DOI 10.31891/2307-5732-2020-289-5-106-114
УДК 004.9

О.М. ЯШИНА, О.А. КРАВЧУК
Хмельницький національний університет

# ДОСЛІДЖЕННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ, АРХІТЕКТУРНИЙ СТИЛЬ REST ТА ЇХ СУЧАСНА РЕАЛІЗАЦІЯ НА JAVA

*В статті наведено результати дослідження мікросервісної архітектури, наведені її переваги та недоліки, проведено її порівняння з загальноприйнятим підходом до розробки програмного забезпечення, описаний архітектурний підхід REST, який найчастіше використовують з досліджуваною архітектурою. Реалізована програма на Java є прикладом дотримання визначених понять за допомогою сучасних інструментів розробки.*
*Ключові слова: мікросервісна архітектура, REST, Java.*

O.M. YASHYNA, O.A. KRAVCHUK
Khmelnytskyi national University

## RESEARCH OF MICROSERVICE ARCHITECTURE, REST ARCHITECTURAL STYLE AND THEIR MODERN IMPLEMENTATION ON JAVA

*Software architecture is a set of important decisions about the organization of the software system. The success and speed of the project development, the complexity of its support and understanding of the program structure depends on the correct choice of software architecture. For this reason, developers often use new architectural solutions to build software, trying to find the best solutions to a problem. The common practice of program development was the development on the principle of monolithic architecture, which envisages the project as a single program, which is responsible for all necessary functionality. Of course, this approach had its advantages, such as ease of development, testing and deploying. But with project`s growth, the monolithic architecture begins to make some flaws in the development process: the addition of a certain new functionality is accompanied by an increase in the code base, observance of limits of responsibility of internal modules is blurred, compiling and deploying takes much more time. Given the drawbacks of monolithic architecture above, developers have begun to take other approaches. One is microservice architecture. Microservice Architecture - A variant of service architecture (modular software development approach) software, focused on the interaction of as small as possible, loosely coupled and easily replaceable modules – microservices [1]. With such an architecture, the project is a set of small services, the communication between which is done through lightweight mechanisms (for example, HTTP, which can be used in REST architectural style to get an increased productivity and simplified architecture) [2]. Each of these services must fulfil its clearly defined business task. This architecture has the following advantages: better scaling, independent module development and deploying, better reusability etc.*
*Keywords: microservice architecture, REST, Java*

### Постановка проблеми

Архітектура програмного забезпечення є сукупністю важливих рішень про організацію програмної системи. Від правильного вибору архітектури залежить успішність та швидкість розробки проєкту, складність його підтримки та розуміння структури програми. Саме через це розробники часто використовують нові архітектурні рішення для побудови програмного забезпечення, намагаючись знайти оптимальні варіанти вирішення тієї чи іншої проблеми.

### Аналіз останніх досліджень та публікацій

Методологічною основою нашого дослідження стали праці таких дослідників як Кріс Річардсон, Мартін Фаулер, Рой Філдінг, Філ Кальчадо та багато інших. Загалом ці дослідники описують REST, мікросервісну архітектуру, її переваги та недоліки над іншими, а також застосування її кращих сторін на практиці.

### Формулювання цілей статті

Метою статті є визначення переваг та недоліків мікросервісної архітектури, визначення поняття архітектурного стилю REST та створення програми на Java за допомогою вище вказаних підходів.

### Виклад основного матеріалу

Загальноприйнятою практикою розробки програм була розробка за принципом монолітної архітектури, яка передбачає проєкт як одну програму, яка відповідає за весь необхідний функціонал. І, безумовно, такий підхід мав свої переваги:

- простота розробки – IDE (середовище розробки) та інші інструменти для створення програмного забезпечення сфокусовані на побудові проєкту як одного цілого;
- легкість тестування – проєкт легше тестується, коли весь функціонал знаходиться в одній програмі;
- простота розгортання – проєкт легко розгорнути, оскільки він складається з одного файлу (наприклад, WAR, JAR тощо).

Але, при розвитку та збільшенні проєкту, монолітна архітектура розпочинає вносити певні недоліки в процес розробки:

- додавання певного нового функціоналу супроводжується наростанням кодової бази;
- швидкість розробки сповільнюється через необхідність внесення змін в одну велику систему;
- дотримання границь відповідальності внутрішніх модулів розмивається;
- будь-яка зміна проєкту вимагає його перекомпіляції;
- компіляція, збирання та розгортання програми займає більше часу;

- розуміння проєкту та його архітектури ускладнюються через збільшення його розмірів, новим розробникам потрібно більше часу для інтеграції.

Враховуючи вище наведені недоліки монолітної архітектури, розробники почали використовувати інші підходи. Одним із них є мікросервісна архітектура.

Мікросервісна архітектура – варіант сервісної архітектури (модульний підхід ро розробки програм) програмного забезпечення, орієнтований на взаємодію максимально, наскільки це можливо, невеликих, слабо пов'язаних і легко замінних модулів – мікросервісів [1]. При такій архітектурі проєкт являє собою набір невеликих сервісів, комунікація між якими відбувається за допомогою легких механізмів (наприклад, HTTP) [2]. Кожен з таких сервісів повинен виконувати свою чітко визначену бізнес-задачу. Розглянемо різницю монолітної та мікросервісної архітектури на рисунку, наведеному нижче (рис. 1):

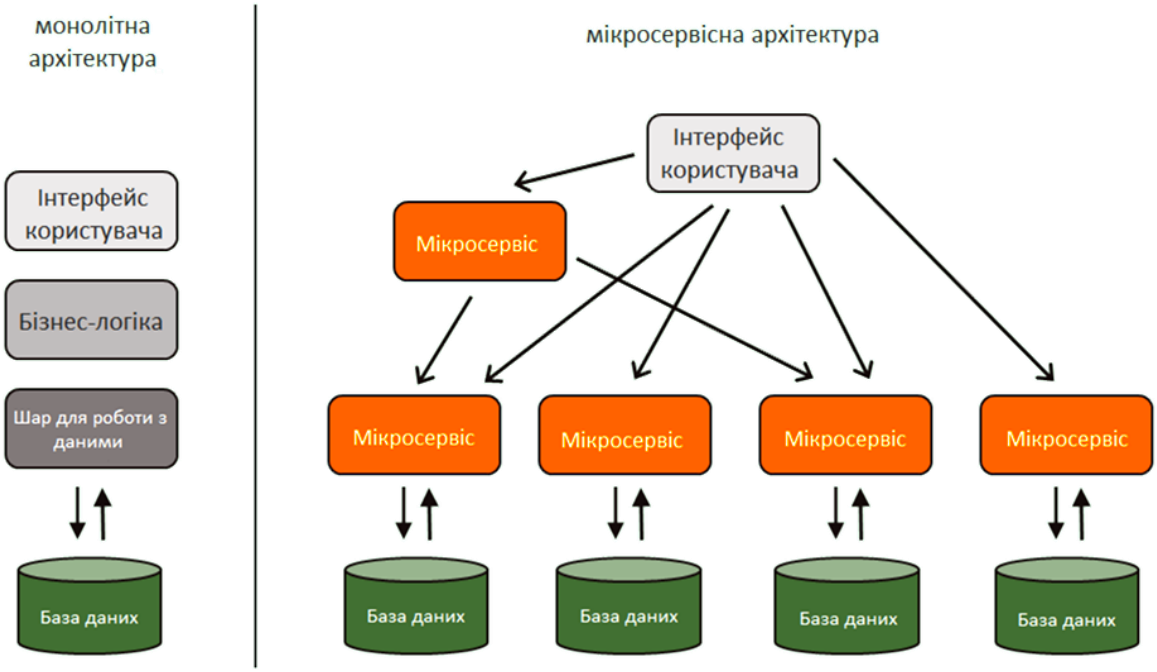


**Рис. 1. Монолітна та мікросервісна архітектури**

Серед переваг такої архітектури можна виділити наступні [3]:

- незалежна розробка – невеликі за розмірами незалежні компоненти можуть створюватися малими незалежними командами розробників. Група розробників може працювати над змінами/розробкою одного сервісу, не змінюючи і навіть не знаючи про інший сервіс. Кількість часу, необхідна на вивчення сервісу значно зменшується, і реалізовувати нові функції стає значно простіше;
- незалежне розгортання – кожен окремий компонент проєкту можна запускати незалежно від інших. Це дозволяє випускати новий функціонал швидше і з меншими ризиками. Виправляючи помилки в одному сервісі, його можна перезапускати без необхідності перезапускати інші сервіси;
- незалежне масштабування – кожен сервіс можна масштабувати незалежно від іншого. Наприклад, випускаючи новий функціонал, для збільшеного навантаження на сервери можна відмасштабувати лише сервіси, які будуть використовуватися користувачами активніше. Це дозволяє зменшити витрати та виконувати масштабування гнучкіше, ніж при монолітній архітектурі. Також, для кожного сервісу можна підібрати найкращу конфігурацію ресурсів (пам'яті, процесора тощо), в той час як монолітна система буде працювати на одному комп'ютері;
- можливість повторного використання – кожен компонент реалізує свою чітко визначену бізнес-задачу. За рахунок цього, кожен сервіс можна використовувати в інших проєктах, виконуючи менше змін для адаптації;
- краща ізоляція неполадок – в мікросервісній архітектурі краще ізольовані неполадки. Наприклад, витік пам'яті в одного сервіса вплине лише на нього, в той час як інші сервіси продовжать працювати в звичайному режимі. В монолітній архітектурі така проблема виведе з роботи всю систему;
- краща здатність до залучення нових технологій – кожен сервіс може використовувати інший набір технологій, іншу мову розробки тощо. Якщо в випадку невдалого використання технології в мікросервісній архітектурі необхідно змінити лише компоненти, які використовують цю технологію, в монолітній системі необхідно буде переписувати всю програму.

Однак, при всіх позитивних моментах, мікросервісна архітектура має також недоліки:

- збільшена складність для розробників – якщо розробнику необхідно виконати завдання, яке пов'язане з декількома сервісами, зазвичай це збільшує складність розробки, оскільки йому необхідно буде конфігурувати та запускати ці декілька сервісів на своєму комп'ютері, що є важчим ніж запуск однієї програми;

- складність розподілених систем – у випадку взаємодії декількох сервісів необхідно бути готовим до часткових збоїв, високої латентності та недоступності інших віддалених сервісів;
- складність тестування – зростає складність у написанні автоматичних інтеграційних тестів, оскільки система складається з незалежних між собою сервісів;
- зростання складності експлуатації – команді, яка займається експлуатацією та підтриманням сервісів у робочому стані буде важче виконувати свої обов'язки. Замість однієї програми необхідно буде контролювати певну кількість сервісів, і чим їх більше, тим більша кількість потенційних проблемних місць;
- необхідна серйозна компетентність – розробка проєкту за допомогою мікросервісної архітектури може дати продуктивні результати, якщо роботу виконували професіонали. Однак, маючи низький рівень освіченості, розробка таких проєктів може привести до заплутаності в організації сервісів та неефективності результату роботи;
- необхідність встановлення чітких границь – для кожного сервісу повинні бути визначені чіткі границі його бізнес-задачі. Якщо цього не буде зроблено, тоді залежності між сервісами будуть зростати, що може привести до того, що через ці самі залежності необхідно буде запускати певні сервіси як одну єдину групу, де один сервіс не може функціонувати без іншого. В такому випадку систему можна назвати розподіленим монолітом.

Зважаючи на такі недоліки може виникнути питання, чи варто взагалі переходити від звичної монолітної архітектури на мікросервіси? У відповідь на це, розглянемо рішення відомих компаній [4]:

- Amazon - "Якщо повернутися до 2001 року, веб-сайт Amazon.com був великим архітектурним монолітом", - Боб Брігхем, спікер від Amazon на конференції Amazon's: Invent 2015. Тоді Amazon боровся з управлінням своєї швидко зростаючої популярності та важливості в Інтернеті. Брігхем також пояснив, що компанія "зауважила, що потрібен тривалий час, щоб зміна коду перейшла від реалізації розробників до запуску у виробництво, де клієнти могли її використовувати". На сьогоднішній день Amazon є одним з найбільших прихильників мікросервісів, допомагаючи іншим компаніям з запуском, керуванням та аналізом бізнес-рішень у вигляді мікросервісів за допомогою Amazon Web Services (AWS);
- Netflix – У 2009 році Netflix прийняв рішення про розподіл моноліту на мікросервіси, коли їх розробки з монолітною архітектурою спричинили проблеми на шляху зростання технологічного гіганта. Також, попередня архітектура приводила до регулярних відключень серверів. Прийняття мікросервісної архітектури було одним із факторів, що призвели до феноменального зростання Netflix. Сьогодні ця компанія є одним з найвідоміших прикладів успішного застосування даної архітектури;
- Sound cloud – На сьогоднішній день, на цю платформу завантажується близько 12 годин музики та звуку щохвилини, їх сервіси використовують сотні мільйонів людей щодня. Щоб масштабувати свій проєкт, компанія розробила та опублікувала декілька компонентів та інструментів, які допомагають запускати міграцію баз даних у потрібному масштабі, провела серйозний аналіз щодо того, як фреймворк, на якому був написаний проєкт, отримує доступ до баз даних та обробляє величезну кількість повідомлень. Врешті-решт компанія вирішила кардинально змінити спосіб розробки проєктів, оскільки команда "відчувала, що завжди латала систему, та не вирішувала основної проблеми масштабування" [5]. Таким чином, вирішуючи цю проблему компанія перейшла до мікросервісної архітектури.

Тепер розглянемо не менш важливий чинник успішності проєкту з такою архітектурою – передача даних. Оскільки проєкт з мікросервісною архітектурою складається з певної кількості окремих сервісів, вони повинні між собою реалізовувати механізм передачі інформації. На сьогоднішній день найбільш широко для цієї мети використовується підхід REST.

REST (Representational state transfer) – архітектурний стиль взаємодії компонентів розподіленого програмного забезпечення в мережі [6]. Як правило, такий підхід використовується для побудови веб-сервісів. REST являє собою набір погоджених обмежень, при дотриманні яких підвищується продуктивність та спрощується архітектура. Системи, які підтримують цей підхід називаються RESTful-системами. Щоб система вважалась сконструйованою в цьому архітектурному стилі, вона повинна задовольняти наступні критерії:

- Client-Server. Система повинна бути розділена на клієнтів і на серверів. Поділ інтерфейсів означає, що, наприклад, клієнти не пов'язані зі зберіганням даних, яке залишається всередині кожного сервера, так що мобільність коду клієнта поліпшується. Сервери не пов'язані з інтерфейсом користувача або станом, так що сервери можуть бути простішими і більш масштабованими. Сервери та клієнти можуть бути замінні і розроблятися незалежно, допоки інтерфейс не змінюється.
- Stateless. Сервер не повинен зберігати будь-яку інформацію про клієнтів. У запиті повинна зберігатися вся необхідна інформація для обробки запиту і якщо необхідно, ідентифікації клієнта. Стан сесії при цьому зберігається на стороні клієнта. Інформація про стан сесії може бути передана сервером будь-якому іншому сервісу (наприклад, в службу бази даних) для підтримки стійкого стану користувача.
- Cache. Кожна відповідь повинна бути зазначена чи є він кешованою чи ні, для запобігання повторного використання клієнтами застарілих або некоректних даних у відповідь на подальші запити.
- Uniform Interface. Визначається єдиний інтерфейс між клієнтами і серверами. Це спрощує і відокремлює архітектуру, яка дозволяє кожній частині розвиватися самостійно. Щоб задовольняти цей критерій, необхідно виконувати наступні 4 умови:

1) ідентифікація ресурсів – кожен ресурс (користувач, товар тощо) в REST повинен бути

ідентифікованийза допомогою стабільного ідентифікатора, який не зміниться при зміні стану ресурса. Наприклад, таким ідентифікатором може виступати URI (Uniform Resource Identifier, ідентифікатор ресурса);

2) маніпуляція ресурсами через представлення – якщо клієнт зберігає представлення ресурса, то він може його змінити чи видалити;

3) само-описувані повідомлення – кожне повідомлення повинно містити в собі достатньо інформації, щоб було зрозуміло, як його обробити;

4) HATEOAS (гіпермедіа як засіб зміни стану додатка) - Статус ресурсу передається через вміст тіла запиту, параметри рядка запиту, заголовки запитів і запитуваний URI. Це називається гіпермедіа (або гіперпосилання з гіпертекстом).

- Шарування системи – клієнт зазвичай не здатний точно визначити, взаємодіє він безпосередньо з сервером або ж з проміжним вузлом, в зв'язку з ієрархічною структурою мереж (маючи на увазі, що така структура утворює шари). Застосування проміжних серверів здатне підвищити масштабованість за рахунок балансування навантаження і розподіленого кешування. Кожен компонент системи може бачити лише компонент наступного шару. Наприклад, якщо клієнт хоче отримати відео з сервісу, який в свою чергу отримує його з віддаленого репозиторія, то клієнт нічого не повинен знати про віддалений репозиторій.

Також варто розуміти, що підхід REST не прив’язаний до конкретних технологій та протоколів, однак на практиці майже завжди використовуючи цей підхід передбачається використання HTTP протоколу.

HTTP (англ. HyperText Transfer Protocol – «протокол передачі гіпертексту») – протокол прикладного рівня передачі даних. Спочатку дані передавались лише у вигляді гіпертекстових документів в форматі «HTML» [7]. На сьогоднішній день цей протокол використовується для передачі довільних даних. Основою HTTP є технологія «клієнт-сервер», для якої є необхідне існування клієнта, який буде ініціювати з’єднання і відправляти запит, і сервера, який очікує новий запит, виконує певні дії та надсилає відповідь з результатом.

Основним об'єктом маніпуляції в HTTP є ресурс, на який вказує URI в запиті клієнта. Зазвичай такими ресурсами є файли, що зберігаються на сервері, але ними можуть бути і інші логічні об’єкти (наприклад, записи в базі даних).

Розглянемо структуру повідомлення, яке клієнт відсилає серверу. Воно складається з трьох частин:

- стартова стрічка – визначає тип повідомлення. У собі містить метод, який визначає тип запиту, URI, який є адресом до якого буде надісланий запит, та версія протоколу HTTP;
- заголовки – інформація, яка характеризує тіло повідомлення, описує його формат тощо;
- тіло запиту – частина повідомлення, у якій безпосередньо містяться дані, які необхідно передати. Не є обов’язковою частиною запиту.
- Для взаємодії з сервером з архітектурним підходом REST зазвичай використовують наступні 4 методи:
- GET – використовується для запиту вмісту заданого ресурсу. За допомогою цього методу також можна розпочати якийсь процес. При використанні цього метода можна використовувати параметри, які додаються до URI після знаку ‘?’.
- POST – використовується для передачі даних користувача вказаній адресі. Зазвичай, використовується для додавання нових об’єктів/ресурсів (наприклад, новий коментар, товар, замовлення тощо).
- PUT – використовується для завантаження вмісту запита на вказаний URI. Різниця між цим методом і POST полягає в тому, що URI ресурс в випадку POST запиту буде оброблювати дані, передані клієнтом, в той час як з методом PUT клієнт передбачає, що завантажуваний вміст вже відповідає тому, що знаходиться за даним URI ресурсу. Зазвичай використовується для оновлення об’єктів/ресурсів.
- DELETE – слугує для видалення ресурсу.

Працюючи з базами даних, легко поставити у відповідність чотирьом базовим функціям для роботи з базою даних (CRUD – create, read, update, delete) методи HTTP протоколу, які використовуються у REST: create - POST, read-GET, update - PUT та delete-DELETE. Також, говорячи про HTTP, варто відзначити популярні формати передачі інформації в тілі запита. На сьогодні, є два основні формати: XML та JSON.

XML (Extensible Markup Language) – розширювана мова розмітки, запропонована як стандарт побудови мов розмітки ієрархічно структурованих даних для обміну між різними застосунками, зокрема, через Інтернет. XML розроблялась як мова з простим формальним синтаксисом, зручним для створення і обробки документів програмами і одночасно зручним для читання і створення документів людиною. Розширення XML – це конкретна граматика, створена на базі XML і представлена словником тегів і їх атрибутів, а також набором правил, що визначають які атрибути і елементи можуть входити до складу інших елементів.

JSON (JavaScript Object Notation) – текстовий формат обміну даними, заснований на JavaScript. Як і багато інших текстові форматів, JSON легко читається людьми. За рахунок своєї лаконічності в порівнянні з XML, формат JSON може бути більш підходящим для серіалізації складних структур. Застосовується в веб-додатках як для обміну даними між клієнтом і сервером, так і між серверами.

Розглянемо приклад одного й того самого списку студентів у описаних вище форматах:

Як видно, формат JSON є набагато простішим для сприйняття та розуміння ніж XML. Це є однією з причин, чому він поступово витісняє свого конкурента з ринку.

XML

```
 <?xml version="1.0" encoding="UTF-8"?>
<students>
      <student>
         <Ім`я>Андрій</Імя>
         <Прізвище>Мельник</Прізвище>
      </student>
      <student>
         <Ім`я>Василь</Імя>
         <Прізвище>Ломаченко</Прізвище>
      </student>
      <student>
         <Ім`я>Олександр</Імя>
         <Прізвище>Усик</Прізвище>
      </student>
</students>
```

JSON

```
{"students":[
  { "Ім`я":"Андрій", "Прізвище":"Мельник" },
  { "Ім`я":"Василь", "Прізвище":"Ломаченко" },
  { "Ім`я":"Олександр", "Прізвище":"Усик" }
]}
```

Перейдемо до розробки показового сервісу, який притримується обмежень стилю REST. Побудуємо простий мікросервіс, який буде виконувати CRUD операції над об'єктами User (створимо таблицю в базі даних, яка буде зберігати наші записи). Для передачі даних будемо використовувати протокол HTTP, формат даних JSON (як було показано вище, він є легшим для розуміння ніж XML).

Створювати програму будемо на мові Java, оскільки вона містить велику кількість інструментів для швидкої розробки програмного забезпечення. Одним з таких інструментів є Spring Framework - універсальний фреймворк з відкритим кодом для Java-платформ, який дозволяє легко та швидко створювати додатки в стилі REST. Підключити його до проєкту можна за допомогою спеціальних інструментів, таких як Gradle або Maven. За допомогою цього фреймворку ми створимо клас-контролер UserController, який буде приймати HTTP-запити, та передавати інформацію з запиту у клас-сервіс UserService. В класі-сервісі в залежності від обраного методу та переданих параметрів будуть виконуватись певні дії над даними, які будуть представлені у вигляді об'єктів класу User, які зберігаються в базі даних. Підключення та робота з базою даних буде реалізована за допомогою інтерфейсу UserRepository. Після того, як UserService виконає роботу він поверне результат у контролер, який в свою чергу поверне його користувачу. При запуску програми, виконувати HTTP-запити необхідно виконувати на адрес localhost:8080 (localhost у випадку локального звернення до сервісу, 8080-стандартний порт на якому Spring запускає програму)

Створимо в базі даних таблицю user з полями id, email, name, number (за допомогою MySQL Workbench, рис. 2):

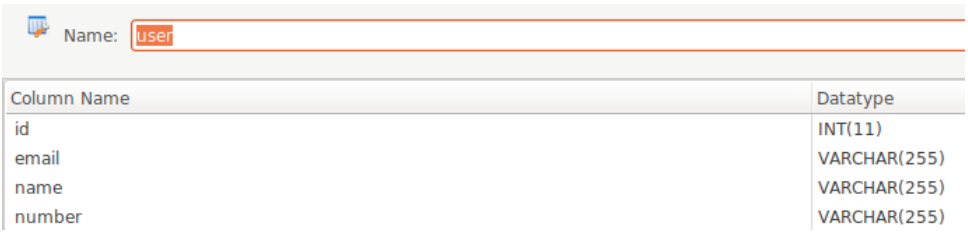


**Рис. 2. Структура таблиці user**

Перейдемо до створення класів, які є необхідними для роботи додатку. Імпорти функцій, класів, конструктори, методи get/set для полів класів не включені в наведений нижче код задля більшої компактності коду. Клас User виглядає наступним чином:

```
@Entity
@Table(name="user")
public class User {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Integer id;
        private String name;
        private String number;
        private String email;
}
```

Анотація @Entity вказує, що цей об'єкт буде використаний як репрезентація об'єкта з бази даних. @Table з параметром name вказує назву таблиці, з якою об'єкт буде пов'язаний. Анотації @Id та @GeneratedValue (strategy = GenerationType.*IDENTITY*) вказують, що поле Integer id є ідентифікатором у базі

даних, який генерується автоматично на стороні БД. Окрім id, клас містить поля, які відповідають полям таблиці в базі даних.

Інтерфейс UserRepository:

```
public interface UserRepository extends CrudRepository<User, Integer> {
}
```

Наслідування класу CrudRepository – єдине, що необхідно для з'єднання з базою даних. Всю роботу за нас виконує Spring за допомогою наслідуваного класу. Generics типи для класу, який ми наслідуємо, визначають яким класом ми будемо репрезентувати записи з бази даних (User), а також якого типу в цього класу буде ідентифікатор (Integer).

Клас UserService:

```
@Service
public class UserService {
	@Autowired
	private UserRepository userRepository;
	public User update(User newUser, Integer id) {
		return save(newUser, id);
	}
	public User save(User newUser) {
		return save(newUser, newUser.getId());
	}
	private User save(User newUser, Integer id) {
		newUser.setId(id);
		return userRepository.save(newUser);
	}
	public User getUser(Integer id) {
		Optional<User> optionalUser = userRepository.findById(id);
		return optionalUser.orElse(null);
	}
	public void deleteUser(Integer id) {
		userRepository.deleteById(id); }}
```

Анотація @Service позначає, що клас UserService є серівс-компонентом програми, ініціалізація якого в інших класах буде реалізовуватись фреймворком. @Autowired над екземпляром інтерфейса UserRepository дозволяє віддати його вчасну ініціалізацію під контроль фреймворка. Назва кожного методу відповідає бізнес-задачі, яку він виконує. Методи save та update приймають як параметр об'єкт-користувача, якого необхідно зберегти або оновити. Методи getUser та deleteUser слугують для отримання та видалення користувача за його id, який передається як параметр у ці методи.

Клас UserController виглядає наступним чином:

```
@RestController
@RequestMapping(path = "/user")
public class UserController {

	@Autowired
	private UserService userService;

	@GetMapping(path = "/{id}")
	public User get(@PathVariable Integer id) {
		return UserTransformer.transform(userService.getUser(id));
	}

	@PostMapping
	public User add(@RequestBody User user) {
		return userService.save(user);
	}

	@PutMapping(path = "/{id}")
	public User update(@RequestBody User user, @PathVariable Integer id) {
		return userService.update(user, id);
	}

	@DeleteMapping(path = "/delete/{id}")
	public void delete(@PathVariable Integer id) {
		userService.deleteUser(id);
	}
}
```

Анотація @RestController дає знати фреймворку, що цей клас є контролером, який буде очікувати HTTP-запити та обробляти їх. @RequestMapping(path = "/user") визначає шлях URI до методів цього контролера (за допомогою значення, присвоєного аргументу path). Тобто, щоб виконати запит до одного з

методів цього контролера, необхідно до адреси додати префікс /user. Наприклад, при локальному підключенні URI до цього контролера буде наступним: localhost:8080/user . Анотації @GetMapping, @PostMapping, @PutMapping та @DeleteMapping над методами позначають, що саме вони будуть викликані при відповідних методах (GET, POST, PUT та DELETE), вказаних в HTTP-запиті. Аргумент path для цих анотації відіграє ту саму роль, що і для анотації @RequestMapping. @PathVariable перед аргументом, який передається методу, вказує на те, що ця змінна буде взята з запиту, який був надісланий (її розташування визначається в параметрі path анотації над методом). Наприклад, щоб виконати запит на отримання користувача, необхідно виконати HTTP запит з методом GET за URI localhost:8080/user/{id} (у випадку локального підключення), де на місці {id} варто вказати ідентифікатор користувача. @RequestBody вказує, що об'єкт, який передається в метод, повинен знаходитись та буде взятий з тіла HTTP-запита. За замовчуванням, формат передачі об'єкта через HTTP запит – JSON, а тому немає необхідності виконувати додаткові налаштування додатка. Назва кожного методу відповідає бізнес-задачі, яку він виконує.

Клас Main:

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

Анотація @SpringBootApplication разом з методом SpringApplication.*run* (Main.class, args); вказують, що програма буде виконуватись та ініціалізуватись повністю під контролем Spring фреймворка.

Файл з властивостями application.properties містить три наступні значення, які є необхідними для визначення та підключення до бази даних — url-адреса, логін та пароль:

```
        spring.datasource.url=jdbc:mysql://localhost:3306/RestDB?useLegacyDatetimeCode=false&serv
erTimezone=UTC
        spring.datasource.username=springuser
        spring.datasource.password=springPassword
```

Як результат, запустивши програму ми отримаємо робочий мікросервіс, який містить CRUD операції для таблиці в базі даних, яка зберігає інформацію про користувачів. Перевіримо її роботу. Спочатку додамо нового користувача за допомогою HTTP метода POST, передавши у тілі запита користувача у форматі JSON (запити виконуватимемо у програмі postman, рис. 3):

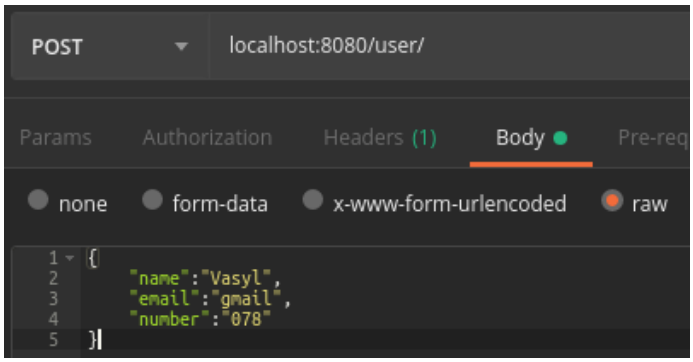


**Рис. 3. POST-запит до сервісу**

Як результат, мікросервіс записав переданий об'єкт в базу даних, та повернув нам результат запису з новим присвоєним користувачеві id (рис. 4):

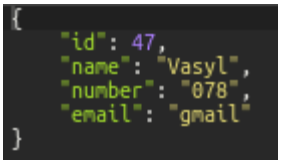


**Рис. 4. Результат POST запиту**

Потрібно підкреслити той момент, що до мікросервісу здійснюється звертання за URI localhost:8080/user/, оскільки саме такий шлях вказано в класі-контролері. Далі, за допомогою методу PUT можна змінити деякі значення користувача (рис. 5, результат оновлення буде повернений та показаний на рис. 6):

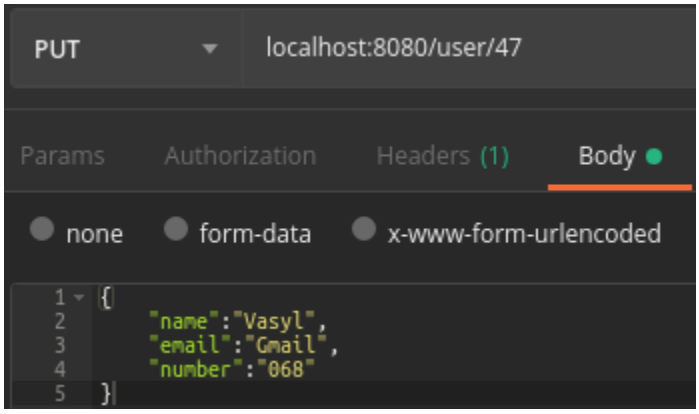


**Рис.5. PUT запит до сервісу**

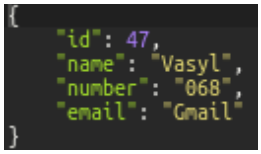


**Рис. 6. Результат PUT запиту**

Також ми можемо отримати щойно оновленого користувача за допомогою HTTP метода GET (рис. 7 та 8):

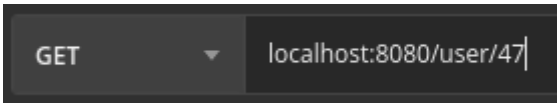


**Рис. 7. GET запит до сервісу**

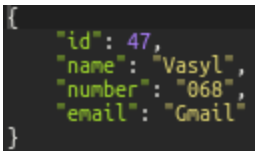


**Рис. 8. Результат GET запиту**

Також, ми можемо видалити користувача з бази даних, виконавши запит з HTTP методом DELETE (рис. 9, запит нічого не поверне, оскільки в контролері для метода DELETE ми вказали, щоб нічого не поверталось):

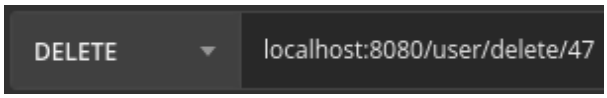


**Рис. 9. DELETE запит до сервісу**

Отже, створено готовий мікросервіс який можна застосовувати в інших проєктах з мікросервісною архітектурою.

## Висновки

Таким чином, можна зробити висновок про те, що мікросервісна архітектура та архітектурний стиль REST є доцільними методиками для створення гнучких, швидких та ефективних веб-додатків, які можна повністю розкрити за допомогою мови Java.

## Література

1. Chris Richardson: Microservices Patterns: With examples in Java, Manning Publications; 1 edition (November 19, 2018), ISBN: 9781617294549

2. Мартін Фаулер, Джеймс Льюїс Мікросервіси [Електронний ресурс]. – Режим доступу : https://martinfowler.com/articles/microservices.html

3. Стівен Уоттс, Лора Шифф Огляд архітектури моноліту проти мікросервісів [Електронний ресурс]. – Режим доступу : https://www.bmc.com/blogs/microservices-architecture/

4. Кая Ісмаїл 7 технічних гігантів, що охоплюють мікросервіси [Електронний ресурс]. – Режим доступу : https://www.cmswire.com/information-management/7-tech-giants-embracing-microservices/

5. Філ Кальчадо Будівництво продуктів на SoundCloud. Частина I: Справа з монолітом [Електронний ресурс]. – Режим доступу : https://developers.soundcloud.com/blog/building-products-at-

soundcloud-part-1-dealing-with-the-monolith

6. Рой філдінг Архітектурні стилі та дизайн мережевих архітектур програмного забезпечення [Електронний ресурс]. – Режим доступу : https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

7. Fielding, Roy T., Gettys, James, Mogul, Jeffrey C., Nielsen, Henrik Frystyk, Masinter, Larry, Leach, Paul J. Berners-Lee, Tim (June 1999). Hypertext Transfer Protocol – HTTP/1.1

**References**

1. Chris Richardson: Microservices Patterns: With examples in Java, Manning Publications; 1 edition (November 19, 2018), ISBN: 9781617294549

2. Martin Fowler, James Lewis Microservices. URL: https://martinfowler.com/articles/microservices.html

3. Stephen Watts, Laura Shiff An Overview of Monolithic vs Microservices Architecture. URL: https://www.bmc.com/blogs/microservices-architecture/

4. Kaya Ismail 7 Tech Giants Embracing Microservices. URL: https://www.cmswire.com/information-management/7-tech-giants-embracing-microservices/

5. Phil Calçado Building Products at SoundCloud. Part I: Dealing with the Monolith. URL: https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith

6. Roy Fielding. Architectural Styles and the Design of Network-based Software Architectures. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

7. Fielding, Roy T.; Gettys, James; Mogul, Jeffrey C.; Nielsen, Henrik Frystyk; Masinter, Larry; Leach, Paul J.; Berners-Lee, Tim (June 1999). Hypertext Transfer Protocol – HTTP/1.1

Надійшла / Paper received : 08.11.2020 р. Надрукована/Printed :27.11.2020 р.