*ISSN 2522-9818 (print)*
*ISSN 2524-2296 (online)* *Innovative technologies and scientific solutions for industries. 2023. No. 3 (25)*

УДК 004.41:004.75 DOI: https://doi.org/10.30837/ITSSI.2023.25.064

С. Переяславська, О. Смагіна

# ПРОЄКТУВАННЯ РІВНЯ МАРШРУТИЗАЦІЇ В МІКРОСЕРВІСНИХ АРХІТЕКТУРАХ НА ПЛАТФОРМІ *SPRING*

**Предметом дослідження** є маршрутизація запитів у мікросервісній архітектурі. **Мета статті** – розроблення цілісної концепції проєктування рівня маршрутизації запитів у мікросервісній архітектурі на прикладі стеку технологій *Spring*. **Завдання**: проаналізувати сучасні підходи щодо структури мікросервісної архітектури; визначити сутність маршрутизації та встановити процеси, що забезпечують маршрутизацію запитів; визначити стек технологій *Spring*, які реалізують маршрутизацію; спроєктувати рівень маршрутизації застосунку на платформі *Spring*. Упроваджуються такі **методи**: аналіз і синтез для вивчення технологій взаємодії між службами; абстрагування та узагальнення для визначення структури мікросервісної архітектури, рівня маршрутизації, узагальнення технологій, що забезпечують взаємодію між сервісами; моделювання з метою побудови моделі мікросервісної архітектури з виокремленням рівня маршрутизації та зв'язків з іншими структурами моделі. Здобуто такі **результати**: досліджено структуру мікросервісної архітектури, зокрема рівень маршрутизації; визначено роль шаблонів проєктування, що забезпечують маршрутизацію: *Service discovery, API Gateway, Load Balancer* тощо; проаналізовано види міжпроцесної взаємодії (синхронна, асинхронна, гібридна) та визначено переваги й доцільність застосування; розглянуто моделі відмовостійкості системи; визначено стек технологій на платформі *Spring* для реалізації рівня маршрутизації; розроблено модель проєкту багаторівневої мікросервісної архітектури із застосуванням стеку технологій *Spring*, що реалізує найбільш ефективні рішення в контексті маршрутизації запитів. **Висновки:** мікросервісну архітектуру доцільно розглядати як багаторівневу структуру, що будується на функціональних рівнях і зв'язках між ними; рівнем маршрутизації мікросервісів потрібно вважати всі процеси, пов'язані з налагодженням міжпроцесної взаємодії, виявленням сервісів, балансуванням навантаження та забезпеченням відмовостійкості, створенням єдиної точки входу; *Spring* є популярною платформою розроблення мікросервісної архітектури, що надає необхідні інструменти для реалізації маршрутизації запитів; розроблена модель проєкту є прикладом ефективних рішень щодо проєктування багаторівневої архітектури із застосуванням стеку технологій *Spring* у контексті маршрутизації запитів.

**Ключові слова**: мікросервісна архітектура; маршрутизація; *Spring; Service discovery; API Gateway; Load Balancer*.

## Вступ

Еволюція підходів до проєктування програмного забезпечення, зростання складності розподілених застосунків, необхідність постійного управління змінами спричинили появу різних архітектурних рішень, спрямованих на вдосконалення уніфікованої моделі розроблення програмного забезпечення. У сучасних розподілених системах спостерігається відхід від традиційних способів побудови проєкту, де вся система зосереджена в єдиному й неподільному блоці, та зміщення акцентів у напрямі застосування мікросервісів.

Архітектура мікросервісів – це своєрідний стиль сервіс-орієнтованої архітектури (*SOA*). Вона побудована з мікросервісів, що є відносно невеликими й слабко пов'язаними та працюють у своїх процесах і взаємодіють через протоколи зв'язку, зокрема *HTTP/HTTPS, Web Socket, AMQP* тощо [1].

Варто заначити, що мікросервіси в деякому сенсі є наступним кроком в еволюції сервіс-орієнтованої архітектури, оскільки підтримують багато концепцій, що й *SOA*. Обидва підходи мають загальні риси, притаманні розподіленим архітектурам, компоненти в цих архітектурах більш автономні, завдяки чому спрощується обслуговування та контроль над змінами. Утім існують певні відмінності, що допомагають виокремити мікросервіси як самостійний напрям у розвитку архітектури розподілених систем. Незважаючи на те, що мікросервіси й *SOA* покладаються на "сервіс" як основний компонент архітектури, вони розрізняються за характеристиками сервісів (служб). Концептуальні відмінності полягають у обсязі відповідальності, покладеної на окрему службу. У *SOA* служба може обробляти широкий спектр функцій і доменів даних, тоді як мікросервіс управляє одним доменом даних і одним набором відповідних функцій або однією функцією в цьому домені. Кожен мікросервіс має розгортатися незалежно один від одного. Нарешті, кожен мікросервіс має володіти своєю пов'язаною моделлю даних і може ґрунтуватися на різних технологіях зберігання даних (*SQL, NoSQL*) і різних мовах програмування.

ISSN 2522-9818 (print)
*Сучасний стан наукових досліджень та технологій в промисловості. 2023. № 3 (25)* ISSN 2524-2296 (online)

Ці характеристики забезпечують чимало переваг мікросервісної архітектури, але водночас вони створюють проблеми для розвитку та масштабування програмної системи: зростання кількості мікросервісів призводить до ускладнення архітектури і, як наслідок, утруднення процесів, пов'язаних зі взаємодією між мікросервісами.

Питання про те, як мікросервіси взаємодіють один з одним, є найважливішим і фундаментальним рішенням, що необхідно прийняти під час упровадження системи на основі архітектури мікросервісів. У монолітній програмі, що працює в єдиному процесі, компоненти викликають один одного за допомогою методів рівня мови або викликів функцій. Тому проблем із взаємодією не виникає. Інша річ, коли йдеться про мікросервіси, що працюють в окремих процесах та є автономними й слабко пов'язаними. У цьому разі мається на увазі міжпроцесна взаємодія, продуктивність якої відрізнятиметься від внутрішньопроцесної, що потребує конкретних підходів, пов'язаних із вибором шаблонів проєктування мікросервісів, моделі взаємодії, балансування навантаження тощо.

Під час взаємодії мікросервісів у мережі може виникати досить значна кількість проблем, пов'язаних із доступністю мікросервісу, затримкою передачі, втратою, дублюванням пакетів, навантаженням трафіку й пам'яті тощо. Це викликає необхідність оптимізації та пошуку підходів, що допоможуть вирішити зазначені проблеми. Крім того, у сучасних розподілених мікросервісних системах велика частка мережної взаємодії відбувається за допомогою API-інтерфейсів прикладного програмування. API-інтерфейси постійно розширюються, клієнтів стає більше, зростає кількість звернень, що призводить до значного зниження ефективності роботи системи. Тому потрібна певна інфраструктура для забезпечення маршрутизації запитів та прискорення API-трафіку.

Зазначені проблеми стають очевидними, коли прагнемо міркувати про систему загалом, щоб зрозуміти залежності, аналіз впливу тощо. Тому рішення про те, як організувати зв'язок між службами, є важливим завданням у розробленні мікросервісів. У цьому контексті постає питання проєктування рівня маршрутизації в мікросервісній архітектурі. Від правильного вибору шаблону зв'язку, компонентів, що забезпечують взаємодію між службами, залежить продуктивність системи.

## Аналіз останніх досліджень і публікацій

Аналіз літературних джерел довів, що проблемам, пов'язаним із мікросервісною архітектурою, приділялася увага в багатьох наукових працях. Так, *Kazanavicius* та *D. Mazeika* [2] порівнювали технології взаємодії, що базуються на *HTTP Rest, GraphQL, gRPC* та брокерах повідомлень *RabbitMQ* і *Kafka*. Оцінка продуктивності в умовах порівняння затримки в процесі оброблення повідомлень і пропускної здатності довела, що найнижчі результати затримки для повідомлення до одного мільйона символів були отримані технологією *RabbitMQ*. Виклики *RabbitMQ* були вдвічі швидшими за інші технології. З іншого боку, *RabbitMQ* показав найвищі результати затримки для повідомлень, які містили 10 млн символів. Це було втричі-вчетверо повільніше, ніж в інших ситуаціях. Найкращі результати затримки для повідомлень завдовжки до 10 млн символів здобули технології *GraphQL* і *HTTP Rest*. Брокер *Kafka* становив 40%, а *gRPC* – 16%, що повільніше, ніж технології *GraphQL* і *HTTP Rest.*

Під час оцінювання метрик установлено, що найменший розмір запиту/відповіді отриманий технологією *HTTP Rest*. Запит/відповідь *GraphQL* були приблизно вдвічі-тричі більші, ніж інші. Якщо розмір повідомлення є важливим критерієм для вибору технології зв'язку, то *HTTP Rest* є рекомендованим рішенням. З іншого боку, *GraphQL* підтримує дистанційне надсилання запитів, тому потенційно за один запит/відповідь *GraphQL* може передати стільки ж інформації, скільки здатні передати кілька запитів/відповідей за допомогою інших технологій.

Міжпроцесну взаємодію в мікросервісній архітектурі розглядали І. Бугаєва, М. Розум [3]. У статті досліджуються синхронний та асинхронний підходи для реалізації взаємодії, проведено порівняльний аналіз на основі наявних досліджень для виявлення переваг і недоліків кожного з цих підходів. Автори дійшли висновку, що асинхронна взаємодія мікросервісів, реалізована на основі протоколу *AMQP*, забезпечує кращу продуктивність і вищу доступність мікросервісів. Синхронна форма взаємодії мікросервісів, яка реалізується за допомогою *REST* або *gRPC*, може забезпечити вищу пропускну здатність, ніж асинхронний метод, коли навантаження на систему відносно невисоке. За умови збільшення кількості користувачів доцільно

ISSN 2522-9818 (print)
ISSN 2524-2296 (online) Innovative technologies and scientific solutions for industries. 2023. No. 3 (25)

застосовувати асинхронний метод зв'язку з використанням черг повідомлень. Крім того, такий спосіб обміну повідомленнями між мікросервісами є більш надійним. Черги повідомлень є також кращим вибором за необхідності синхронізації інформації між сервісами.

Роботи, присвячені проблемам балансування навантаження в мікросервісній системі [4, 5], дали змогу чітко зрозуміти роль і місце цієї технології під час розроблення масштабованого застосування. Так, у дослідженні *M. Autili* та інших авторів [5] розглядаються базові концепції балансування навантаження. Зазначається, що ця технологія є найкращим підходом для підтримки масштабованості, управління швидкістю надходження або кількістю одночасних запитів. Балансувальник намагається розподілити робоче навантаження запитів між кількома екземплярами мікросервісу з метою оптимізації використання ресурсів, максимізації пропускної здатності, мінімізації часу відповіді та уникнення вузьких місць (тобто перевантаження будь-якого окремого екземпляра). У сфері мікросервісів зазвичай розрізняють два типи балансувальників навантаження, а саме: клієнтські та серверні. У роботі [5] автори пропонують новий гібридний підхід до балансування навантаження мікросервісів, що поєднує в собі переваги балансування на боці клієнта й на боці сервера.

*H. Wang* та інші дослідники у праці [4] розглядають різні алгоритми балансування навантаження на основі методу опитування, хеш-алгоритму, штучного інтелекту тощо. У згаданій роботі описуються принципи, переваги й недоліки основних технологій балансування навантаження, запропоновано стратегії механізму балансування навантаження *Kubernetes* і *Spring Cloud Framework*, що мають абсолютну частку ринку.

Продуктивність шаблонів проєктування мікросервісів за допомогою багатопотокових структур, що генерують заздалегідь визначену кількість потоків, у своєму дослідженні проаналізували *A. Akbulut* та *H. G. Perros* [6]. Автори здобули результати продуктивності, пов'язані з часом відповіді на запит, ефективним використанням обладнання, витратами на хостинг і рівнем втрати пакетів для трьох шаблонів проєктування мікросервісів, а саме *API Gateway, Chain of responsibility, Asynchronous Messaging*.

Під час дослідження шаблону *API Gateway* експериментально встановлено, що потрібно використовувати понад один екземпляр мікросервісу, коли кількість потоків перевищує 140. Горизонтальне масштабування дає змогу зменшити або залишити постійним середній час обслуговування зі збільшенням кількості потоків. *API Gateway* забезпечує гнучкість, коли потрібно керувати запитами з кількох каналів. Це полегшує логіку виконання синхронних комунікацій між однаково збалансованими службами й забезпечує артефакти масштабування. У роботі [6] наголошується, що екосистема мікросервісів без API-шлюзу вважається поганою практикою або антипатерном.

Досліджуючи шаблон *Chain of responsibility*, *A. Akbulut* та *H. G. Perros* встановили, що одна й та сама програма може потребувати різної оперативної пам'яті та ресурсів для різних реалізацій шаблонів проєктування. Отже, шаблон дизайну є важливим рішенням, що безпосередньо впливає на вартість хостингу. Ідеальним шаблоном проєктування мікросервісу для функцій, які послідовно обробляють спільні дані, є шаблон проєктування *Chain of responsibility*. Під час дослідження автори помітили, що ця архітектура дає змогу масштабувати серверні послуги з вигодою від використання апаратного забезпечення приблизно на 30 % порівняно з еквівалентною структурою мікросервісу.

Аналізуючи асинхронні взаємодії, *A. Akbulut* та *H. G. Perros* наголошують, що ємність черги, реалізованої за допомогою *RabbitMQ*, обмежена фіксованим обсягом пам'яті, що спричиняє переповнення черги і, як наслідок, втрату повідомлень. Досліджуючи результати реалізації асинхронного обміну повідомленнями з *RabbitMQ* для двох різних конфігурацій, автори праці [6] помітили, що втрата пакетів мінімізована з меншими конфігураціями (менше ніж 122,288 Гб *RAM* і менше ніж шість *CPU* 4 ГГц).

*A. Akbulut* та *H. G. Perros* зробили висновок, що не існує єдиного шаблону мікросервісів, який би був кращим за інші. Навпаки, для різних сценаріїв ефективними будуть різні шаблони проєктування.

У роботі [7] дослідники порівняли та критично оцінили найбільш перспективні фреймворки *JVM*, що підтримують розвиток мікросервісів (*Spring Boot, Quarkus, Micronaut*). Згідно з дослідженнями продуктивними виявилися *Quarkus* і *Spring Boot*. *Micronaut* також зміг досягти конкурентних результатів, але проблеми, виявлені під час стрес-тестів, спричинили відставання від конкурентів. Час компіляції файлів *JAR* помітно довший для *Micronaut* і *Quarkus*,

*Сучасний стан наукових досліджень та технологій в промисловості. 2023. № 3 (25)*
*ISSN 2522-9818 (print)*
*ISSN 2524-2296 (online)*

оскільки вони більшість операцій, що в разі *Spring Boot* опрацьовуються під час виконання, здійснюють у процесі компіляції.

Низка дослідників [8] розглянула різні підходи до впровадження мікросервісів на базі чотирьох фреймворків мікросервісів (*Spring Boot 2.2.4 (Java 1.8), Go Micro 1.18.0, (Go 1.13), Moleculer 0.13.0 (Node.js 13.1.4), Lagom 1.6.0 (Scala 2.12.8)*). Модель порівняння для фреймворків будувалася за трьома критеріями: функції, шаблони проєктування та продуктивність. Три з чотирьох фреймворків, досліджених у цій роботі (*Lagom, Moleculer* і *Go Micro*), заохочують розробників структурувати протокол зв'язку перед розробленням фактичних мікросервісів. Навпаки, *Spring Boot* не змушує розробників дотримуватися певного стилю коду для визначення API програми. Однією з можливих причин є те, що *Spring Boot* – це фреймворк загального призначення, що не розроблений спеціально для мікросервісів. Цей факт підкреслює важливість поєднання шаблону зв'язку в розподілених системах загалом і в мікросервісах зокрема (*Spring Boot / Spring Cloud*). Крім того, автори наголошують, що прийняття структури мікросервісу з використанням стандартів або популярних інструментів може бути набагато вигіднішим для організації в довгостроковій перспективі. У зв'язку з цим *Spring Boot / Spring Cloud* є одним із найпопулярніших програмних фреймворків, що надає розробникам явну перевагу для створення мікросервісів, оскільки цей фреймворк має значний рівень підтримки та доступної інтеграції.

## Визначення не вирішених раніше частин загальної проблеми. Мета роботи, завдання

Аналіз літературних джерел довів, що в роботах, присвячених мікросервісній архітектурі, автори здебільшого досліджували окремі питання, пов'язані з комунікацією служб. Водночас існує потреба в комплексному дослідженні технологій маршрутизації запитів у мікросервісній архітектурі. Такий підхід дасть змогу розглядати маршрутизацію як багатокомпонентний та одночасно цілісний процес, краще розуміти питання ефективної взаємодії між всіма компонентами мікросервісної архітектури, що безпосередньо впливатиме на вибір технологій та інструментів реалізації.

**Мета статті** – розроблення цілісної концепції проєктування рівня маршрутизації запитів у мікросервісній архітектурі на прикладі стеку технологій *Spring*. Для досягнення поставленої мети потрібно виконати такі **завдання**: проаналізувати сучасні підходи щодо структури мікросервісної архітектури; визначити сутність маршрутизації та встановити основні процеси, що забезпечують маршрутизацію запитів; визначити стек технологій *Spring*, що реалізують маршрутизацію; на підставі проведеного дослідження спроєктувати рівень маршрутизації мікросервісного застосунку на платформі *Spring*.

## Матеріали й методи

Методологічною основою роботи є такі методи наукового дослідження: аналіз і синтез використовуються для вивчення технологій взаємодії між службами; абстрагування та узагальнення застосовуються для визначення загальної структури мікросервісної архітектури, рівня маршрутизації, узагальнення технологій, що забезпечують взаємодію між сервісами; моделювання впроваджується з метою побудови моделі мікросервісної архітектури з виокремленням рівня маршрутизації та зв'язків з іншими структурами моделі.

## Результати досліджень та їх обговорення

Сучасна архітектура мікросервісів має складну структуру, що підтверджується дослідженнями [4, 9, 10]. *Y. Zhang* та низка вчених визначають багаторівневі мікросервісні застосунки як складні системи масового обслуговування зі зв'язками між службами [9]. У роботі [4] мікросервісну архітектуру розглядають як багатокомпонентну структуру, що містить реєстрацію та виявлення служб, зв'язки між ними, моніторинг, балансування навантаження, масштабування, відмовостійкий і безпековий механізми, конфігурацію та інші технології керування мікросервісами. На думку авторів роботи [10], мікросервіси є складними розподіленими системами. Це дає підставу розглядати мікросервісну архітектуру як багаторівневу структуру, що будується на функціональних рівнях та зв'язках між ними. Така архітектура зазвичай містить рівні користувача (клієнта), конфігурації, маршрутизації, мікросервісів, даних тощо. Перелік та склад рівнів залежить від завдань, вимог до мікросервісної архітектури й под. Саме такий підхід, на нашу думку, забезпечить

ISSN 2522-9818 (print)
ISSN 2524-2296 (online) Innovative technologies and scientific solutions for industries. 2023. No. 3 (25)

спрощеність проєктування мікросервісних застосунків завдяки прозорості всіх зв'язків між рівнями та їх компонентами.

Рівень маршрутизації пов'язаний насамперед із сервісами й запитами до них. На сайті *Amazon AWS (https://aws.amazon.com/)* подається визначення поняття маршрутизації як процесу вибору шляху в будь-якій мережі. Отже, інфраструктура маршрутизації в мікросервісній архітектурі відповідає за спрямування запиту до служби. На це впливає низка факторів: оркестрування мікросервісів, що також передбачає зв'язок і координацію служб [2], обрана міжпроцесна взаємодія тощо. У роботі [2] визначаються шаблони архітектури мікросервісів, що безпосередньо впливають на маршрутизацію запитів, а саме: *API Gateway* (API-шлюз), *Service discovery* (виявлення служб) та *Hybrid* (гібридний шаблон, що поєднує в собі реєстр служб і API-шлюз). Тому до рівня маршрутизації мікросервісів, на наш погляд, належать усі процеси, пов'язані з налагодженням міжпроцесної взаємодії, виявленням сервісів, балансуванням навантаження та забезпеченням відмовостійкості, створенням єдиної точки входу (шлюзу).

У шаблоні мікросервісної архітектури існує кілька сервісів, що мають взаємодіяти один з одним. Є чимало екземплярів кожної служби, нові служби можуть додаватися, а вже наявні вилучатися під час виконання. Необхідно, щоб служби знали, які з них ще доступні та де розташовані кінцеві точки для них. Тому для вирішення цієї проблеми використовуються механізми виявлення служб.

Виявлення служб (*Service discovery*) – це процес визначення місця розташування наявних сервісів, що відповідають запиту на основі опису їх функціональної та нефункціональної семантики [11]. Підходи до виявлення служб відрізняються підтримкою мов опису сервісів, організацією пошуку та використовуваними засобами вибору служб. Ключовим компонентом виявлення служби є реєстр служб (сервісів) – база даних, що зберігає розташування екземплярів служб. Він надає інтерфейс для реєстрації, перевірки працездатності тощо.

Існує два основних шаблони виявлення служб, – на боці клієнта та на боці сервера [12]:

– методи виявлення служб на боці сервера полягають у делегуванні відповідальності за виявлення мережного розташування служб сервера. Цей патерн дає змогу клієнтським програмам надсилати запит до сервісу через балансувальник навантаження, що звертається до реєстру сервісів, перш ніж перенаправити клієнтський запит;

– методи виявлення служб на боці клієнта містять виклик реєстру служб для виявлення мережного розташування інших служб. Послуги реєструються в реєстрі. Коли клієнту потрібно зв'язатися зі службою, виявлення служби здійснює запит до реєстру, отримує розташування потрібної клієнту служби й направляє запит за призначенням. Отже, клієнту не потрібно заздалегідь знати мережне розташування служби, щоб отримати доступ до неї.

Окрім виявлення служб, у мікросервісній архітектурі важливо надавати API-рівень, що забезпечує інтеграцію з широким спектром послуг, якими користуються клієнти. *V. Silva* та інші дослідники в роботі [12] звертають увагу на проблему, що може виникати під час міжпроцесної взаємодії, особливо коли йдеться про великі й складні програми на основі мікрослужб. Зазвичай кожна мікрослужба має свій API. Найпростішим рішенням є безпосереднє надання API для інших, як це роблять моноліти. Однак це може бути неідеальним рішенням, коли доводиться керувати десятками API, що безпосередньо доступні користувачам та іншим мікросервісам. Це ускладнює роботу архітекторів і розробників серверної частини, яким потрібно буде підтримувати узгодженість усіх API, над якими працюють різні команди, а також для розробників зовнішньої частини, що покладаються на ці API. Зазначений підхід може спричинити зниження продуктивності програми під час виклику кількох API, проблеми підтримки протоколів зв'язку тощо. Дослідження [6] свідчить, що найбільш ідеальною архітектурою для такого сценарію є шаблон проєктування *API Gateway* (API-шлюз).

API-шлюз – це єдина точка входу, що забезпечує доступ до різних мікросервісів. Це спрощує роботу клієнта завдяки переміщенню логіки виклику кількох мікросервісів до шлюзу API. *API Gateway* діє як проксі-рівень, що відокремлює серверну інфраструктуру. Наявних клієнтів не цікавлять зміни у внутрішній інфраструктурі, оскільки вони взаємодіють лише з рівнем шлюзу API [12]. Шлюз абстрагує мікросервіси від їх споживачів та надає кінцеву точку або URL-адресу для клієнтських програм, а потім внутрішньо зіставляє запити з групою служб. Тобто однією з переваг *API Gateway* є інкапсуляція внутрішньої структури програми, що, на нашу думку, створює додаткові засоби безпеки у використанні мікросервісів.

*Сучасний стан наукових досліджень та технологій в промисловості. 2023. № 3 (25)*
*ISSN 2522-9818 (print)*
*ISSN 2524-2296 (online)*

*J. Kazanavicius* та *D. Mazeika* [2] звертають увагу на можливість використання гібридного шаблону, що поєднує функції реєстру служб і API-шлюзу, однак роль цих шаблонів виконує шина повідомлень. Клієнти взаємодіють лише з цією шиною, що працює як реєстр і як шлюз. Сервіси взаємодіють один з одним через шину повідомлень, і прямий зв'язок між мікросервісами не використовується. Перевагами шаблону є легкість міграції, а недоліками – високий зв'язок між службами та шиною повідомлень.

Під час масштабування мікросервісів можуть виникати проблеми, пов'язані з рівномірним розподілом запитів між службами й перевантаженням системи. Балансування навантаження є найбільш відомим підходом для підтримки масштабованості в архітектурі мікросервісів [5]. Завдяки цьому можна ефективно зменшити затримку запиту в комунікації мікросервісів, підвищити доступність системи та ефективно покращити сервісні можливості та використання ресурсів [4].

Ця технологія застосовується для рівномірного пересилання запитів від користувача або служб до отримувачів за допомогою певної стратегії. Алгоритми балансування навантаження можна поділити на дві категорії: алгоритм статичного балансування навантаження та алгоритм динамічного балансування навантаження. Статичне балансування застосовує методи опитування, хешування адреси джерела, узгодженого хешування, зваженого опитування, випадковий метод тощо. Алгоритми динамічного балансування навантаження ґрунтуються на методах мінімального числа з'єднань, найшвидшої відповіді тощо [4]

У мікросервісній архітектурі зазвичай розрізняють два типи балансувальників навантаження, а саме клієнтські та серверні. Балансування навантаження на боці сервера – це централізований підхід до розподілу запитів між доступними екземплярами мікросервісу. Клієнти взаємодіють з балансувальником навантаження, що надсилає запити відповідним екземплярам служби. Балансування навантаження на боці клієнта – повністю розподілений підхід, згідно з яким кожному екземпляру клієнта (і кожному екземпляру мікросервісу) призначається локальний балансувальник навантаження [5].

Один із ключових компонентів рівня маршрутизації в мікросервісній архітектурі є міжпроцесна взаємодія (*interprocess communication – IPC*), що відіграє набагато важливішу роль, ніж у монолітних застосунках. Міжпроцесна взаємодія між службами має вирішальне значення в архітектурі мікросервісів, оскільки вона дає змогу окремим мікросервісам бездоганно працювати разом як єдина система.

Зазвичай кожен екземпляр служби – це процес. Отже, служби мають взаємодіяти за протоколом внутрішньопроцесної взаємодії, наприклад, HTTP, SOAP, AMQP або протоколом, таким як TCP, залежно від характеру кожної служби. Важливо зазначити, що найбільш популярні комунікаційні технології, що використовуються для мікросервісів, ґрунтуються на протоколі HTTP та шаблонах обміну повідомленнями [2]. Ці технології взаємодії між мікросервісами реалізуються, відповідно, в синхронному та асинхронному режимах.

У синхронному зв'язку застосовується взаємодія запит/відповідь: один мікросервіс надсилає запит іншому та чекає, поки цей сервіс обробить результат і надішле відповідь. У цьому режимі запитувач блокує свою роботу на час очікування відповіді. Найпоширенішими структурами для реалізації синхронного стилю зв'язку в мікросервісах є інтерфейс прикладного програмування репрезентативної передачі стану (*REST API*) на основі протоколу HTTP та фреймворк віддаленого виклику процедур *gRPC*.

Більшості синхронних комунікацій властивий стиль "один до одного". Кожен запит має оброблятися лише одним отримувачем або службою. Але за цим стилем можна використовувати численні екземпляри служби з метою масштабування, однак доведеться застосовувати механізм або методи балансування навантаження на боці клієнта [13]. У цьому разі кожна служба має інформацію про адреси розташування всіх екземплярів. Цю інформацію можна взяти із сервера виявлення служб або надати вручну у властивостях конфігурації. Також кожна служба має вбудований клієнт маршрутизації, що може обрати один екземпляр цільової служби й надіслати туди запит.

Асинхронний режим зв'язку може бути реалізований у мікросервісах, коли служби взаємодіють одна з одною за допомогою шаблону обміну повідомленнями на базі асинхронних протоколів, таких як AMQP. У цій формі IPC-мікросервіси мають посередника між сервісами для координації запитів і відповідей (наприклад, брокер *RabbitMQ* або *Apache Kafka*).

У деяких ситуаціях *RESTful* також можна використовувати як асинхронний або так званий "відкладений синхронний" зв'язок. Застосовуючи

ISSN 2522-9818 (print)
ISSN 2524-2296 (online) Innovative technologies and scientific solutions for industries. 2023. No. 3 (25)

його асинхронно, служба негайно відповідає на запит кодом відповіді (як прийнятий запит) та посиланням для отримання фактичних результатів після завершення операції [1]. Недоліком такого підходу є те, що створюються додаткові HTTP-запити. Це також спричиняє складнощі для клієнта, оскільки він тепер має перевіряти процес виконання запиту [14].

Одна з фундаментальних відмінностей асинхронного зв'язку порівняно з синхронним полягає в тому, що клієнт (служба) більше не здійснює прямий виклик до іншої служби й не очікує негайної відповіді. Замість цього клієнт публікує запит до брокера повідомлень. Одна чи декілька служб візьмуть запит від брокера й оброблять його, потім повернуть результат брокеру. Оскільки зв'язок є асинхронним, у цьому режимі клієнт не блокує роботу під час очікування відповіді [14].

Асинхронний режим обміну повідомленнями може бути реалізований за моделями "один до одного" (черга) та "один до багатьох" (тема). Асинхронний зв'язок на основі повідомлень з одним отримувачем означає, що існує зв'язок "точка-точка", що доставляє повідомлення точно одному зі споживачів, який має доступ до каналу, і що повідомлення обробляється лише один раз. Як більш гнучкий підхід можна використати механізм публікації/підписки, щоб повідомлення від відправника були доступні для мікросервісів-підписників або зовнішніх програм.

*B. Reselman* у роботі [15] виокремлює ще один тип взаємодії між мікросервісами – гібридний, який поєднує в собі синхронну й асинхронну взаємодію. Наприклад, гібридна служба підтримує як HTTP, так і протоколи обміну повідомленнями. Сучасні застосунки з архітектурою мікросервісів часто використовують комбінацію синхронної та асинхронної взаємодії. Наприклад, взаємодія з одним отримувачем може відбуватися за синхронним протоколом (HTTP тощо) під час виклику звичайної служби із зовнішнім API, для асинхронної взаємодії декількох служб використовуються протоколи повідомлень (AMQP).

У цьому контексті важливим є розуміння переваг і недоліків різних режимів міжпроцесної взаємодії. На основі аналізу праць [3, 10] можна підсумувати, що різні фактори, зокрема завантаження запитів, IT-середовище й мережні технології, визначають продуктивність зв'язку між мікросервісами. Не можна однозначно визначити, яка з комунікаційних технологій краща. Це залежить від конкретного застосування. Але можна зауважити, що асинхронний зв'язок є надійнішим і стабільнішим механізмом зв'язку, ніж HTTP (*Rest*), і забезпечує більшу автономію мікросервісу [2]. *S. Newman* у дослідженні [11] аналізує переваги й недоліки кожної форми зв'язку та зазначає, що в простих мікросервісних архітектурах серйозних проблем із використанням синхронних блокувальних викликів не виникає. Але цей тип зв'язку стає неактуальним, коли ускладнюється маршрутизація та з'являється більше викликів. Асинхронний зв'язок є доцільним у разі, коли в архітектурі передбачаються тривалі процеси з довгими ланцюжками викликів, що нелегко реструктурувати. Звісно, водночас треба вирішити, який тип асинхронного зв'язку застосовуватиметься: черга повідомлень, подійно-орієнтована взаємодія тощо.

Сказане вище доводить, що вибір механізму IPC – важливе архітектурне рішення, що може вплинути на рівень доступності служб. IPC має забезпечувати ефективний зв'язок між мікросервісами, водночас цей зв'язок повинен бути слабким. Вона також має забезпечувати покращену масштабованість, легкість обслуговування та автономність між службами [3]. Але спосіб зв'язку має бути таким, щоб не перешкоджати меті мікросервісів [16].

Отже, під час проєктування мікросервісної архітектури на рівні маршрутизації треба обґрунтовано підходити до визначення IPC, виявлення сервісів, балансування навантаження, створення єдиної точки входу. Але, на наш погляд, є певні фактори, що можуть пливати на якість цих процесів. Під час взаємодії мікросервісів виникають ситуації, коли з будь-яких причин один із сервісів, що використовується, перестає відповідати. У разі, коли збій є тимчасовим, наприклад, у зв'язку з повільним мережним з'єднанням тощо, вирішити проблему допомагають повторні виклики. Але коли йдеться про більш суттєві проблеми, що виникають у зв'язку з повною відмовою сервісу, то це рішення не спрацює, адже повторні виклики лише витрачатимуть ресурси. Тому важливо передбачити здатність системи справлятися зі збоями без повного руйнування та гарантувати якісну взаємодію між мікросервісами, тобто забезпечити відмовостійкість.

У роботі [17] розглядається модель відмовостійкої системи за такими напрямами:

– дії, пов'язані з недоступністю служби, що реалізуються за трьома схемами – швидкий збій, коли служба негайно повертає повідомлення

*Сучасний стан наукових досліджень та технологій в промисловості. 2023. № 3 (25)*
*ISSN 2522-9818 (print)*
*ISSN 2524-2296 (online)*

про помилку; тайм-аут можна використовувати, щоб встановити обмеження на службу, доки вона не відповість; кешування застосовується, коли служба, що викликається, недоступна, але інші служби все ще можуть читати/записувати інформацію в кеші на боці клієнта чи проксі;

– захист служб від перевантаження, що реалізується за двома моделями, а саме в автоматичному вимикачі *Circuit Breaker*, що запобігає доступу до компонента збою, швидко обробляє помилки, налаштовує резервну дію, уникає перевантаження. Друга модель використовує механізм "рукостискання", що здійснюється під час кожної взаємодії між запитувачем і службою, яка отримує запит. Сервіс, що отримує запит, може відмовити, якщо є перевантаження. Недоліком "рукостискання" є те, що потрібні додаткові запити й відповіді. Крім зазначених моделей, проблему може вирішувати балансування навантаження. *Load Balancer* допомагає знизити навантаження на служби й водночас зменшити вірогідність збоїв;

– локалізація несправності за допомогою моделі перегородки (*bulkheads*). Перегородки обмежують або ізолюють служби, які вийшли з ладу, щоб зменшити вплив несправностей на продуктивність системи.

*S. Newman* у праці [11] зазначає, що, окрім цих моделей, є декілька способів збільшити відмовостійкість системи: розподіл ризиків завдяки розміщенню сервісів на різних хостах, збільшення ізольованості сервісів тощо.

У роботах дослідників [11, 17], які вивчали механізми забезпечення відмовостійкості, зазначається, що застосування ключових шаблонів вимагає зваженого підходу з огляду на необхідний рівень відмовостійкості, архітектури мікросервісів тощо.

На рис. 1. зображена узагальнена концептуальна схема проєктування рівня маршрутизації в мікросервісній архітектурі. Схема містить основні компоненти, що забезпечують маршрутизацію: міжпроцесна взаємодія, виявлення сервісів, шлюз, балансування навантаження й підтримка відмовостійкості. Під час проєктування необхідно визначити основні моделі, шаблони або алгоритми відповідно до цих компонентів. Прийняті рішення будуть істотно впливати на архітектуру застосунку та вибір програмних засобів для її реалізації.

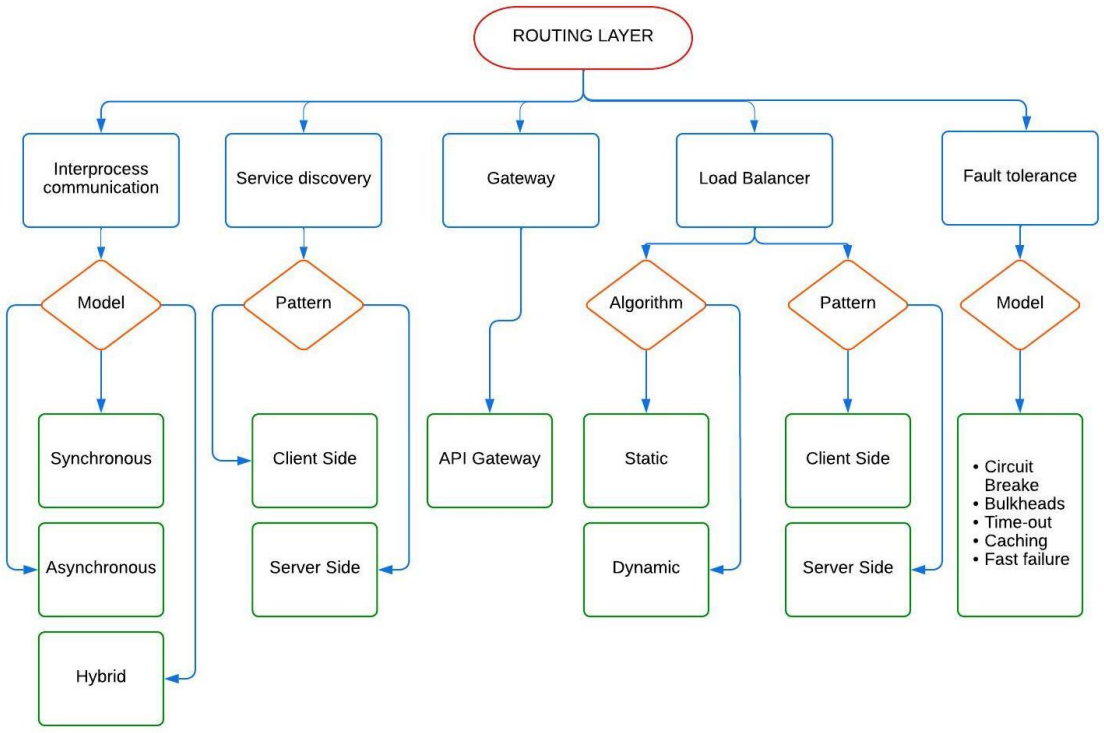


**Рис. 1.** Концептуальна схема проєктування рівня маршрутизації

Для практичного дослідження розглянутого питання було спроєктовано мікросервісну архітектуру, що забезпечує роботу ланцюжка постачань їжі. Система містить чотири служби: *Account, Products, Order, Delivery*. Усі вони є програмами, що розгортаються незалежно. Подана архітектура має

ISSN 2522-9818 (print)
ISSN 2524-2296 (online)
Innovative technologies and scientific solutions for industries. 2023. No. 3 (25)

гібридну міжпроцесну взаємодію, яка містить синхронні та асинхронні зв'язки. Клієнт за допомогою мікросервісу *Account* створює новий або переглядає / оновлює наявний акаунт користувача. Усі акаунти зберігаються у відповідній базі даних. Мікросервіс *Products* також має точку входу для клієнта з метою отримання/створення/оновлення переліку продуктів. Клієнти за допомогою мікросервіса *Order* мають змогу створити нове замовлення, отримати перелік створених замовлень, а також переглянути їх статус (нове, передано в доставку). Ідентифікатор користувача для нового замовлення перевіряється з переліком акаунтів, що отримується за допомогою синхронного запиту до мікросервісу *Account*. Аналогічно, ідентифікатор продукту для нового замовлення перевіряється з переліком продуктів, що отримується з допомогою синхронного запиту до мікросервісу *Products*. Після того як замовлення створено та збережено у базі даних, до служби доставки *Delivery* від служби *Order* асинхронним способом буде відправлено інформацію про замовлення та його адресу. Зв'язок між службою замовлення та службою доставки відбувається через брокер повідомлень. Оскільки служба замовлення є джерелом повідомлення, вона надсилає повідомлення на свій канал запиту *order-request*. З іншого боку, служба доставки прослуховує вхідні повідомлення на каналі *order-request* та публікує статус замовлення в каналі *order-reply*, який слухає служба замовлення. Після певного проміжку часу оновиться статус замовлення на "передано в доставку". Архітектурна модель проєкту наведена на рис. 2.

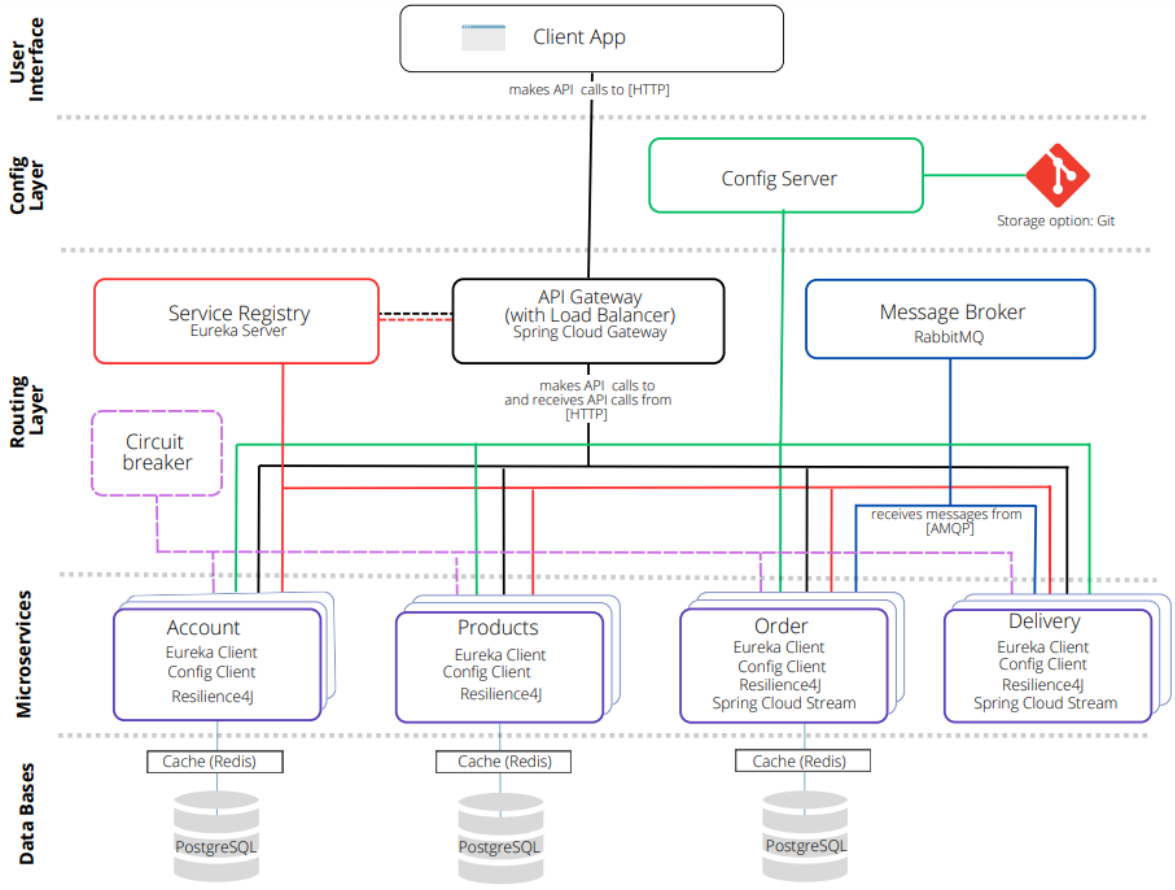


**Рис. 2.** Архітектурна модель проєкту

Під час обрання стеку технологій брався до уваги досвід розробників та ефективність платформ для створення мікросервісної архітектури. Вибір фреймворків ґрунтувався на підтримці мови програмування *Java*. Застосування *Java*-платформ для розроблення мікросервісних проєктів на сьогодні є дуже популярним. Про це свідчить звіт розробників *Java* за 2023 рік від *JRebel by Perforce* [18], згідно з яким програми на основі мікросервісів були найпопулярнішими серед розробників *Java* (32 %). Відповіді показали, що більшість організацій або мали програми, повністю основані на мікросервісах

*Сучасний стан наукових досліджень та технологій в промисловості. 2023. № 3 (25)*
*ISSN 2522-9818 (print)*
*ISSN 2524-2296 (online)*

(31 %), або перебували на стадії планування (31 %), або перейшли на архітектуру мікросервісів (30 %).

Обрання *Spring* як основної платформи для проєктування мікросервісної архітектури ґрунтувалося на аналізі праць [7, 8, 18]. Зі стеку *Spring* застосовуються *Spring Boot* та *Spring Cloud*, що 2023 року залишаються найкращим рішенням для розроблення мікросервісних застосунків (59 %) [18]. Тоді як *Spring Boot* є ефективним способом розроблення автономних застосунків, *Spring Cloud* надає набір інструментів для створення загальних шаблонів у розподілених системах, таких як виявлення служб, керування конфігурацією тощо. Це означає, що *Spring Boot* – радше середовище розроблення повного стеку й має покладатися на *Spring Cloud* для створення особливих можливостей мікросервісів [8].

Для проєктування рівня маршрутизації мікросервісів оберемо відповідні ресурси. Нагадаємо, що до складу шаблонів та компонентів, що відповідають за маршрутизацію мікросервісів, додаємо *Service discovery, API Gateway, Load Balancer*, брокери повідомлень для організації міжпроцесної асинхронної взаємодії, компоненти відмовостійкості. У табл. 1 узагальнено ресурси *Spring* для реалізації маршрутизації в мікросервісних застосунках і які підтримуються на цей час.

**Таблиця 1**. *Ресурси Spring для реалізації маршрутизації в мікросервісних застосунках*

| № | Процеси рівня маршрутизації | Ресурси *Spring* | Особливості |
|---|---|---|---|
| 1 | Міжпроцесна взаємодія | *Spring Cloud Stream* | Опрацювання потокової інформації, пов'язаної із системами обміну повідомленнями *RabbitMQ* або *Apache Kafka* |
| | | *Spring for Apache Kafka* | Асинхронна модель, рекомендується використовувати брокер *Apache Kafka* |
| | | *Spring AMQP* | Асинхронна модель, рекомендується використовувати брокер *RabbitMQ* |
| 2 | Виявлення сервісів | *Spring Cloud Netflix (Eureka)* | Клієнт-серверна архітектура: *Eureka Server*, *Eureka Client* |
| | | *Spring Cloud Zookeeper* | Екземпляри можна зареєструвати в *Zookeeper*, а клієнти можуть виявити екземпляри за допомогою компонентів, керованих *Spring*. Підтримує *Spring Cloud Load Balancer* – клієнтське рішення для балансування навантаження |
| | | *Spring Cloud Consul* | Клієнт для використання *Config Server* та *Service Discovery*, вбудованих у *Consul* |
| 3 | Балансування навантаження | *Spring Cloud Load Balancer* | Забезпечує балансування навантаження під час взаємодії мікросервісів |
| | | *Spring Cloud Gateway* | Має вбудований балансувальник навантаження між екземплярами мікросервісів |
| 4 | Забезпечення відмовостійкості | *Spring Cloud Circuit Breaker* | Підтримує *Resilience4J, Spring Retry* (забезпечує декларативну підтримку повторних спроб для програм *Spring*) |
| 5 | Створенням єдиної точки входу (*Gateway*) | *Spring Cloud Gateway* | Сервіс для маршрутизації вхідних запитів між мікросервісами й балансування між екземплярами мікросервісів. Інтеграція *Spring Cloud Discovery Client* |

*Spring Cloud* містить сервіси, що виконують функції *Service discovery*, а саме *Spring Cloud Netflix (Eureka), Spring Cloud Zookeeper, Spring Cloud Consul.* Відповідно до цього як шаблон виявлення служб будемо застосовувати *Netflix Eureka* [19], що інтегрована в *Spring Cloud. Eureka* – це служба *RESTful* (передача репрезентативного стану), яка використовується для виявлення, балансування навантаження та аварійного перемикання серверів середнього рівня. Цей застосунок призначений для роботи в хмарі *Amazon AWS*, але може бути розгорнутий за її межами.

Крім того, в *Eureka* передбачена можливість розгортання сервісів на різних платформах, що, на нашу думку, є певною перевагою порівняно з таким програмним застосунком, як *Kubernetes*. У разі, коли частина сервісів, що використовуються, розгорнута на *Kubernetes*, а інші сервіси працюють у інших середовищах, виявлення сервісів на рівні застосунків із використанням *Eureka* буде охоплювати всі середовища, тоді як рішення на базі *Kubernetes* сумісні лише з *Kubernetes* [19].

*Eureka* має клієнт-серверну архітектуру. Служба містить *Eureka Server*, а також клієнтську частину на основі *Java*, – *Eureka Client*, яка значно спрощує

ISSN 2522-9818 (print)
ISSN 2524-2296 (online) Innovative technologies and scientific solutions for industries. 2023. No. 3 (25)

взаємодію зі службою. Комунікація за допомогою *Eureka* відбувається таким чином. Сервіси реєструються в *Eureka*, реєстрація служби має короткий час життя *TTL (Time to Live)*, що вимагає від клієнтів постійної взаємодії із серверами. Непрацездатні служби або вузли перестануть працювати, тим самим спричиняють їх тайм-аут й вилучення з реєстру [20].

Клієнт *Eureka Client* має вбудований балансувальник навантаження, що виконує базове циклічне балансування навантаження на основі кількох факторів, таких як трафік, використання ресурсів, умови помилок тощо, що забезпечує гарну відмовостійкість. У цьому сенсі *Eureka* є гарною альтернативою для *AWS Elastic Load Balancer* – балансувальника навантаження для сервісів, що піддаються впливу вебтрафіку кінцевих користувачів. *AWS ELB* є традиційним рішенням для балансування навантаження на основі проксі-сервера [21], тоді як *Eureka* відрізняється тим, що балансування навантаження відбувається на рівні екземпляра/ сервера/хоста. Ще один важливий аспект, що відрізняє балансування навантаження на основі проксі-сервера від балансування навантаження з використанням *Eureka*, полягає в тому, що в останньому випадку програма може бути стійкою до збоїв балансувальників навантаження, оскільки інформація про доступні сервери кешується на клієнті. Це вимагає незначного обсягу пам'яті, але підвищує стійкість до відмов. Крім того, *Eureka* допомагає знайти інформацію про служби, з якими потрібно взаємодіяти, але не накладає жодних обмежень на протокол або спосіб зв'язку. Наприклад, можна використовувати *Eureka* для отримання адреси цільового сервера та застосовувати протоколи HTTP, HTTPS або будь-які інші механізми RPC [19].

Як API-шлюз застосовуватимемо *Spring Cloud Gateway*, що є реалізацією API-шлюзу на основі фреймворку *Spring 5, Project Reactor і Spring Boot 2.0*. Цей неблокувальний шлюз *Spring Cloud Gateway* замінює застарілий *Zuul* від *Netflix* і, на відміну від *Zuul*, підтримує реактивність і *Web Sockets*. *Spring Cloud Gateway* може не тільки ефективно пересилати всі запити від клієнтів, але й виконувати функції, що відповідають за безпеку, моніторинг і поточне обмеження в системі мікросервісу. Ця служба інтегрується з *Circuit Breaker* та *Spring Cloud Discovery Client* [22].

Технологія, що реалізує асинхронний зв'язок – *RabbitMQ*, брокер повідомлень із відкритим вихідним кодом. Це легкий і масштабований брокер повідомлень на основі протоколу AMQP, який діє як посередник між різними сервісами. *RabbitMQ* підтримує різноманітні шаблони обміну повідомленнями, зокрема "точка-точка", "публікація-підписка" та "запит-відповідь". Він також надає розширені функції, такі як маршрутизація повідомлень, підтвердження повідомлень, черги повідомлень, що не доставлені [22].

Для опрацювання потокової інформації застосовуватимемо *Spring Cloud Stream* – платформу для створення високомасштабованих мікросервісів, що керовані подіями, пов'язаних із системами обміну повідомленнями, такими як *RabbitMQ* або *Apache Kafka*. Налаштування брокерів відбувається самим *Spring Cloud Stream*. Зв'язок із брокером здійснюється з допомогою бібліотеки *Stream*. *Spring Cloud Stream* працює в розподілених мікросервісах, що реагують на потоки вхідної інформації. Він створений на основі *Spring Boot*, працює зі *Spring MVC* або *Spring Web Flux* і може використовуватися для розроблення масштабованих програм обміну повідомленнями та потокового оброблення [23].

У наведеному прикладі реалізована асинхронна модель обміну повідомленнями між службою замовлення та службою доставки за допомогою брокера повідомлень *RabbitMQ* та *Spring Cloud Stream*. Відправник повідомлення (служба *Order*) надсилає повідомлення, а потім очікує відповіді від отримувача (служба *Delivery*). У цій моделі відправник піклується про статус повідомлення, а саме отримане воно чи ні.

Фактор відмовостійкості забезпечувався *Spring Cloud Circuit Breaker*, що міститься в складі *Spring Cloud*. *Circuit Breaker* забезпечує абстракцію для різних реалізацій автоматичного вимикача. Він надає послідовний API для використання в програмах, допомагає обрати реалізацію автоматичного вимикача, що найкраще відповідає потребам у програмі [23]. Будемо застосовувати бібліотеку *Resilience4J*, реалізацію якої підтримує *Spring Cloud Circuit Breaker*, і яка є наступницею *Hystrix* від *Netflix*. За допомогою цієї бібліотеки можна реалізувати такі шаблони для використання в службах:

– автоматичний вимикач *circuitbreaker* – припиняє відправлення запитів під час збою служби, що викликається;

– повторні спроби (*retry*) – повторює спроби звернутися до служби;

– герметичні відсіки (модель перегородки) (*bulkhead*) – обмежує кількість конкурентних вихідних запитів, щоб уникнути перевантаження;

– обмежувач частоти (*ratelimit*) – обмежує швидкість вхідних запитів до служби;

*Сучасний стан наукових досліджень та технологій в промисловості. 2023. № 3 (25)*
*ISSN 2522-9818 (print)*
*ISSN 2524-2296 (online)*

– відкат до резервної реалізації (*fallback*) – установлює альтернативні шляхи виконання в разі збою запитів.

*Resilience4J* дає змогу застосувати відразу кілька шаблонів до одного й того самого виклику методу, для чого достатньо забезпечити цей метод відповідними анотаціями.

Крім рівня маршрутизації, наведений приклад мікросервісної архітектури містить конфігураційний рівень, який подано *Spring Cloud Config*, що забезпечує серверну (*Config Server*) та клієнтську (*Config Client*) підтримку зовнішньої конфігурації в розподіленій системі. Служба пропонує інтеграцію із системою контролю версій, щоб зберегти конфігурацію в безпеці [23]. До рівня мікросервісів належать служби *Account, Products, Order, Delivery*. Рівень інформації містить бази даних *PostgreSQL* до сервісів *Account, Products, Order*.

Необхідно зазначити, що традиційні бази даних часто занадто не надійні. Унаслідок цього кожна сучасна розподілена архітектура потребує кешування. Кеш знижує затримку та прискорює взаємодію між сервісами в мікросервісних архітектурах. Кеш – це рівень зберігання інформації з високою швидкістю доступу, на якому розміщена підмножина всіх даних. З кешу отримати їх буде значно швидше, ніж за умови звернення до основного сховища інформації. *Spring Framework* забезпечує підтримку прозорого додавання кешування до програми. У нашому прикладі застосовуватимемо *Redis*.

Запропонована модель мікросервісної архітектури не містить всі компоненти (моніторинг і журналювання, безпекове питання, розгортання тощо), які можуть застосовуватися під час проєктування відповідних застосунків. Це пов'язано з тим, що в розробленій моделі увага приділялася саме дослідженню технологій, пов'язаних з маршрутизацією запитів у мікросервісній архітектурі.

## Висновки й перспективи подальшого розвитку

Під час дослідження розроблено цілісну концепцію проєктування рівня маршрутизації запитів у мікросервісній архітектурі на прикладі стеку технологій *Spring*. Здобуті результати дали змогу дійти певних висновків:

– мікросервісну архітектуру доцільно розглядати як багаторівневу структуру, що будується на функціональних рівнях і зв'язках між ними. Такий підхід забезпечить спрощений спосіб проєктування мікросервісних застосунків завдяки прозорості всіх зв'язків між рівнями та їх компонентами;

– маршрутизацію розглядаємо як багатокомпонентний і водночас цілісний процес. До рівня маршрутизації мікросервісів потрібно додати всі процеси, пов'язані з налагодженням міжпроцесної взаємодії, виявленням сервісів, балансуванням навантаження та забезпеченням відмовостійкості, створенням єдиної точки входу;

– аналіз джерел, що ґрунтуються на практичному досвіді, довів популярність *Java*-платформ, насамперед *Spring*, для розроблення мікросервісних проєктів. *Spring Boot* є ефективним способом створення автономних розподілених застосунків, *Spring Cloud* надає інструменти для реалізації шаблонів мікросервісної архітектури;

– розроблена модель проєкту є прикладом підходів до проєктування багаторівневої архітектури. Ця модель реалізує найбільш ефективні рішення в контексті маршрутизації запитів із застосуванням стеку технологій *Spring*. Так, запропонована архітектура має гібридну міжпроцесну взаємодію, що містить синхронні та асинхронні зв'язки. Такий підхід допомагає реалізувати переваги цих форм зв'язку в одному застосунку. А саме тоді, коли необхідно отримати негайну відповідь на запит, використовуються синхронні блокувальні виклики; у разі тривалих процесів, що не потребують блокування, застосовується асинхронна взаємодія, яка є більш надійним і стабільним механізмом зв'язку, ніж HTTP (*Rest*).

*Spring Cloud* надає широкий спектр шаблонів для реалізації основних функцій маршрутизації запитів у мікросервісній архітектурі. Так, *Eureka* (реалізує *Service discovery* та має вбудований *Load Balancer*), *Gateway* (шлюз), *Circuit Breaker* (механізм відмовостійкості), що інтегровані в *Spring Cloud*, забезпечують необхідний стек технологій на рівні маршрутизації.

Для можливості демонстрації міжрівневих зв'язків у запропонованій архітектурній моделі подано основні рівні (користувача, конфігурації, маршрутизації, мікросервісів, даних), що формують цілісну модель проєкту. Маршрутизація тісно пов'язана з процесами, які відбуваються на інших рівнях. Так, кешування на рівні інформації скорочує затримку оброблення запитів до бази даних

*ISSN 2522-9818 (print)*
*ISSN 2524-2296 (online)* *Innovative technologies and scientific solutions for industries. 2023. No. 3 (25)*

та прискорює взаємодію між сервісами, динамічне налаштування властивостей без перезавантажень застосунків відбувається на рівні конфігурації. Процес керування подіями (повідомленнями) в мікросервісній архітектурі, який реалізується за допомогою *Spring Cloud Stream* і брокера повідомлень *RabbitMQ*, тісно пов'язаний з рівнями мікросервісів та маршрутизації.

Отже, у роботі науково обґрунтовано цілісну концепцію проєктування рівня маршрутизації запитів у мікросервісній архітектурі та запропоновано архітектурну модель на платформі *Spring*, що дасть змогу ефективно реалізовувати міжсервісну комунікацію на основі обраного стеку технологій.

Для подальшого розвитку поставленого завдання необхідно розглянути підходи, що мінімізують ризики під час зміни архітектури проєкту (перехід від моноліту до мікросервісів, перенесення інфраструктури в хмару тощо), проблеми узгодженості інформації під час транзакцій між сервісами тощо.

**Список літератури**

1. Abdelfattah A.S., Cerny T. Roadmap to Reasoning in Microservice Systems: A Rapid Review. *Applied Sciences.* Vol. 13. № 3. 2023. 1838 p. DOI: https://doi.org/10.3390/app13031838

2. Kazanavicius J., Mazeika D. Evaluation of microservice communication while decomposing monoliths. *Computing and Informatics.* Vol. 42. 2023. P. 1–36. DOI: https://doi.org/10.31577/cai_2023_1_1

3. Бугаєва І., Розум М. Реалізація міжпроцесної взаємодії в мікросервісній архітектурі. *Вісник Одеського національного морського університету*. № 67. 2022. P. 81–89. DOI: https://doi.org/10.47049/2226-1893-2022-1-81-89

4. H. Wang et al. Research on load balancing technology for microservice architecture. *MATEC Web of Conferences*. Vol. 336. № 08002. 2021. DOI: https://doi.org/10.1051/matecconf/202133608002

5. Autili M., Perucci A., Lauretis L. Hybrid Approach to Microservices Load Balancing. *Microservices. Science and Engineering.* 2020. P. 149–169. DOI: https://doi.org/10.1007/978-3-030-31646-4_10

6. Akbulut A., Perros H.G. Performance Analysis of Microservice Design Patterns. *IEEE Internet Computing*. Vol. 23. Issue 6. 2019. P. 19–27. DOI: https://doi.org/10.1109/mic.2019.2951094

7. Wycislik L., Latusik L., Kaminska A. M. A Comparative Assessment of JVM Frameworks to Develop Microservices. *Applied Sciences.* Vol. 13. 2023. DOI: https://doi.org/10.3390/app13031343

8. H. Dinh-Tuan et al. Development Frameworks for Microservice-based Applications: Evaluation and Comparison. *In Proceedings of the European Symposium on Software Engineering (ESSE '20). Association for Computing Machinery,* New York, NY, USA, 2020. P. 21–29. DOI: https://doi.org/10.1145/3393822.3432339

9. Zhang Y. et al. Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices. *In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21). Association for Computing Machinery,* New York, NY, USA, 2021. P. 167–181. DOI: https://doi.org/10.1145/3445814.3446693

10. Яшина О. М., Кравчук О. А. Дослідження мікросервісної архітектури, архітектурний стиль Rest та їх сучасна реалізація на Java. *Herald of Khmelnytskyi national university*. Issue 5. 2020. P. 106–114. DOI: 10.31891/2307-5732-2020-289-5-106-114

11. Newman S. Building Microservices: Designing Fine-Grained Systems 2nd Edition. O'Reilly Media, London, 2021. 612 p.

12. Silva V. A. M., França B. B. N. Microservices Design Patterns and Software Evolution. Relatório Técnico - IC-PFG-20-07. Projeto Finalde Graduação. 2020. URL: https://www.ic.unicamp.br/~reltech/PFG/2020/PFG-20-07.pdf (дата звернення: 11.07.2023).

13. Pachikkal C. Interservice Communication in Microservices. International Journal of Advanced Research in Science, Communication and Technology (IJARSCT). Vol. 5. Issue 2. 2021. URL: https://ijarsct.co.in/Paper1281.pdf (дата звернення: 11.07.2023).

14. S. A. Asri et al. Implementation of Asynchronous Microservices Architecture on Smart Village Application. *International Journal on Advanced Science Engineering and Information Technology.* Vol. 12. № 3. P. 1236–1243. DOI:10.18517/ijaseit.12.3.13897

15. Reselman B. Synchronous vs. asynchronous microservices communication patterns. URL: https://www.theserverside.com/answer/Synchronous-vs-asynchronous-microservices-communication-patterns (дата звернення: 11.07.2023).

16. Sanjana G B., Girish Rao Salanke N S. High Resilient Messaging Service for Microservice Architecture. *International Journal of Applied Engineering Research.* Vol. 16. № 5. 2021. P. 357–361. DOI: https://dx.doi.org/10.37622/IJAER/16.5.2021.357-361

17. Hosea E., Palit H, Dewi L. P. Fault Tolerance pada Microservice Architecture dengan Circuit Breaker dan Bulkhead Pattern. Jurnal Infra. Vol 9, № 2. 2021. URL: https://publication.petra.ac.id/index.php/teknik-informatika/article/view/11452/10062 (дата звернення: 11.07.2023).

18. Java Development Trends and Analysis. JRebel by Perforce. 2023 Java Developer Productivity Report. URL: https://www.jrebel.com/resources/java-developer-productivity-report-2023 (дата звернення: 11.07.2023).

19. Netflix Open Source Software Center. URL: https://netflix.github.io/ (дата звернення: 11.07.2023).

20. Wang Y. Toward service discovery and autonomic version management inself-healing microservices architecture. *ECSA '19: Proceedings of the 13th European Conference on Software Architecture*. Vol.2. 2019. P. 63–66. https://doi.org/10.1145/3344948.3344952

*ISSN 2522-9818 (print)*
*Сучасний стан наукових досліджень та технологій в промисловості. 2023. № 3 (25)* *ISSN 2524-2296 (online)*

21. Indrani V., Swaroopa D., Deepthisri D. Perceptions of Client and Server Side Load Balancing in Microservices. International Journal of Innovative Research in Engineering & Management (IJIREM). Vol. 7. Issue 4. 2020. P. 54–57. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3703126 (дата звернення: 11.07.2023).

22. Maharjan R. et al. Benchmarking Message Queues. *Telecom.* Vol. 4. 2023. P. 298–312. DOI: https://doi.org/10.3390/telecom4020018

23. Spring by VMware Tanzu. URL: https://spring.io/projects/spring-cloud (дата звернення: 11.07.2023).

## References

1. Abdelfattah, A.S., Cerny, T. (2023), "Roadmap to Reasoning in Microservice Systems: A Rapid Review". *Applied Sciences.* Vol. 13. № 3. 1838 p. DOI: https://doi.org/10.3390/app13031838

2. Kazanavicius, J., Mazeika, D. (2023), "Evaluation of microservice communication while decomposing monoliths". *Computing and Informatics.* Vol. 42. P. 1–36. DOI: https://doi.org/10.31577/cai_2023_1_1

3. Buhaieva,, I., Rozum, M. (2022), "Implementation of interprocess interaction in microservice architecture". ["Realizatsiia mizhprotsesnoi vzaiemodii v mikroservisnii arkhitekturi"], *Bulletin of Odessa National Maritime University*, № 67. P. 81–89. DOI: https://doi.org/10.47049/2226-1893-2022-1-81-89

4. Wang, H. et al. (2021), "Research on load balancing technology for microservice architecture". *MATEC Web of Conferences*. Vol. 336. № 08002. DOI: https://doi.org/10.1051/matecconf/202133608002

5. Autili, M., Perucci, A., Lauretis, L. (2020), "Hybrid Approach to Microservices Load Balancing". *Microservices. Science and Engineering.* P. 149–169. DOI: https://doi.org/10.1007/978-3-030-31646-4_10

6. Akbulut, A., Perros, H.G. (2019), "Performance Analysis of Microservice Design Patterns". *IEEE Internet Computing*. Vol. 23. Issue 6. P. 19–27. DOI: https://doi.org/10.1109/mic.2019.2951094

7. Wycislik, L., Latusik, L., Kaminska, A. M. (2023), "A Comparative Assessment of JVM Frameworks to Develop Microservices". *Applied Sciences.* Vol. 13. DOI: https://doi.org/ 10.3390/app13031343

8. Dinh-Tuan, H. et al. (2020), "Development Frameworks for Microservice-based Applications: Evaluation and Comparison". *European Symposium on Software Engineering (ESSE '20). Association for Computing Machinery,* New York, NY, USA, P.21–29 DOI: https://doi.org/10.1145/3393822.3432339

9. Zhang, Y. et al. (2021), "Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices". *In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21). Association for Computing Machinery*, New York, NY, USA, P. 167–181. DOI: https://doi.org/10.1145/3445814.3446693

10. Yashina, O.M., Kravchuk, O.A. (2020), "Research on microservice architecture, Rest architectural style and their modern implementation in Java". ["Doslidzhennia mikroservisnoi arkhitektury, arkhitekturnyi styl Rest ta yikh suchasna realizatsiia na Java"], *Herald of Khmelnytskyi national university*. Issue 5. P. 106–114. DOI: 10.31891/2307-5732-2020-289-5-106-114

11. Newman, S. (2021), *Building Microservices: Designing Fine-Grained Systems 2nd Edition*. O'Reilly Media, London, 612 p.

12. Silva, V. A. M., França, B. B. N. (2020), "Microservices Design Patterns and Software Evolution". Relatório Técnico - IC-PFG-20-07. Projeto Finalde Graduação, available at: https://www.ic.unicamp.br/~reltech/PFG/2020/PFG-20-07.pdf (last accessed: 11.07.2023).

13. Pachikkal, C. "Interservice Communication in Microservices". International Journal of Advanced Research in Science, Communication and Technology (IJARSCT). Vol. 5. Issue 2, available at: https://ijarsct.co.in/Paper1281.pdf (last accessed: 11.07.2023).

14. Asri, S.A. et al. (2020), "Implementation of Asynchronous Microservices Architecture on Smart Village Application". *International Journal on Advanced Science Engineering and Information Technology*. Vol. 12. № 3. P. 1236–1243. DOI:10.18517/ijaseit.12.3.13897

15. Reselman, B. Synchronous vs. asynchronous microservices communication patterns, available at: https://www.theserverside.com/answer/Synchronous-vs-asynchronous-microservices-communication-patterns (last accessed: 11.07.2023).

16. Sanjana, G B., Girish Rao Salanke, N S. (2021), "High Resilient Messaging Service for Microservice Architecture". *International Journal of Applied Engineering Research.* Vol. 16. № 5. P. 357–361. DOI: https://dx.doi.org/10.37622/IJAER/16.5.2021.357-361

17. Hosea, E., Palit, H, Dewi, L. P. (2021), "Fault Tolerance pada Microservice Architecture dengan Circuit Breaker dan Bulkhead Pattern". Jurnal Infra. Vol 9, № 2, available at: https://publication.petra.ac.id/index.php/teknik-informatika/article/view/11452/10062 (last accessed: 11.07.2023).

18. "Java Development Trends and Analysis. JRebelby Perforce". 2023 Java Developer Productivity Report, available at: https://www.jrebel.com/resources/java-developer-productivity-report-2023 (last accessed: 11.07.2023).

19. "Netflix Open Source Software Center", available at: https://netflix.github.io/ (last accessed: 11.07.2023).

20. Wang, Y. (2019), "Toward service discovery and autonomic version management inself-healing microservices architecture". *ECSA '19: Proceedings of the 13th European Conference on Software Architecture*. Vol.2. P. 63–66. https://doi.org/10.1145/3344948.3344952

*ISSN 2522-9818 (print)*
*ISSN 2524-2296 (online)* *Innovative technologies and scientific solutions for industries. 2023. No. 3 (25)*

21. Indrani, V., Swaroopa, D., Deepthisri, D. (2020), "Perceptions of Client and Server Side Load Balancing in Microservices". International Journal of Innovative Research in Engineering & Management (IJIREM). Vol. 7. Issue 4, P. 54–57, available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3703126 (last accessed: 11.07.2023).

22. Maharjan, R. et al. (2023), "Benchmarking Message Queues". *Telecom.* Vol. 4. P. 298–312. DOI: https://doi.org/10.3390/telecom4020018

23. "Spring by VMware Tanzu", available at: https://spring.io/projects/spring-cloud (last accessed: 11.07.2023).

*Надійшла 14.08.2023*

*Відомості про авторів / About the Authors*

**Переяславська Світлана Олександрівна** – кандидат педагогічних наук, доцент, Державний заклад "Луганський національний університет імені Тараса Шевченка", доцент кафедри інформаційних технологій та систем, Полтава, Україна; e-mail: pereyaslav9@gmail.com; ORCID ID: https://orcid.org/0000-0001-9873-0447

**Смагіна Ольга Олександрівна** – кандидат педагогічних наук, доцент, Державний заклад "Луганський національний університет імені Тараса Шевченка", доцент кафедри інформаційних технологій та систем, Полтава, Україна; e-mail: smagina1804@gmail.com; ORCID ID: https://orcid.org/0000-0002-6024-5152

**Pereiaslavska Svitlana** – PhD (Pedagogical Sciences), Associate Professor, State Institution "Luhansk Taras Shevchenko National University", Associate Professor at the Department of Information Technologies and Systems, Poltava, Ukraine.

**Smahina Olga** – PhD (Pedagogical Sciences), Associate Professor, State Institution "Luhansk Taras Shevchenko National University", Associate Professor at the Department of Information Technologies and Systems, Poltava, Ukraine.

# DESIGNING THE ROUTING LEVEL IN MICROSERVICE ARCHITECTURES ON THE SPRING PLATFORM

The **subject matter** of research is the routing of requests in the microservice architecture. The **goal** of the article is to develop a target design concept for the level of request routing in the microservice architecture using the Spring technology stack as an example. **Tasks**: to analyse modern approaches to the structure of microservice architecture; programming of the routing entity and establishment of processes that ensure the routing of requests; programming stacks of Spring technologies that implement routing; design the routing layer of the application on the Spring platform. The following **methods** are used: analysis and synthesis to study technologies of interaction between services; abstraction and generalization to determine the structure of the microservice architecture, routing level, generalization of technologies that ensure interaction between services; modelling for the purpose of building a model of microservice architecture, highlighting the level of routing and connections with other structural models. The following **results** were obtained: the structure of the microservice architecture was investigated, in particular the level of routing; the role of design patterns that provide routing is defined: Service discovery, API Gateway, Load Balancer, etc.; the types of interprocess interaction (synchronous, asynchronous, hybrid) were analysed and the advantages and expediency of the application were determined; models of system fault tolerance are presented; a stack of technologies on the Spring platform is defined for the implementation of the routing layer; a model of a multi-level microservice architecture project was developed using the Spring technology stack, which implements the most effective solutions in the context of request routing. **Conclusions**: it is advisable to consider microservice architecture as a multi-level structure built on functional levels and connections between them; the level of microservices routing should include all processes related to the establishment of interprocess interaction, service detection, load balancing and fault tolerance, and the creation of a single entry point; Spring is a popular microservice architecture development tool platform that provides requirements for implementing request routing; the developed project model is an example of effective solutions for designing a multi-level architecture using the Spring technology stack in the context of request routing.

**Keywords:** microservice architecture; routing; Spring; Service discovery; API Gateway; Load Balancer.

*Бібліографічні описи / Bibliographic descriptions*

Переяславська С. О., Смагіна О. О. Проєктування рівня маршрутизації в мікросервісних архітектурах на платформі Spring. *Сучасний стан наукових досліджень та технологій в промисловості*. 2023. № 3 (25). С. 64–78. DOI: https://doi.org/10.30837/ITSSI.2023.25.064

Pereiaslavska, S., Smahina, O. (2023), " Designing the routing level in microservice architectures on the Spring platform", *Innovative Technologies and Scientific Solutions for Industries*, No. 3 (25), P. 64–78. DOI: https://doi.org/10.30837/ITSSI.2023.25.064