводить до утворення *UI* (user interface) *команди* (команди, яка працює над дружнім інтерфейсом користувача), *команди, що працює над серверною частиною*, і *команди, яка розробляє БД*. Коли команди розділено подібним чином, навіть невеликі зміни забирають багато часу через необхідність крос-командної взаємодії. Це призводить до того, що команди розміщують бізнес-логіку на тих рівнях, до яких вони мають доступ. Мікросервісний підхід дозволяє виділити з*они відповідальності команд*, а також оптимізувати кількість їх членів, що позитивно відображається на якості розроблених програмних комплексів і спрощує комунікації між різними командами, а, отже, дозволяє якісніше налаштувати бізнес-процеси та підвищити продуктивність розробки в цілому [9].

Продукти, а не проекти

Більшість компаній із розроблення програмного забезпечення (ПЗ) використовують проектну модель, в якій метою є розробка певної функціональності, яку після цього вважають завершеною. Після завершення її передають команді підтримки і проектну команду ліквідовують.

Прихильники мікросервісів цураються цієї моделі, стверджуючи, що команда має володіти продуктом протягом усього його життєвого циклу. Це приводить до того, що розробники регулярно спостерігають за тим, як програмний продукт під час експлуатації себе поводить, тому що приходиться надалі підтримувати його дороблення.

Програмний продукт є набором функційних можливостей, які розширюють під нові задачі користувачі з метою збільшення їх бізнес-можливостей.

Звичайно, цього можна також досягти і у випадку з великим прикладним застосунком, але висока «гранулярність» сервісів спрощує встановлення відносин між розробниками сервісу і його користувачами [11].

Складні обробники та неінтелектуальні протоколи обміну даними

При проектуванні комунікацій між сервісами часто механізми передачі даних містять значну частину логіки. Прикладом тут є Enterprise Service Bus (ESB). ESB-продукти часто охоплюють можливості передачі, оркестрування і трансформацію повідомлень.

Мікросервісний підхід потребує альтернатив: складні приймачі повідомлень і неінтелектуальні канали передачі даних. Застосунки, побудовані з використанням мікросервісної архітектури, мають бути настільки незалежними та сфокусованими, наскільки це можливо: вони містять власну доменну логіку і виступають як фільтри в класичному Unix, тобто отримують запити, застосовують логіку і відправляють відповідь. При такому підході відкриваються великі можливості вибору архітектурних рішень і транспортних протоколів. Також зміщення зони відповідальності з транспортного протоколу на мікросервіси дозволяє використати такі архітектурні рішення, як CQRS, що в свою чергу, при використанні паттерну AMQP Message Bus, знімає майже всі обмеження при розподіленні монолітного програмного комплексу на мікросервіси. Найменшою неподільною одиницею в такому випадку стає метод, а розгортання та

масштабування такого комплексу можливе на будь-якому апаратному забезпеченні з будь-якою топологією мережі та якісно проявляє себе при високих навантаженнях.

Децентралізоване керування

Одним із наслідків централізованого керування є тенденція до стандартизації використовуваних платформ. Досвід показує, що такий підхід обмежує вибір, адже велика кількість проблем вимагає різноманітних підходів до їх вирішення. Краще використовувати коректний інструмент для кожної конкретної роботи. Розподіляючи застосунок на сервіси, маємо вибір, як побудувати кожен з них. Можна використовувати Node.js для простих сторінок зі звітами, C ++ також можна використовувати.

Команди, які розробляють мікросервіси, також мають інший підхід до стандартизації. Замість того, щоб використовувати набір визначених стандартів, вони надають перевагу ідеї побудови корисних інструментів, які інші розробники можуть використовувати для вирішення схожих проблем. Ці інструменти, як правило, відокремлені з програмного коду одного з проектів, і розподіляються між різними командами. Тепер, коли git і github стали де-факто стандартною системою контролю версій, практики відкритого коду стають широко використовуваними у внутрішніх проектах компаній.

Зростання платформи JVM – це один з останніх прикладів змішування мов у межах єдиної платформи. Перехід до більш високорівневих мов для отримання переваг, пов'язаних із використанням високорівневих абстракцій, є поширеною практикою протягом багатьох років. Багато монолітних застосунків не вимагають такого рівня оптимізації продуктивності і високорівневих можливостей DSL-подібних мов. Замість цього «моноліти», як правило, використовують єдину мову і схильні до обмеження кількості використовуваних технологій. Мікросервісний підхід дозволяє створювати кожну частину програмного комплексу з використанням технологій, що найкраще підходять до розв'язання конкретної прикладної задачі. Не існують уже обмеження при виборі СУБД або сховища даних для інформаційно-аналітичного забезпечення. Найбільш навантажені вузли прикладної системи можуть бути розроблені з використанням таких мов програмування, як C# або C++, що мають на порядок вищу швидкість [7; 10].

Автоматизація інфраструктури

Методики автоматизації інфраструктури еволюціонували за останні роки. Еволюція хмарних рішень і AWS, зокрема, зменшили операційну складність побудови, розгортання і функціонування мікросервісів [1 – 5].

Багато продуктів і систем, що використовують мікросервісну архітектуру, побудовано командами з великим досвідом в Continuous Delivery і Continuous Integration [1 – 5]. Команди, що будують застосунки подібним чином, інтенсивно використовують різні методики автоматизації інфраструктури.

Для виконання кожного кроку автоматичного тестування застосунок розгортають у окремому середовищі, для чого використовують автоматичне розгортання.

Розвиток методів і інструментів автоматичного розгортання та коду дозволяє підвищити періодичність доставки нового функціоналу як окремих сервісів, так і програмних комплексів в цілому, а автоматизоване тестування на етапі розгортання зменшує кількість помилок, що потрапляють до основних серверів, а, отже, збільшують продуктивність команд.

Проектування для відмови

Наслідком використання сервісів як компонентів є необхідність проектування застосунків так, щоб вони могли працювати при відмові окремих сервісів. Будь-яке звернення до сервісу може не спрацювати через його недоступність. Це є недоліком мікросервісів порівняно з монолітом, оскільки вносить додаткову складність програмному застосунку. Сервіси можуть відмовити в будь-який час, дуже важливо мати можливість швидко виявити неполадки, і, якщо можливо, автоматично відновити працездатність сервісу. Мікросервісна архітектура робить великий акцент на моніторингу застосунків у режимі реального часу, перевірянні як технічних елементів, так і бізнес-метрик [3; 4].

Це особливо важливо у випадку з мікросервісною архітектурою, тому що розподіл на окремі процеси і комунікації через події призводить до несподіваної поведінки. Моніторинг украй важливий для виявлення небажаних випадків такої поведінки і швидкого їх усунення [6; 7].

Висновки

Останнім часом мікросервісний підхід до розробки прикладного програмного забезпечення набув великої популярності у компаніях, що розробляють складні, високонавантажені прикладні системи. Використовуючи їх досвід на практиці, а також, розробляючи власні стандарти та протоколи обміну повідомленнями при коректній організації доменної моделі, такий підхід може бути успішно застосований до побудови будь-якої системи з клієнт-серверною архітектурою та дозволити покращити показники відмовостійкості та тривіальності підтримки такої системи.

Автоматизовані системи розгортання, тестування та моніторингу підвищать продуктивність команд, та дозволять змістити акцент їх уваги до, власне, процесу розробки програмного забезпечення, а не його налагоджування чи регресійного тестування в пошуках невеликих помилок, які складно віднайти в монолітних програмних застосунках.

Список літератури

1. *Сравнение службы приложений, облачных служб и виртуальных машин Azure. [Електронний ресурс]. Режим доступу: URL: https://azure.microsoft.com/ru- ru/documentation/articles/choose-web-site-cloud-service-vm/*
2. *Знакомство с Windows Azure. Для ИТ-специалистов // Таллоч М.; пер. с нгл. – М.: ЭКОМ Паблишерз, 2014. – 154 с.*
3. *Подробное описание возможностей разработки с Microsoft Azure Cloud Services. [Електронний ресурс]. Режим доступу: URL: http://habrahabr.ru/company/microsoft/blog/242543/*
4. *Гибридное облако Microsoft: Руководство по типовым решениям. [Електронний ресурс]. Режим доступу: URL: https://www.microsoftvirtualacademy.com/ru/training-courses/- microsoft--8242?l=frbVWKWCB_8604984382*
5. *Библиотека технической документации по Azure. [Електронний ресурс]. Режим доступу: URL: https://msdn.microsoft.com/ru-ru/library/dn578280.aspx*
6. *ISO / IEC 17788 Information technology – Distributed application platforms and services – Cloud computing – Overview and vocabulary.*
7. *NIST SP 500-291 NIST Cloud Computing Standards Roadmap.*
8. *ISO / IEC 17789 Information technology – Cloud computing – Reference architecture.*
9. *Тімінський О.Г. Виникнення, розвиток і проблеми інформаційних технологій управління [Текст] / О.Г. Тімінський // Управління розвитком складних систем. – 2016. – No 25. – С. 86 –90.*
10. *Доманецька І.М. Дослідження впливу моделі даних на ефективність роботи високонавантажених систем [Текст] / І. М. Доманецька, Я. В. Матейко, О. В. Федусенко, В. М. Хроленко, А. О. Федусенко // Управління розвитком складних систем. – 2014. – № 17. – С. 81 –89.*
11. *Сухонос М.К. Анализ проблем управления IT – проектами как активными объектами. / М.К. Сухонос, И.В. Белецкий, А.Ю.Старостина, А.А. Богославец, Т.А. Медведева // Управління розвитком складних систем. – 2016. – №27. – С. 84 – 91.*

Стаття надійшла до редколегії 14.04.2017

Рецензент: д-р екон. наук, проф. В.Л. Плескач, Київський національний університет ім. Тараса Шевченка, Київ.

Кузоров Евгений Олегович
Аспирант кафедры прикладных информационных систем, *orcid.org/0000-0001-5848-8537*
Киевский национальный университет им. Тараса Шевченко, Киев

ПОСТРОЕНИЕ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ ДЛЯ БОЛЬШИХ ИНФОРМАЦИОННЫХ СИСТЕМ

Аннотация. *Рассмотрены преимущества микросервисной архитектуры. Стремительное развитие технологий и многочисленное количество пользователей глобальной сети Интернет заставляют разрабатывать и внедрять новые методики по построению сложных информационных систем. Рассмотрена возможность микросервисной архитектуры оптимизировать проектирование сверхбольших прикладных систем, в том числе командную работу разработчиков, а также коммуникации между ними, уменьшить цену разработки, внедрения и сопровождения программных комплексов, с гибким свойством к масштабируемости и модификации, альтернативой монолитной в случае использования клиент-серверного подхода. Рассмотрены методы построения микросервисных систем и разбиение сложных монолитных систем на микросервисы. Показано, как микросервисный подход помогает снизить стоимость поддержки больших программных комплексов.*

Ключевые слова: архитектура программного обеспечения; микросервисная архитектура; микросервис; автоматизация инфраструктуры

Kuzorov Evgeniy Olegovich
Postgraduate student of applied information system department, *orcid.org/0000-0001-5848-8537*
Kyiv's Taras Shevchenko National University, Kyiv

DEVELOPMENT OF MICROSERVICE ARCHITECTURE FOR LARGE SYSTEMS

Abstract. *This article discusses the benefits microservice architecture. The rapid development of technology and alarge number of users for the Internet are forced to develop and implement new techniques of building complex information systems. Considered how microservice architecture helps to optimize the design of very large scalable applications, including teamwork development, and communication between them, reduce the price of development and implementation and maintenance of software systems with flexible properties to scalability and modifications, alternative monolithic when using client-server approach. Considered mikroservise methods of partitioning systems and complex monolithic systems. Showed how mikroservice approach helps to reduce the maintenance cost of large software systems. Correct usage of microservice architecture can onptimize business and development processed in company which relates to large information system or involved to mainteinence of one. Also it helps to grow information system horizontally either than vertically, which is the best practice used by lots of large companies. Microservice approach became the mainstream in architecture of large systems.*

Keywords: software architecture; mikroservis architecture; automated deployment; mikroservis; automation infrastructure

References

1. Comparison of application services, cloud services and Azure virtual machines. (2015). Retrieved from URL: https://azure.microsoft.com/en-us/documentation / articles / choose-web-site-cloud-service-vm /

2. Talloche M .; Per. With. (2014). Introduction to Windows Azure. For IT professionals. EKOM Publishers, 154.

3. Detailed description of the development possibilities with Microsoft Azure Cloud Services. Retrieved from URL: http://habrahabr.ru/company/microsoft/blog/242543/

4. Microsoft Hybrid Cloud: A Guide to Typical Solutions.Retrieved from URL: https://www.microsoftvirtualacademy.com/ru/training-courses/- microsoft--8242?l=frbVWKWCB_8604984382

5. Library of technical documentation for Azure. Retrieved from URL: https://msdn.microsoft.com/ru-ru/library/dn578280.aspx

6. ISO / IEC 17788 Information technology – Distributed application platforms and services – Cloud computing – Overview and vocabulary.

7. NIST SP 500-291 NIST Cloud Computing Standards Roadmap.

8. ISO / IEC 17789 Information technology – Cloud computing – Reference architecture.

9. Timinsky A.G. (2016). Origin, development and problems of information technology enterprise management. Management of development of difficult systems, 25, 86–90.

10. Domanetska, Irina, Matejko, Yaroslav, Fedusenko, Elena, Hrolenko, Vladimir, Fedusenko, Anatoly. (2014). The influence of these models the performance of highly loaded systems, 17, 81-89

11. Sukhonos, M.K., Biletskiy, I.V., Bohoslavets, A.A., Miedviedieva, T.O. (2016). Analysis of management issues of IT projects as active objects. Management of Development of Complex Systems, 27, 84–91.

Посилання на публікацію

APA *Kuzorov, E.O. (2017). Development of microservice architecture for large systems. Management of Development of Complex Systems, 30, 116–120. [in Ukrainian]*

ГОСТ *Кузоров Є.О. Побудова мікросервісної архітектури для великих інформаційних систем [Текст] // Є.О. Кузоров // Управління розвитком складних систем. – 2017. – № 30. – С. 116 – 120.*