DOI 10.31891/2307-5732-2024-335-3-55
УДК 004.273

ГРИГА СЕРГІЙ
Хмельницький національний університет
https://orcid.org/0009-0002-4409-4531
e-mail: grigha71@gmail.com
ПРАВОРСЬКА НАТАЛІЯ
Хмельницький національний університет
https://orcid.org/0000-0001-6001-3311
e-mail: margana2000007@gmail.com

МЕТОДИ РЕАЛІЗАЦІЇ МІКРОСЕРВІСНИХ АРХІТЕКТУР: ПЕРЕВАГИ ТА НЕДОЛІКИ, ВПРОВАДЖЕННЯ ТА ТЕСТУВАННЯ ПРИ РОЗРОБЦІ ПРОГРАМНИХ ЗАСТОСУНКІВ

Розробка мікросервісної архітектури є складним та комплексним процесом, що потребує глибокого розуміння багатьох аспектів її реалізації. У даній статті було розглянуто дану архітектуру, її переваги та недоліки, у яких випадках її доцільно використовувати, деякі важливі методи, технології, патерни та підходи до тестування. Особливий акцент робиться на важливість коректного розбиття функціоналу системи на мікросервіси, що можна ефективно провести за допомого підходу Еріка Еванса «domain-driven design» та використанню граничних контекстів.

Серед важливих технологій, які було розглянуто, відносяться контейнеризація та оркестрація контейнерів, що при використанні у зв'язці дозволяють помістити кожен мікросервіс у окремий контейнер які можна швидко та легкого запустити на різних операційних системах без необхідності попереднього налаштування їхніх параметрів, допомагає ефективно використовувати і розподіляти ресурси між ними, а також спрощує процес управління.

Ефективність, та ряд інших важливих параметрів програмного продукту, напряму залежать від обраних патернів проектування та якості їхньої реалізації, наприклад, патерн вимикач при коректній розробці та налаштуванні може забезпечувати стабільну роботу всієї системи навіть за умови некоректної роботи окремих мікросервісів, що тимчасово обмежуються і стають недоступними для виклику до виправлення всіх помилок, чи шлюз та патерн «Sega», що дозволяють забезпечити маршрутизацію запитів та підтримувати коректну роботу кожного мікросервісу за умов виникнення різного виду помилок.

Тестування мікросервісної архітектури є надзвичайно важливим та комплексним завданням, від якого залежить вся подальша робота системи, тому було визначено, що його доцільно проводити одночасно за допомогою декількох різних методів тестування, а саме юніт-тестів, інтегрованого та наскрізного тестування, що у сукупності дозволяють на різних рівнях та етапах розробки перевірити коректність роботи, як усієї системи, так і її окремих елементів, що дозволяє швидко виявити помилки та усунути їх до розгортання програмного забезпечення.

В результаті досліджень було визначено та проаналізовано переваги та недоліки мікросервісної архітектури, доцільність її використання та ряд сучасних технологій, патернів проектування та способів тестування, що в сукупності допоможуть уникнути ряду проблем при розробці та створити якісну мікросервісну архітектуру.

Ключові слова: мікросервіси, мікросервісна архітектура, технології розробки програмних застосунків, патерни проектування, конструювання програмного забезпечення, тестування.

HRYHA SERHII , PRAVORSKA NATALYA
Khmelnitsky national university, Ukraine

METHODS FOR IMPLEMENTING MICROSERVICE ARCHITECTURES: ADVANTAGES AND DISADVANTAGES, IMPLEMENTATION AND TESTING IN THE DEVELOPMENT OF SOFTWARE APPLICATIONS

Developing a microservice architecture is a complex and comprehensive process that requires a deep understanding of many aspects of its implementation. This article discusses this architecture, its advantages and disadvantages, when it is appropriate to use it, and some important methods, technologies, patterns, and approaches to testing. Particular emphasis is placed on the importance of correctly dividing the system functionality into microservices, which can be effectively done with the help of Eric Evans' domain-driven design approach and the use of boundary contexts.

Among the important technologies that were considered were containerization and container orchestration, which, when used in conjunction, allow each microservice to be placed in a separate container that can be quickly and easily run on different operating systems without the need to pre-configure their parameters, helps to efficiently use and allocate resources between them, and simplifies the management process.

Efficiency, and a number of other important parameters of a software product, directly depend on the selected design patterns and the quality of their implementation, for example, the switch pattern, if properly designed and configured, can ensure stable operation of the entire system even if individual microservices are not working correctly, temporarily limited and become unavailable for calling until all errors are corrected, or the gateway and the Sega pattern, which allow you to route requests and maintain the correct operation of each microservice in the event of a problem.

Testing of microservice architecture is an extremely important and complex task, on which all further operation of the system depends, so it was determined that it should be carried out simultaneously using several different testing methods, namely unit tests, integrated and end-to-end testing, which together allow at different levels and stages of development to check the correctness of the operation of both the entire system and its individual elements, which allows you to quickly identify errors and eliminate them before deploying the software.

As a result of the research, the advantages and disadvantages of microservice architecture, the feasibility of its use, and a number of modern technologies, design patterns and testing methods were identified and analyzed, which together will help to avoid a number of problems in the development and create a high-quality microservice architecture.

Keywords: microservices, microservice architecture, software application development technologies, design patterns, software design, testing.

Постановка проблеми

На сьогоднішній день розробка програмних продуктів відбувається в умовах стрімкого розвитку різноманітних технологій, а одним із важливих аспектів є підхід до розробки архітектури програмного забезпечення. У даному контексті використання мікросервісної архітектури набуває все більшої популярності серед розробників, що дозволяє зосередитись на розділені функціональних можливостей програми на невеликі за розміром та незалежні сервіси.

Однак, як і всі інші архітектури, разом із рядом переваг виникають різноманітні виклики та проблеми, що потребують швидкого вирішення. До таких проблем можна віднести підхід до розбиття функціоналу на мікросервіси, методи їх взаємодії, роботи, підвищена складність розробки системи, проблеми з розгортанням та тестуванням.

З даних причин надзвичайно важливим є чітке розуміння доцільності використання мікросервісної архітектури, її основних переваги та недоліки при використанні, сучасні методи і підходи до її розробки, технології, патерни проектування та найбільш ефективні методи тестування. Лише комплексне розуміння даних аспектів всіх цих аспектів дозволить мінімізувати негативний вплив недоліків даної архітектури, покращить якість всієї системи.

Аналіз останніх джерел

На сьогоднішній день існує велика кількість досліджень, що пов’язані з мікросервісною архітектурою. Більшість із них акцентує увагу на конкретних проблемах архітектури, методах, інструментах та технологіях, що використовуються при розробці мікросервісної архітектури, а також різноманітних аспектах, що можуть напряму впливати на якість розробки. Однак, вичерпні дослідження мікросервісної архітектури майже повністю відсутні.

У дослідженні [1] дослідники акцентують увагу на тому, що мікросервісна архітектура має ряд серйозних недоліків та викликів, які варто ретельно оцінити на етапі проектування та визначити потенційні можливості їхнього вирішення. В роботі проводиться систематичний огляд літератури та відбір ряду статей, що використовувались як основа для подальшого дослідження. На їхній базі дослідники змогли визначити дев’ять основних категорій викликів, сорок підкатегорій і потенційні напрями їх вирішення.

У дослідженнях [2] та [3] дослідники аналізують сучасні підходи ефективного використання контейнерів у мікросервісній архітектурі включно із застосуванням хмарних технологій, тоді як у дослідженні [4] на основі опитування експертів та порівняння їхніх результатів з існуючими дослідженнями, для визначення можливостей мікросервісної та безсерверної архітектури у порівнянні з монолітною архітектурою. Результати дослідження вказують на те, що дані архітектури можуть ефективно вирішувати ряд проблем, що включають масштабованість і продуктивності.

На основі проаналізованих джерел можна зробити висновок, що дослідження мікросервісної архітектури здійснюється у багатьох напрямках, але вони не є повністю вичерпними і потребують подальших детальних досліджень у даних напрямках.

Виклад основного матеріалу

Мікросервісна архітектура – це підхід до розробки програмного забезпечення, що передбачає створення застосунку у вигляді набору невеликих та незалежних компонентів, що мають назву мікросервіси, кожен з яких виконує конкретну функцію, що працює повністю автономно та комунікує з іншими сервісами за допомогою легких механізмів, наприклад, HTTP.

Як будь-яка інша архітектура, вона має як переваги, так і серйозні недоліки. До переваг використання мікросервісної архітектури відносять:

– можливість незалежного розгортання компонентів програмного продукту;

– можливість залучення великої кількості незалежних команд розробників, що можуть використовувати різні мови програмування чи інші технології;

– можливість повторного використання коду;

– можливість незалежного маштабування кожного сервісу;

– високий рівень ізоляції неполадок, що дозволяє уникати поломки всієї системи.

До недоліків мікросервісної архітектури відносять:

– менша швидкість виконання програмного коду у порівнянні з іншими архітектурами;

– збільшену складність розробки програмного продукту ;

– висока складність підтримки сервісів через їхню велику кількість ;

– висока вартість розробки ;

– необхідність достатньо високого рівня знань розробників для отримання якісного результату.

Окремо варто визначити доцільність використання мікросервісної архітектури в кожному конкретному випадку. Це особливо важливий етап розробки, адже дана архітектура, хоч і має серйозні переваги над іншими, недоліки є не менш критичними. В загальному випадку можна виділити три основних причини використання мікросервісної архітектури:

– програмний продукт є об’ємним за розміром і є плани по його майбутньому розвитку при наявності декількох функцій, що можуть потребувати незалежного масштабування;

– програмний продукт передбачає використання динамічних мов програмування та є великим за об’ємом;

– команда розробників складається із великої кількості спеціалістів, що використовують різні мови програмування та технології.

Одним із перших серйозних викликів розробки мікросервісної архітектури є розбиття функціональних можливостей системи на окремі мікросервіси. Дати однозначну відповідь на дане запитання неможливо, адже не існує жодної кількісної величини, адже вони не враховують контексту бізнес-функцій. Для вирішення поставленого завдання доцільно використовувати підхід Еріка Еванса «domain-driven design» (DDD), що фокусується на вивчені предметної області бізнесу чи окремих процесів.

В основі даного підходу знаходиться тісна співпраця замовника та розробника, що ставить собі за мету розуміння всіх процесів обома сторонами, в основі якого лежить розподіл застосунку на домени.

Домен являє собою область знань чи діяльності, що описує ряд завдань. Наприклад, у секторі фінансових послуг доменом може виступати управління платежами чи кредитні операції. Кожен домен складається із субдоменів, що є множиною конкретних завдань чи бізнес аспектів, наприклад, у домені управління платежами можуть буди субдомени обробки платежів, управління транзакціями, перевірка балансу. У свою чергу субдомени можна робити на менші елементи:

– основні субдомени – критично важливі для бізнесу та забезпечують ключові функції;

– допоміжні субдомени – це набір функцій, що підтримують основні субдомени та забезпечують їхню стабільну роботу, але вони не є критичними;

– загальні субдомени – це набір неунікальних, загальних функцій чи сервісів, що використовуються в різних системах.

На базі основних, допоміжних і загальних субдоменів здійснюється подальше створення мікросервісів. Наприклад, компанія займається доставкою посилок, тому її доменом буде доставка та менеджер доставок. Для забезпечення їхньої коректної роботи у моделі використовуються субдомени.

У випадку даного прикладу субдоменами виступає створення маршруту, підбір персоналу, відслідковування посилки, повернення посилки, оплата, технічна підтримка, користувачі та персональні акції. Варто враховувати, що розробку моделі доцільно здійснювати з врахуванням майбутнього функціоналу, наприклад персональні акції можуть бути реалізовані, як майбутні оновлення застосунку. Відповідну модель зображено на рисунку 1.

Але варто розуміти, що одне лише розбиття функціоналу на окремі домени різних рівнів не може в повному обсягу визначити чіткі межі їхніх завдань. Для вирішення даної проблеми у DDD існує поняття граничного контексту.

Рисунок 1. Модель розбиття системи доставки товарів на домени

Граничний контекст (Bounded Context) – це логічна мережа, що визначає область відповідальності конкретної моделі в мережах системи. Основним призначенням граничного контексту є визначення: де починається і закінчується конкретна модель, із визначенням чітких меж її термінології, правил та поведінки. У «Domain-Driven Design» використання даного контексту ставить собі за мету розподіл складних доменів на окремі ізольовані та керовані частини з метою уникнення конфліктів між моделями. Розбиття на контексти здійснюється на основі аналізу функціональних відмінностей кожного домену.

На відміну від моделі на основі доменів, яка представляє собою відображення предметів реального світу, але у різних частинах системи, не є доцільним використання однакових представлень ідентичних речей. Наприклад, домени «Менеджер доставки» та «Доставка» містять дані про посилку, але «Менеджер доставки» потребує лише даних про кінцеве місце призначення та місце відправки, тоді як домен «Доставка» додатково

потребує даних про клієнта для його ідентифікації. Такий підхід до проектування дозволяє зменшити складність всієї системи та дозволить робити зміни в одній моделі без вплину на іншу. Приклад розбиття на граничні контексти можна побачити на рисунку 2.

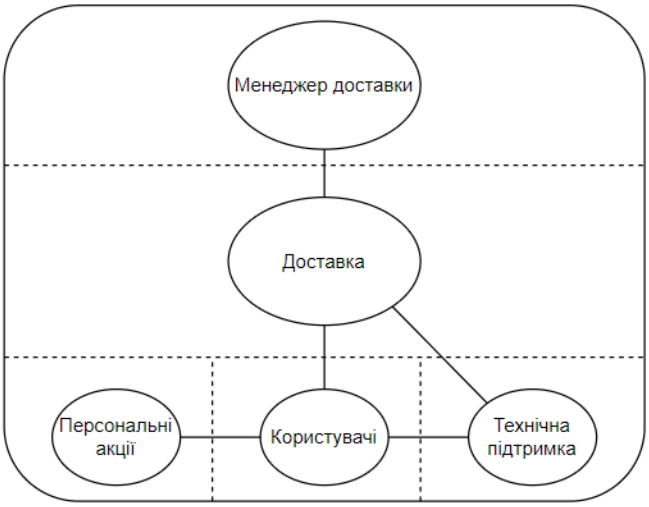

Рисунок 2. Граничні контексти

Технології для створення мікросервісів

У мікросервісній архітектурі важливо забезпечити максимальну ізольованість кожного мікросервіса від інших, високий рівень портативності та можливість швидкого розгортання. Для забезпечення цих характеристик доцільно використовувати контейнери.

Контейнер – це легкі та ізольовані середовища, що використовуються для запуску декілька ізольованих просторів на одному ядрі, що включають у себе програмний код, середовище для його виконання, інструменти, бібліотеки та налаштування. З їх допомогою будь-який мікросервіс може бути запущеним у різних середовищах розгортання і при цьому працювати коректно.

Сучасне програмне забезпечення для створення контейнерів, зазвичай, складається з трьох етапів:

– етап 1 – створення файлів з налаштуваннями та файлу з програмним кодом та необхідними додатковими компонентами;

– етап 2 – доставка отриманих файлів на сервер, де буде розгорнуто програму;

– етап 3 – запуск отриманих файлів для створення контейнера.

Головною проблемою при використання контейнерів є забезпечення ефективного управління контейнерами. Для вирішення даної проблеми доцільним є застосування процесу оркестрації контейнерів.

Оркестрація контейнерів – це процес автоматизації розгортання, управління та забезпечення взаємодії між контейнерами, що дозволяє розробникам керувати кластером контейнерів. Оркестрація дозволяє забезпечити ефективний розподіл навантаження між контейнерами, високий рівень доступності та відмовостійкості, а також забезпечити комунікацію між контейнерами.

Оркестрація контейнерів виконується за допомогою відповідних платформ та складається з наступних етапів:

– визначення конфігурації сервісів, що будуть запущені в контейнерах;

– налаштування мережі для ефективної взаємодії між контейнерами;

– запуск процедури оркестрації;

– моніторинг стану системи.

Архітектурні патерни проектування

Для створення ефективної архітектури важливо вірно використовувати різноманітні патерни проектування. Для мікросервісної архітектури виділяють ряд важливих патернів, що впливають на різні аспекти системи. Першим важливим патерном є вимикач.

Вимикач (Circuit Breaker) – це патерн проектування, що використовується в мікросервісній архітектурі з метою покращення стабільності роботи та стійкості системи. Основною метою його використання є запобігання виклику сервісів, що не можуть відправити відповідь.

Принцип роботи даного патерну передбачає наявність трьох станів для кожного мікросервіса, що можуть накладати обмеження:

– першим є закритий стан, що передбачає, що всі запити до мікросервіса є успішними, а сервіс працює коректно;

– другим є відкритий стан. Він призначався мікросервісу у випадках появи різних збоїв, наприклад, занадто великий час очікування відповіді, що допомагає запобігти додатковому навантаженню на систему.

– третій стан називається напіввідкритим. Мікросервіс переходить у даний стан коли він знаходиться у відкритому стані певний період часу. Він дозволяє виконання обмеженої кількості запитів для перевірки коректності роботи сервісу. Якщо збої тривають, мікросерсів знову переходить у відкритий стан, а в іншому випадку у закритий стан.

Таким чином окремі мікросервіси, у яких виникають збої будуть обмежуватись, але всі інші функціональні можливості системи працюють коректно.

Ще одним важливим патерном проектування є шлюз (API Gateway). Шлюз – це патерн проектування, який передбачає наявність сервісу, що виступає в ролі єдиного можливого входу всіх зовнішніх запитів до мікросервісів, що дозволяє забезпечити маршрутизацію запитів, обробку помилок, безпеку даних та об'єднання відповідей від кількох сервісів в одну. На рисунку 3 наведено схему використання шлюзу.

Окрім виконання своєї основної задачі шлюз також можна використовувати для виконання додаткових функцій, наприклад, аутентифікацію користувачів, логування та моніторинг, визначення списку дозволених та заблокованих IP-адрес.

Однією із ключових проблем у мікросервісній архітектурі є реалізація розподілених операцій, що охоплюють відразу декілька сервісів, що може бути складно особливо у випадках наявності окремих баз даних для кожного сервісу. Для вирішення цієї проблеми розробники використовують патерни взаємодії сервісів, що розподіляють операцію на декілька локальних транзакцій, прикладом якого є патерн «Sega».

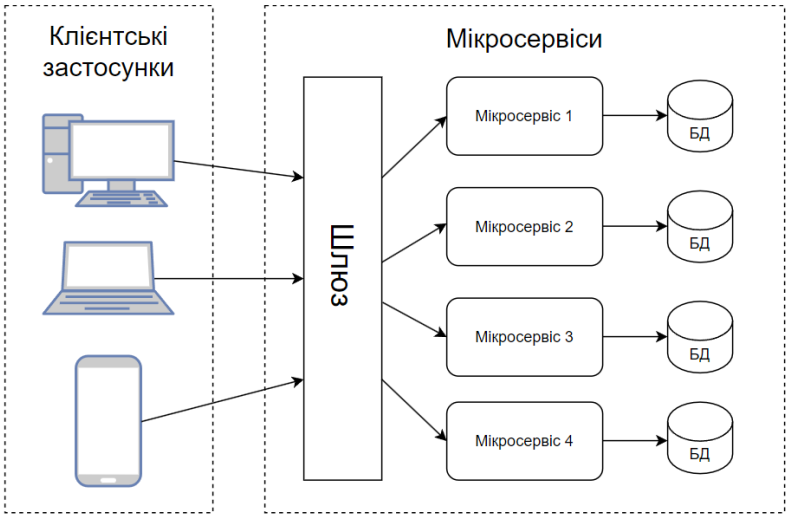

Рисунок 3. Схема використання шлюзу

Його сенс полягає в тому, що одна велика транзакція розбивається на декілька локальних транзакцій всередині кожного мікросервіса, що у подальшому виконуються поетапно. Важливим аспектом даного патерна є досягнення атомарності на кожному кроці виконання транзакції, що дозволить уникнути будь-яких змін у системі та подальшого виконання транзакції при появі помилки. Схема роботи патерна «Sega» наведено на рисунку 4.

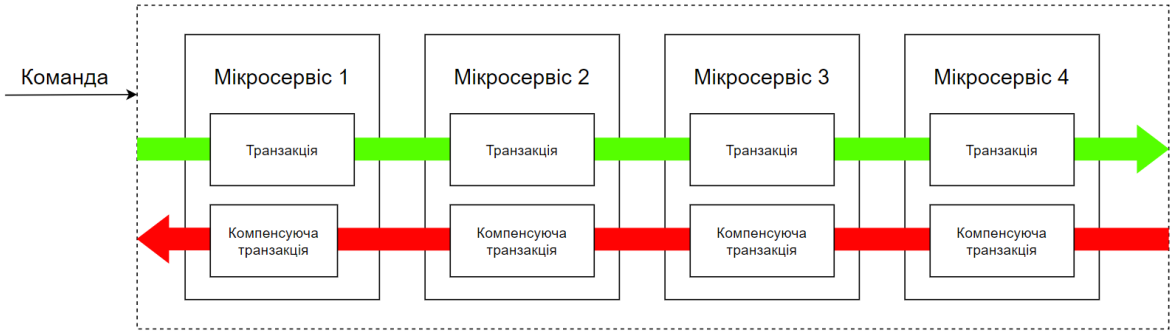

Рисунок 4. Схема використання шлюзу

Тестування мікросервісів

Оскільки мікросервісна архітектура є складною за рахунок її розподілу на незалежні мікросервіси, надзвичайно важливим аспектом розробки є тестування мікросервісів як окремо, так і у взаємодії з іншими. З цією метою розробники програмного забезпечення часто використовують процес безперервної інтеграції, що передбачає автоматизацію збірки та тестування системи кожного разу, коли до неї вносяться будь-які зміни.

Для швидкого тестування найнижчих рівнів та перевірки їхньої логіки застосовуються юніт-тести, що мають обмежену область застосування і прив'язуються до конкретних функцій та методів в межах окремого мікросервісу.

Наступним важливим видом тестів є інтеграційне тестування. У зв'язку з тим, що кожен мікросервіс виконує певну частину завдань важливим аспектом є забезпечення їхньої коректної взаємодії. Для цього створюються тестові сценарії, що передбачають взаємодію між різними мікросервісами, проводиться їх запуск та аналіз результатів.

Ще одним важливим та складним типом тестування є наскрізне тестування, з допомогою якого здійснюється перевірка всієї архітектури, що включає: інтерфейси застосунків, бізнес логіку та обмін даними. Головними недоліками даного тестування є те, що воно повільне, вся система повинна бути розгорнута повністю, а інтерпретація результатів невдалого тестування є складним і часто потребує проведення додаткових ручних тестів.

Висновки

На основі проведеного аналізу мікросервісної архітектури та її різних аспектів можна стверджувати, що практика останніх років демонструє позитивний розвиток архітектури, її методів реалізації, технологій та патернів проектування, хоча недоліки її використання досі мають суттєвий вплив на всю систему. У зв'язку з цим перед фінальним затвердженням архітектури варто додатково перевірити доцільність використання мікросервісної архітектури, переглянути бізнес-функції майбутньої системи та визначити можливості для потенційного розвитку.

На початкових етапах розробки мікросервісної архітектури надзвичайно важливим є визначення майбутніх мікросервісів, що будуть розроблені у майбутньому. Для цього варто використовувати підхід «domain-driven design» для розбиття домени, що описують певну діяльність чи ряд завдань, а використання даного підходу у зв'язці з граничними контекстами допоможе визначити відповідальність кожної моделі та спростити систему.

Сучасні технології, такі як, контейнеризація та оркестрація контейнерів демонструють, що мікросервіси можуть швидко розгортатись на різних системах та можливість відносно простого управління ними, а сучасні патерни проектування можуть вирішити ряд проблем, що наявні у мікросервісній архітектурі, але не позбутись їх повністю, покращити взаємодію між мікросервісами та підвищити надійність системи.

Тестування у мікросервісній архітектурі є одним із найважливіших етапів розробки програмного забезпечення, адже при наявності великою кількості мікросервісів неможливо визначити чи будуть вони коректно взаємодіяти між собою. З даних причин тестування є комплексним процесом, що включає в себе різні види тестування, наприклад, юніт-тести, інтеграційне та наскрізне шифрування.

Всі отримані дані будуть використані для кращого розуміння сучасних практик розробки мікросервісної архітектури, визначення переваг, недоліків та доцільності їхнього використання.

Література

1. Söylemez, Mehmet ; Tekinerdogan, Bedir ; Tarhan, Ayça Kolukısa. / Challenges and Solution Directions of Microservice Architectures : A Systematic Literature Review. In: Applied Sciences (Switzerland). 2022

2. Спасітєлєва, Світлана Олексіївна та Чичкань, І. та Шевченко, Світлана Миколаївна та Жданова, Юлія Дмитрівна (2023) Розробка безпечних контейнерних застосунків з мікросервісною архітектурою Кібербезпека: освіта, наука, техніка, 1 (21). с. 193-210.

3. Pahl, C., & Jamshidi, P. (2016). Microservices: a systematic mapping study. In Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) (pp. 137-146)

4. J. Heikkinen, Serverless and microservice architecture in modern software development, JAMK University of Applied Sciences, 2023.

References

1. Söylemez, Mehmet ; Tekinerdogan, Bedir ; Tarhan, Ayça Kolukısa. / Challenges and Solution Directions of Microservice Architectures : A Systematic Literature Review. In: Applied Sciences (Switzerland). 2022

2. Spasitielieva, Svitlana Oleksiivna ta Chychkan, I. ta Shevchenko, Svitlana Mykolaivna ta Zhdanova, Yuliia Dmytrivna (2023) Rozrobka bezpechnykh konteinernykh zastosunkiv z mikroservisnoiu arkhitekturoiu Kiberbezpeka: osvita, nauka, tekhnika, 1 (21). s. 193-210.

3. Pahl, C., & Jamshidi, P. (2016). Microservices: a systematic mapping study. In Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) (pp. 137-146)

4. J. Heikkinen, Serverless and microservice architecture in modern software development, JAMK University of Applied Sciences, 2023.