**Державний торговельно-економічний університет**
**Кафедра інженерії програмногозабезпечення та кібербезпеки**

# ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА

**на тему:**

## «Мікросервісна архітектура програмного додатку організації освітньої діяльності кафедри»

Студента 2м курсу, 2 групи, спеціальності 121 «Інженерія програмного забезпечення» освітньої програми «Інженерія програмного забезпечення» _____ Ющенка Олександра Олександровича
підпис студента

Науковий керівник кандидат педагогічних наук, доцент кафедри інженерії програмного забезпечення та кібербезпеки _____ Жирова Тетяна Олександрівна
підпис керівника

Гарант освітньої програми кандидат педагогічних наук, доцент кафедри інженерії програмного забезпечення та кібербезпеки _____ Котенко Наталія Олексіївна
підпис гаранта

КИЇВ – 2023

Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення та кібербезпеки
Освітній ступінь магістр
Освітня програма 121 «Інженерія програмного забезпечення»

**Затверджую**
Зав. кафедри інженерії програмного забезпечення та кібербезпеки
Криворучко О. В.
«13» грудня 2022 р.

# Завдання
## на випускну кваліфікаційну роботу студентові

Ющенко Олександру Олександровичу
(прізвище, ім'я, по батькові)

1. Тема випускної кваліфікаційної роботи «Мікросервісна архітектура програмного додатку організації освітньої діяльності кафедри»
Затверджена наказом ректора від «15» лютня 2023 р. № 464
2. Строк здачі студентом закінченої роботи 27 листопада 2023
3. Цільова установка та вихідні дані до роботи
*Метою роботи є дослідження поняття та особливостей мікросервісної архітектури, її розвитку, переваг та недоліків, порівняння з іншими видами архітектури, вивчення області використання та практичне застосування мікросервісної архітектури при розробці програмного додатку.*
*Об'єктом дослідження є архітектура програмного забезпечення.*
*Предмет дослідження є мікросервісна архітектура, її особливості та застосування при розробці програмного додатку організації освітньої діяльності кафедри.*

4. Консультанти роботи із зазначенням розділів, які консультують:

| Розділ | Консультант (прізвище, ініціали) | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| | | | |
| | | | |

5. Зміст випускної кваліфікаційної роботи (перелік питань за кожним розділом)

ВСТУП

РОЗДІЛ 1 ПОНЯТТЯ, ПРИНЦИПИ ТА ОСОБЛИВОСТІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

1.1. Визначення архітектури програмного забезпечення

1.2. Передумови виникнення мікросервісної архітектури

1.3. Основні поняття мікросервісної архітектури

1.4. Принципи та переваги мікросервісної архітектури

1.5. Недоліки мікросервісної архітектури

1.6. Висновки до розділу 1

РОЗДІЛ 2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ДОДАТКУ ІЗ ВИКОРИСТАННЯМ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1. Опис основних функцій програмного додатку

2.2. Створення архітектурних діаграм

2.2.1. Діаграма прецедентів

2.2.2. Діаграма послідовності

2.2.3. Діаграма активності

2.3. Проєктування моделі додатку

2.3.1. Визначення основних сутностей

2.3.2. Концептуальна модель бази даних

2.3.3. Логічна модель бази даних

2.3.4. Фізична модель бази даних

2.4. Визначення предметних контекстів та сервісів додатку

2.5. Висновки до розділу 2

РОЗДІЛ 3 РОЗРОБКА ПРОГРАМНОГО ДОДАТКУ

3.1. Розробка back-end частини

6. Календарний план виконання роботи

| № пор. | Назва етапів випускної кваліфікаційної роботи | Строк виконання етапів роботи | |
|---|---|---|---|
| | | за планом | фактично |
| 1 | 2 | 3 | 4 |
| 1. | *Вибір теми випускної кваліфікаційної роботи* | *07.11.2022* | *07.11.2022* |
| 2. | *Розробка та затвердження завдання на роботу магістра (стац/заоч)* | *13.12.2022* | *13.12.2022* |
| 3. | *Вступ та перелік літературних джерел* | *24.02.2023* | *24.02.2023* |
| 4. | *Розробка технічного завдання* | *15.03.2023* | *15.03.2023* |
| 5. | *Розділ 1. ПОНЯТТЯ, ПРИНЦИПИ ТА ОСОБЛИВОСТІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ* | *10.04.2023* | *10.04.2023* |
| 6. | *Розділ 2. ПРОЄКТУВАННЯ ПРОГРАМНОГО ДОДАТКУ ІЗ ВИКОРИСТАННЯМ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ* | *24.05.2023* | *24.05.2023* |
| 7. | *Розділ 3. РОЗРОБКА ПРОГРАМНОГО ДОДАТКУ ІЗ ВИКОРИСТАННЯМ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ* | *06.09.2023* | *06.09.2023* |
| 8. | *Розробка програми та методики тестування* | *18.10.2023* | *18.10.2023* |
| 9. | *Написання наукової статті* | *17.05.2023* | *17.05.2023* |
| 10. | *Керівництво користувача* | *25.10.2023* | *25.10.2023* |
| 11. | *Висновки та пропозиції* | *01.11.2023* | *01.11.2023* |
| 12. | *Здача випускної кваліфікаційної роботи на кафедру (перша перевірка)* | *06.11.2023* | *06.11.2023* |
| 13. | *Підготовка автореферату та презентації доповіді* | *06.11.2023* | *06.11.2023* |
| 14. | *Попередній захист випускної кваліфікаційної роботи* | *20.11.2023 – 24.11.2023* | *20.11.2023 – 24.11.2023* |
| 15. | *Здача зброшурованої випускної кваліфікаційної роботи* | *27.11.2023* | *27.11.2023* |
| 16. | *Зовнішнє рецензування випускної кваліфікаційної роботи* | *29.11.2023* | *29.11.2023* |
| 17. | *Підготовка до публічного захисту випускної кваліфікаційної роботи* | *05.12.2023-06.12.2023* | *05.12.2023-06.12.2023* |

7. Дата видачі завдання «13» грудня 2022 р.

8. Науковий керівник випускної кваліфікаційної роботи Жирова Т.О.
(прізвище, ініціали, підпис)

9. Гарант освітньої програми Котенко Н.О.
(прізвище, ініціали, підпис)

10. Завдання прийняв до виконання студент Ющенко О.О.
(прізвище, ініціали, підпис)

11. Відгук керівника випускного кваліфікаційного проєкту

_____

Науковий керівник випускної кваліфікаційної роботи

_____

*(підпис, дата)*

Відмітка про попередній захист _____

*(ПІБ, підпис, дата)*

12. Висновок про випускну кваліфікаційну роботу

Випускна кваліфікаційна робота студента _____Ющенко О.О._____

*(прізвище, ініціали)*

може бути допущена до захисту екзаменаційній комісії.

Гарант освітньої програми _____Котенко Н.О._____

*(прізвище, ініціали, підпис)*

Завідувач кафедри _____Криворучко О. В._____

*(підпис, прізвище, ініціали)*

«_____»_____20 ___ р.

# АНОТАЦІЯ

Дана робота присвячена розробці додатку організації освітньої діяльності кафедри із використанням мікросервісної архітектури програмного забезпечення. Основною метою роботи є аналіз поняття мікросервісної архітектури та застосування отриманих знань на навичок на практиці. У ході написання роботи було визначено поняття, основні принципи, переваги та недоліки мікросервісної архітектури, її застосування в ході розробки програмних додатків, були проведені порівняння з іншими видами архітектури, спроєктовано модель бази даних та розроблено практичний приклад додатку із використанням мікросервісної архітектури. Для реалізації додатку були використані сучасні технології програмування та розгортання, був проведений аналіз вимог для прийняття відповідних рішень щодо проєктування додатку. Результуюча система може використовуватись кафедрою для проведення наукових конференцій, а саме для збору матеріалів та статей, виставленню оцінок, коментарів, комунікації з учасниками, пошуку статей, що задовольняють вимогам конференції.

Випускна кваліфікаційна робота на тему «Мікросервісна архітектура програмного додатку організації освітньої діяльності кафедри» містить 53 сторінок, 16 рисунків та 1 таблицю. Перелік використаних джерел налічує 12 найменувань.

**Ключові слова**: архітектура програмного забезпечення, мікросервісна архітектура, предметно-орієнтоване проєктування.

## ABSTRACT

This thesis is devoted to the development of an application for the organization of the department's educational activities with the usage of microservice architecture. The main goal of this paper is analysis of the concept of microservice architecture and practical usage of gained knowledge during the development of the application. This thesis considers the concept of microservice architecture, its principles, advantages, disadvantages, usage during development, comparison to other software architecture types. For the application implementation, modern development and deployment technologies were utilized; technical requirements were thoroughly analyzed to make correct decisions for the design and planning of the application, database model was designed and implemented, application was developed with microservice architecture concepts in mind. The department can use the resulting system to hold conferences, gather and review submissions and papers, communicate, and leave comments for the participants, and accept or reject the submissions to leave only suitable for the conference.

The final qualification work on the topic "Microservice architecture of the software application for the organization of educational activities of the department" contains 53 pages, 16 figures and 1 table. The list of references includes 12 titles.



**Keywords**: software architecture, microservice architecture, domain-driven design.

# СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧКИ

ПЗ – програмне забезпечення.

UML – Unified Modeling Language. Уніфікована мова моделювання, що використовується для проєктування, візуалізації та документування програмної системи, взаємодії її компонентів тощо.

БД – база даних.

СКБД – система керування базами даних.

API – Application Programming Interface. Інструмент для взаємодії частин програми або різних програми одна з одною.

DDD – Domain-driven design. Предметно-орієнтоване проєктування.

CLR – Common Language Runtime. Загальномовна середа виконання.

REST – Representational State Transfer.

HTTP – Hyper Text Transact Protocol.

CRUD – Create Read Update Delete.

CQRS – Command-Query Responsibility Segregation.

JSON – JavaScript Object Notation.

JWT – JSON Web Token.

SPA – Single Page Application.

CORS – Cross-Origin Resource Sharing.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | *ДТЕУ 121 02-26.МР* | | | |
| *Зм.* | *Аркуш* | *№ докум.* | *Підпис* | *Дата* | | | | |
| Зав. каф. | | Криворучко О.В. | | 19.09.23 | *Мікросервісна архітектура програмного додатку організації освітньої діяльності кафедри* | *Стадія* | *Аркуш* | *Аркушів* |
| Керівник | | Жирова Т.О. | | 19.09.23 | | *ПС* | *2* | *53* |
| Гарант | | Котенко Н.О. | | 19.09.23 | | *Факультет інформаційних технологій 2м курс, 2 група* | | |
| Розробив | | Ющенко О.О. | | 19.09.23 | *Перелік умовних скорочень* | | | |

## ЗМІСТ



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | *ДТЕУ 121 02-26.МР* | | | |
| *Зм.* | *Аркуш* | *№ докум.* | *Підпис* | *Дата* | | | | |
| Зав. каф. | | Криворучко О.В. | | 19.09.23 | *Мікросервісна архітектура програмного додатку організації освітньої діяльності кафедри* | *Стадія* | *Аркуш* | *Аркушів* |
| Керівник | | Жирова Т.О. | | 19.09.23 | | *Зміст* | *3* | *53* |
| Гарант | | Котенко Н.О. | | 19.09.23 | | *Факультет інформаційних технологій 2м курс, 2 група* | | |
| Розробив | | Ющенко О.О. | | 19.09.23 | *Зміст* | | | |

# ВСТУП

Інформаційні технології та програмне забезпечення відіграють дуже важливу роль у житті суспільства. Банківська справа, торгівля, медицина, освіта та безліч інших сфер діяльності людини – все це спирається на спеціалізоване програмне забезпечення, що призначене для полегшення та підвищення ефективності праці. Умовами правильного та безперебійного функціонування такого програмного забезпечення є якість та «чистота» програмного коду, що в першу чергу залежить від проаналізованих вимог та правильно підібраної архітектури додатку. Саме тому дана тема є **актуальною** на сьогодні.

**Метою роботи** є дослідження поняття та особливостей мікросервісної архітектури, її розвитку, переваг та недоліків, порівняння з іншими видами архітектури, вивчення області використання та практичне застосування мікросервісної архітектури при розробці програмного додатку.

**Об'єктом дослідження** є архітектура програмного забезпечення.

**Предмет дослідження** – мікросервісна архітектура, її особливості та застосування при розробці програмного додатку організації освітньої діяльності кафедри.

**Ключові слова**: архітектура програмного забезпечення, мікросервісна архітектура.

Випускна кваліфікаційна робота складається з анотації, розділу скорочень та умовних позначень, вступу, трьох основних розділів, висновків, списку використаних джерел та додатків.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | *ДТЕУ 121 02-26.МР* | | | |
| *Зм.* | *Аркуш* | *№ докум.* | *Підпис* | *Дата* | | | | |
| Зав. каф. | | Криворучко О.В. | | 24.02.23 | *Мікросервісна архітектура програмного додатку організації освітньої діяльності кафедри* | *Стадія* | *Аркуш* | *Аркушів* |
| Керівник | | Жирова Т.О. | | 24.02.23 | | *В* | *5* | *53* |
| Гарант | | Котенко Н.О. | | 24.02.23 | | *Факультет інформаційних технологій 2м курс, 2 група* | | |
| Розробив | | Ющенко О.О. | | 24.02.23 | *Вступ* | | | |

# РОЗДІЛ 1
# ПОНЯТТЯ, ПРИНЦИПИ ТА ОСОБЛИВОСТІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

## 1.1. Визначення архітектури програмного забезпечення

Архітектура програмного забезпечення – це набір правил та вимог до внутрішньої структури програми та того, як її компоненти та модулі взаємодіють між собою. Основною задачею архітектури програмного забезпечення є аналіз вимог до системи та вироблення стратегії, що позволить правильно відобразити та реалізувати предметну область задачі через код. Розроблена архітектура документується (у тому числі за допомогою UML діаграм) та використовується розробниками як «шаблон» при розробці програми.

Правильно реалізована архітектура ПЗ допомагає вирішувати проблеми, що описуються її характеристиками, а саме [1]:

- доступність
- надійність
- складність
- придатність до тестування
- розширюваність
- безпека
- гнучкість
- відмовостійкість
- здатність до відновлення
- здатність до розгортання
- продуктивність

Проєктування архітектури ПЗ – це один із перших та найважливіших етапів розробки ПЗ. Помилки допущені на етапі проєктування дуже важко, а іноді і неможливо виправити після їх реалізації у програмному продукті.

## 1.2. Передумови виникнення мікросервісної архітектури

Розвиток архітектури програмного забезпечення тісно пов'язаний із розвитком інформаційних технологій у цілому. Перше спеціалізоване програмне забезпечення не було дуже вибагливим та складним, проте разом із розвитком та ускладненням задач, що потрібно було вирішувати, почали зазнавати розвитку і мови програмування, парадигми та шаблони проєктування.

Першим популярним видом архітектури, що дав передумови та причини до виникнення мікросервісної архітектури, прийнято вважити монолітну (monolithic) архітектуру. Головною особливістю цього виду архітектури було те, що весь додаток побудований за даним принципом був неподільним та самодостатнім. Монолітний додаток містив у собі абсолютно всю логіку, що потрібна для його роботи, він не був залежним від інших додатків та розповсюджувався у вигляді однієї програми.

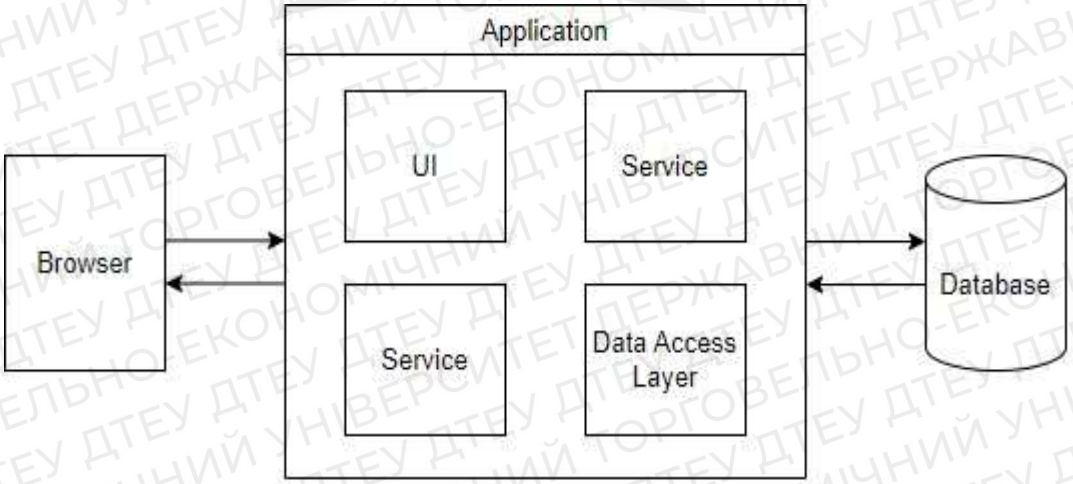


*Рис. 1.1. Приклад монолітної архітектури*

Такий підхід містить набагато більше недоліків, ніж переваг. Головною перевагою даної архітектури є легкість її реалізації. Монолітний додаток – це те, що зазвичай виходить неявно, природньо, оскільки для реалізації такої архітектури не потрібно витрачати багато зусиль – додаток розростається, новий код просто додається до існуючого, функціонал та область використання змішуються.

Вартість впровадження та легкість розгортання теж є сильними сторонами монолітної архітектури, оскільки для запуску однієї програми на сервері не потрібна велика кількість налаштувань та інфраструктури. Вартість інфраструктури що потрібна для розгортання такого додатку буде мінімальною у порівнянні з іншими видами архітектур. Наявність лише одного виконуваного файлу також полегшує процеси тестування та налагодження (debug).

Головним недоліком монолітної архітектури є її непідтримуваність протягом довгого часу. Оскільки монолітний додаток є неподільним, це означає що він розростається дуже швидко разом із впровадженням нового функціоналу. Вихідний код додатку стає незрозумілим та заплутаним, впровадження нових ідей стає тільки важче, все частіше стає у нагоді «рефакторинг», що тільки збільшує вартість підтримки та розробки нового програмного коду.

Іншим важливим недоліком монолітної архітектури є її ненадійність та немасштабованість. Через неподільність монолітного додатку, неможливо масштабувати тільки ту його частину, яка зазнає найбільшого навантаження, можливо масштабувати лише весь додаток уцілому, шляхом запуску додаткових екземплярів додатку (горизонтальне масштабування). З цієї самої причини, необроблена помилка під час виконання монолітної програми ставить під загрозу весь додаток одразу - одна така помилка може «покласти» весь додаток.

Таким чином, можна виділити наступні **передумови виникнення** мікросервісної архітектури:

- *Ускладнення вимог та постійний розвиток*. Додавати новий функціонал потрібно все частіше, це повинно бути легко та відносно недорого. Програмний код повинен бути підтримуваним, щоб зменшити вартість обслуговування додатку протягом великого проміжку часу.
- *Доступність*. Із постійним розвитком інформаційних технологій все більше людей мають доступ до комп'ютерів, смартфонів та інтернету, отже системи повинні витримувати більше навантаження.
- *Відмовостійкість*. Чим більше система, тим складніше стають зв'язки між її частинами, збільшується можливість помилок. Система не повинна припиняти свою роботу, навіть у випадку коли якась її частина відмовила.
- *Заснування таких практик як Agile та DevOps*. Із появою таких нових практик як Agile та DevOps, головними ідеями яких є «гнучкість» команди, швидкість впровадження нових вимог та малий час виходу на ринок, недоліки монолітної архітектури почали ставати все біль помітними.

### 1.3. Основні поняття мікросервісної архітектури

**Мікросервісна архітектура** – це підхід до розробки програмного забезпечення як сукупності окремих незалежних сервісів, що взаємодіють між собою. Кожний такий сервіс відповідає за конкретну функціональність додатку, взаємодіє з іншими сервісами за допомогою API (Application Programming Interface), що вони надають, та може бути розгорнутий та масштабований окремо від інших. Сервіс – це структурна одиниця мікросервісної архітектури, саме це є її головною особливістю.

Оскільки мікросервісна архітектура – це сукупність окремих сервісів, що її складають, рівень незалежності таких сервісів можна оцінити за допомогою двох характеристик: зв'язності та пов'язаності. Відповідно до Ларрі Константина, засновника даних термінів, система вважається стабільною, коли пов'язаність її модулів висока, а зв'язність низька («A structure is stable if cohesion is high, and coupling is low»).

**Зв'язність** (coupling) – це міра того, наскільки модулі системи залежать один від одного. Якщо зв'язність між двома частинами висока, це означає, що зміна одного модуля скоріш за все спричинить зміну іншого, і навпаки, низька зв'язність програмних модулів позволяє змінювати одну частину програми майже незалежно від іншої. У контексті мікросервісної архітектури, такими модулями або частинами є самі сервіси, чим менш сервіси є залежними від внутрішньої реалізації один одного, тим меншою є їх зв'язність.

Існує багато типів зв'язності, що залежать від їх чинників та наслідків:

- Зв'язність реалізації (implementation coupling)
- Темпоральна зв'язність (temporal coupling, тимчасова або зв'язність у часі)
- Зв'язність розгортання (deployment coupling)
- Доменна зв'язність (domain coupling)

Проте найбільше на мікросервісну архітектуру впливає саме предметна зв'язність.

**Доменна (предметна) зв'язність** – логічна зв'язність, що виникає при взаємодії мікросервісів уцілому. Предметної зв'язності неможливо уникнути, проте її наявність можливо мінімізувати. Проблему предметної зв'язності вивчає предметно-орієнтоване проєктування (domain-driven design).

**Предметно-орієнтоване проєктування** - це підхід до розробки програмного забезпечення, який зосереджується навколо предметної області, гарантуючи, що результуюче програмне забезпечення точно моделює реальний проблемний простір, його процеси та моделі. Предметно-орієнтоване проєктування робить сильний акцент на співпраці між експертами предметної області (як правило, нетехнічними зацікавленими сторонами) і командами розробників. Завдяки спільній мові та спільному розумінні предметної області, DDD має на меті створення програмного забезпечення, яке тісно пов'язане з цілями та вимогами бізнесу.

Головні поняття предметно-орієнтованого проєктування [5]:

1. **Єдина мова** (Ubiquitous language). DDD заохочує команду розробників та експертів предметної області використовувати спільну мову для опису концепцій та процесів в межах предметної області. Ця мова повинна послідовно використовуватися в коді (назви сервісів, класів, методів, змінних тощо), обговореннях і документації, долаючи розрив між бізнес і технічними сторонами.
2. **Обмежені контексти** (Bounded context). DDD ділить всю велику предметну область бізнесу на менші частини - обмежені контексти. Кожний обмежений контекст описує окрему область предметної області та має власне призначення, цілі, модель тощо. Таке розподілення позволяє розбити велику предметну область на більш малі задачі та розв’язувати їх поступово.
3. **Агрегат, сутність та об’єкт-значення** (Aggregate, entity, value-object) – це будівельні блоки, що використовуються при проєктуванні моделі предметної області.

   Агрегат – це набір споріднених даних, що оброблюються та сприймаються як єдиний об’єкт. Агрегат забезпечує цілісність даних, оскільки оновлюється або увесь агрегат, або нічого, а також

приховує логіку, необхідну для його обробки. Класичним прикладом агрегату є замовлення в інтернет магазині. Замовлення складається із набору товарів, інформації про доставку, оплату та посилання на покупця.

Сутність – це об’єкт, що описується його властивостями та відношеннями з іншими об’єктами. Сутність завжди можна унікально ідентифікувати та змінювати протягом часу (оновлювати її властивості). Прикладом сутності є обліковий запис користувача.

Об’єкт-значення – це об’єкт, що не має індивідуальності, не змінюється протягом часу та, на відміну від сутності, може бути замінений іншим об’єктом-значенням, за умови що вони мають однакові властивості. Гарним прикладом взаємодії сутності та об’єкта значення є малюнок олівцями. При створенні малюнка, заміна олівців на вплине на кінцевий результат, за умови що вони мають однакові властивості (відтінок, жорсткість тощо), проте кінцеві малюнки, навіть створені за допомогою одних й тих же самих олівців, завжди будуть відрізнятися та мати свої унікальні риси [5].

**Пов’язаність** – це міра того, наскільки функціонально пов’язаний код є згрупованим між собою. Висока пов’язаність означає, що увесь код відповідальний за певний функціонал знаходиться разом, в споріднених модулях або сервісах. Внаслідок високої пов’язаності легше вносити зміни в існуючий функціонал, оскільки набагато легше знайти код, на який ці зміни вплинуть. Пов’язаність та зв’язність є протилежно залежними один від одного, ідеальна система повинна прагнути до мінімальної зв’язності та максимальної пов’язаності [6].

Виділяють наступні типи пов’язаності (від найслабшої до найсильнішої):

- Випадкова (coincidental).
- Логічна (logical).
- Тимчасова (temporal).
- Комунікації (communicational).
- Sequential (послідовна).
- Functional (функціональна).

### 1.4. Принципи та переваги мікросервісної архітектури

*Сервіси є малими та сфокусованими на конкретній задачі*. У кожного сервісу у системі є своє конкретне призначення, що відображає окремий аспект предметної області додатку. Сервіс повинен містити у собі лише той функціонал та логіку, які безпосередньо стосується тієї задачі, що він призначений вирішувати.

За допомогою ділення додатку на множину сервісів можна уникнути заплутаності та змішування програмного коду, адже відтепер вихідний код буде логічно розділений у залежності від бізнес-вимог. Додавання нового функціоналу також стає легше, оскільки зміні зазнається не увесь додаток одразу, а лише його мала частина.

*Автономність та ізольованість*. Кожний сервіс є окремою незалежною сутністю, що запускається як окремий процес операційної системи або додаток у хмарі. Сервіси не знають про внутрішню реалізацію один одного та взаємодіють тільки за допомого API (Application Programming Interface), що вони надають. Зміни, що відбуваються в одному сервісі, не повинні вплинути на роботу інших сервісів та клієнтів, за умови що існуючий програмний інтерфейс сервісу залишився без змін. Така «розв’язаність» (decoupling) сприяє зменшенню залежностей частин додатку

одна від одної, надає можливість змінювати реалізацію сервісів без впливу на всю систему та краще перевикористовувати існуючий функціонал.

Іншою вагомою перевагою є зручність розгортання сервісів. Оновлення декількох рядків коду у монолітному додатку вимагає повного перерозгортання всієї системи, що завжди є повільним, сповненим ризику, процесом. Це також означає, що випуск релізів та оновлень стається рідше, адже малі зміни вигідніше групувати та випускати разом. Внесення та випуск змін у мікросервіси є набагато легшим та безпечнішим процесом, оскільки зміні зазнається лише один конкретний сервіс.

*Організованість навколо бізнес-вимог.* Відповідно до закону Конвея (Melvin Conway, 1968), «будь-яка організація що проєктує систему, створить дизайн зі структурою, яка буде копією структури комунікації в цій організації». Це означає, що структура великих систем часто залежить від структури та організації зв'язків у середині самої компанії, що буде використовувати цю систему. Якщо у компанії є суворе розподілення на команди спеціалістів із front-end, back-end, спеціалістів із баз даних тощо, то кінцева система скоріш за все буде мати точно таке саме розподілення по модулям. Проблема закладається у тому, що при впроваджені нових ідей та функціоналу може виникнути питання, який саме підрозділ має впроваджувати ці зміни. Через суворе розподілення спеціалістів на команди, ці зміни часто погано обговорюються або впроваджуються не в ту частину системи, куди вони належать, що у свою чергу призводить до змішування та ускладення програмного коду.

Мікросервісна архітектура допомагає вирішити цю проблему за допомогою організованості навколо вимог, не навколо ролей. Команда, що працює над конкретним сервісом, може мати спеціалістів з різних областей, що у свою чергу допомагає правильно оцінити та реалізувати зміни. Також, така команда не отримає завдання на впровадження змін, що її не

стосуються, адже за такою структурою можна легко зрозуміти, яка команда відповідальна за конкретну частину додатку.

*Незалежність від технологій*. На відміну від монолітної архітектури, де обраний стек технологій майже не змінюється протягом всього часу, мікросервісна архітектура цілковито заохочує використання різних технологій для різних сервісів. Кожний сервіс у праві користуватися саме такими інструментами, які найкраще підходять для реалізації поставленої задачі.

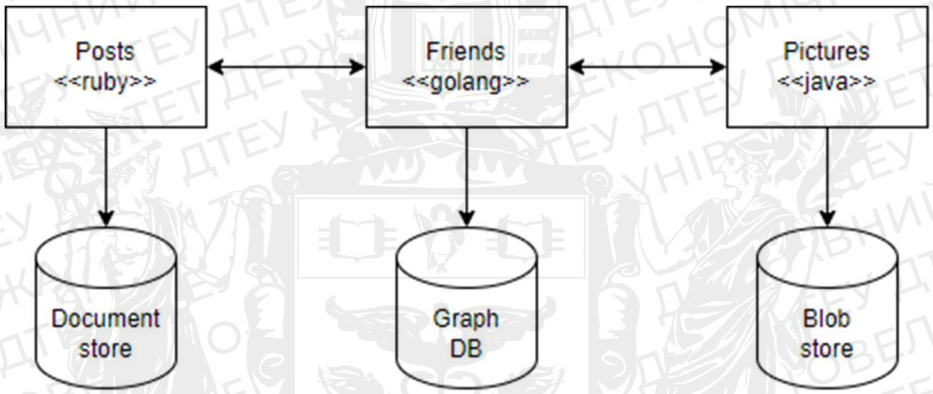


*Рис. 1.2. Приклад сервісів, створених із використанням різних технологій*

*Пристосованість до збоїв*. Впровадження мікросервісної архітектури дозволяє значним чином підвищити відмовостійкість додатку, адже при виході із ладу одного із сервісів, додаток не завершується аварійно, а лише втрачає частину відповідного функціоналу. Альтернативно, окремі сервіси можна вимикати усвідомлено, наприклад у випадку витоку даних або ключів доступу.

*Автоматизація інфраструктури*. Мікросервісну архітектуру дуже зручно використовувати разом із такими практиками як DevOps та Agile, головними принципами яких є гнучкість, швидкість розробки, розгортання та випуску продукту на ринок. Для кожного сервісу додатку зручно створювати процеси автоматизації (CI/CD pipeline), що є відповідальними за збирання коду, виконання модульних тестів, забезпечення необхідної

інфраструктури та розгортання. Наявність таких процесів значно підвищує ефективність розробки та зменшує час, який розробники витрачають на непов'язані із розробкою активності. Процес випуску нових версій продукту також стає значно простішим та менш ризикованим, оскільки можливість людської помилки стає менше: все що необхідно зробити – це запустити необхідний процес.

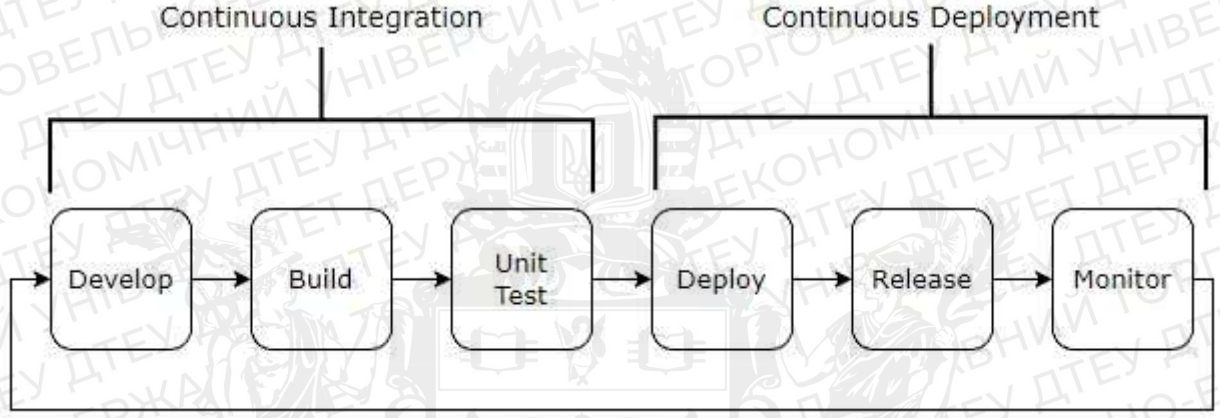


*Рис. 1.3. Приклад автоматизації процесу збирання та розгортання коду*

## 1.5. Недоліки мікросервісної архітектури

*Складність освоєння.* Новим членам команди завжди потребується час на ознайомлення з існуючим програмним додатком, документацією, вихідним кодом тощо. У випадку із мікросервісним додатком, час на таке освоєння та складність самого процесу значно підвищується.

*Складність тестування.* З одної сторони, мікросервіси полегшують тестування, оскільки перевірці зазнається окремий сервіс за раз; тестування окремого, конкретного функціоналу стає легше, оскільки за нього відповідає один сервіс. З іншої сторони, у випадку коли необхідно протестувати велику частину функціоналу (наприклад інтеграційного або e2e тестування), що може потребувати взаємодії великої кількості сервісів, важкість тестування значно зростає. У випадку, коли такий сценарій не проходить тестування, може бути абсолютно неочевидно, який саме із

сервісів викликав помилку, тому кожний з них зазнає повторного тестування. Тестування взаємодії сервісів один з одним також не є простою задачею, адже в залежності від виду комунікації між сервісами, команді тестувальників можуть знадобитися різні навички та інструменти.

*Вартість інфраструктури*. Через те, що кожний сервіс хоститься окремо від інших, кількість необхідної інфраструктури, а отже її кінцева вартість зростає. Різні сервіси можуть потребувати різні операційні системи, різні бази даних, різні обчислювальні потужності, різна засоби комунікації тощо. Якщо додаток розрахований на велику кількість користувачів, окремої уваги потребує налаштування мережі та розподілення трафіку, що теж потребує додаткових ресурсів та витрат. Також, через те, що кожний сервіс користується власним набором технологій, кількість різноманітних ліцензій зростає.

*Вибагливість до експертизи.* Через велику кількість технологій, що можуть використовуватись під час розробки додатку, що використовує мікросервісну архітектуру, нерідко одному розробнику потрібно мати навички із декількох технологій або інструментів. DevOps інженери, що відповідають за розгортання інфраструктури, налаштування мережі, автоматизацію процесів, також мають володіти широким спектром навичок, щоб можна було виконати всі поставлені вимоги. Все це тільки підвищує складність пошуку необхідних спеціалістів та витрати на них

## 1.6. Висновки до розділу 1

У даному розділі було дано визначенню поняттю мікросервісної архітектури, проаналізовано передумови її виникнення, було розглянуто такі її характеристики як зв’язність та пов’язаність, їх класифікація та способи боротьби із ними, зазначено переваги та недоліки даного виду архітектури.

Через свої переваги, мікросервісна архітектура отримала визнання розробників та широке поширення. Першою великою компанією, що почала активно застосовувати та сприяти розвитку мікросервісної архітектури прийнято вважати Netflix. Netflix почали міграцію від монолітної архітектури у 2009 році, через швидке зростання кількості активних користувачів та інформації, що потрібно зберігати та оброблювати, і на кінець 2011 року перенесли весь свій функціонал на мікросервіси у хмарі. Безліч інших великих компаній, таких як Amazon, Microsoft, Uber, Spotify, Twitter, також використовують мікросервіси у своїх додатках.

Сьогодні мікросервісну архітектуру можна використовувати при розробці широкого спектру застосунків, особливо якщо вони мають підтримувати високе навантаження або оброблювати велику кількість інформації у режимі реального часу. Інтернет магазини, соціальні мережі, фінансові додатки, IoT та багато іншого – все це може ефективно використовувати переваги мікросервісної архітектури.

# РОЗДІЛ 2
# ПРОЄКТУВАННЯ ПРОГРАМНОГО ДОДАТКУ ІЗ ВИКОРИСТАННЯМ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

## 2.1. Опис основних функцій програмного додатку

Головною метою роботи є створення порталу для кафедри інженерії програмного забезпечення та кібербезпеки, що позволить проводити наукові конференції, а саме допоможе у процесах збору та рецензування наукових статей, що плануються до представлення та захисту. Додаток повинен надавати можливість реєструватися новим користувачам, створювати нові конференції, додавати учасників до конференцій, надавати їм відповідні ролі та в залежності від отриманої ролі, публікувати чи рецензувати опубліковані статті.

Проаналізувавши основне призначення додатку, можна виділити 4 категорії користувачів:

- Адміністратор (admin)
- Голова конференції (chair)
- Автор (author)
- Рецензент (reviewer)

Зауважимо, що оскільки один і той самий користувач може мати різні ролі у різних конференціях, тільки роль адміністратора є глобальною по всьому додатку, інші ролі мають бути чинними тільки у рамках конкретної конференції.

Опишемо основний функціонал додатку для кожної ролі за допомогою історій користувача:

Як **адміністратор**, я хочу:

- Мати можливість створювати нові конференції, оновляти інформацію про існуючі
- Мати доступ до переліку всіх конференцій
- Додавати або видаляти користувачів у конференції
- Змінювати ролі користувачів в межах конференції

Як **голова конференції**, я хочу:

- Бачити перелік моїх конференцій, мати можливість їх налаштовувати
- Додавати або видаляти нових користувачів до моїх конференцій
- Змінювати ролі користувачів у межах моїх конференцій
- Бачити перелік публікацій конференції
- Бачити детальну інформацію про публікацію, список завантажених файлів, рецензії, коментарі тощо
- Змінювати рецензентів публікацій
- Приймати або відхиляти публікацію після збору всіх рецензій

Як **рецензент**, я хочу:

- Бачити перелік моїх конференцій
- Мати доступ до переліку статей із загальною інформацією про них
- Мати можливість відмітити публікації у яких я зацікавлений
- Мати можливість залишати рецензії та коментарі до статей, у яких я є рецензентом

Як **автор**, я хочу:

- Бачити перелік моїх конференцій
- Мати можливість створювати та оновлювати свої публікації
- Бачити перелік своїх публікацій
- Переглядати свої публікації, рецензії та коментарі до них

## 2.2. Створення архітектурних діаграм

### 2.2.1. Діаграма прецедентів

Діаграма прецедентів (або діаграма варіантів використань) показує варіанти взаємодії користувача із системою та іншими користувачами. Спираючись на зазначений функціонал програмного додатку, можна побудувати наступну діаграму прецедентів (рис. 2.1.):

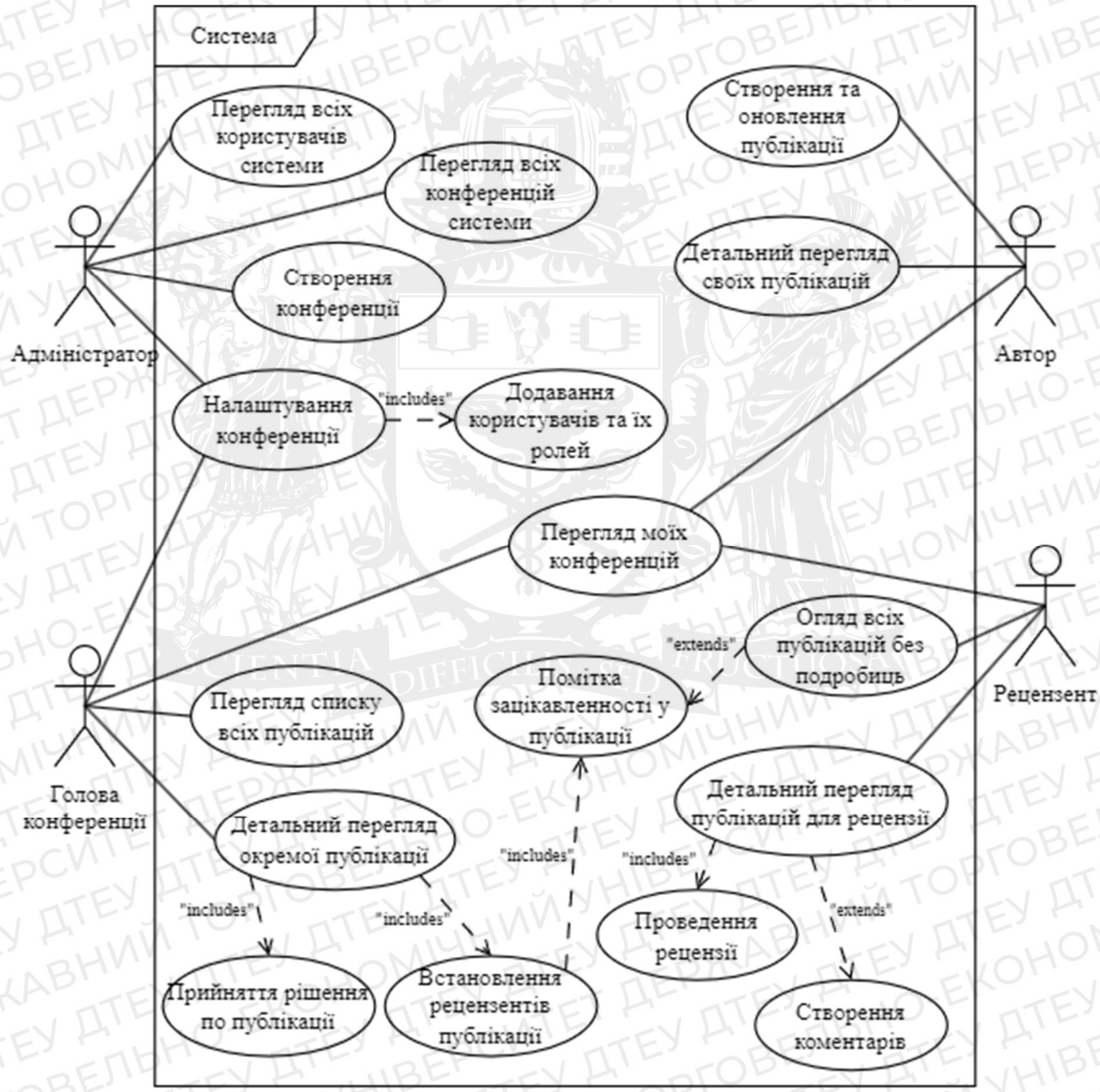


*Рис. 2.1. Діаграма прецедентів*

### 2.2.2. Діаграма послідовності

Діаграма послідовності вказує на те, як елементи та користувачі системи взаємодіють між собою із урахуванням плину часу. Типовий процес опрацювання однієї публікації на діаграмі послідовності виглядатиме наступним чином:

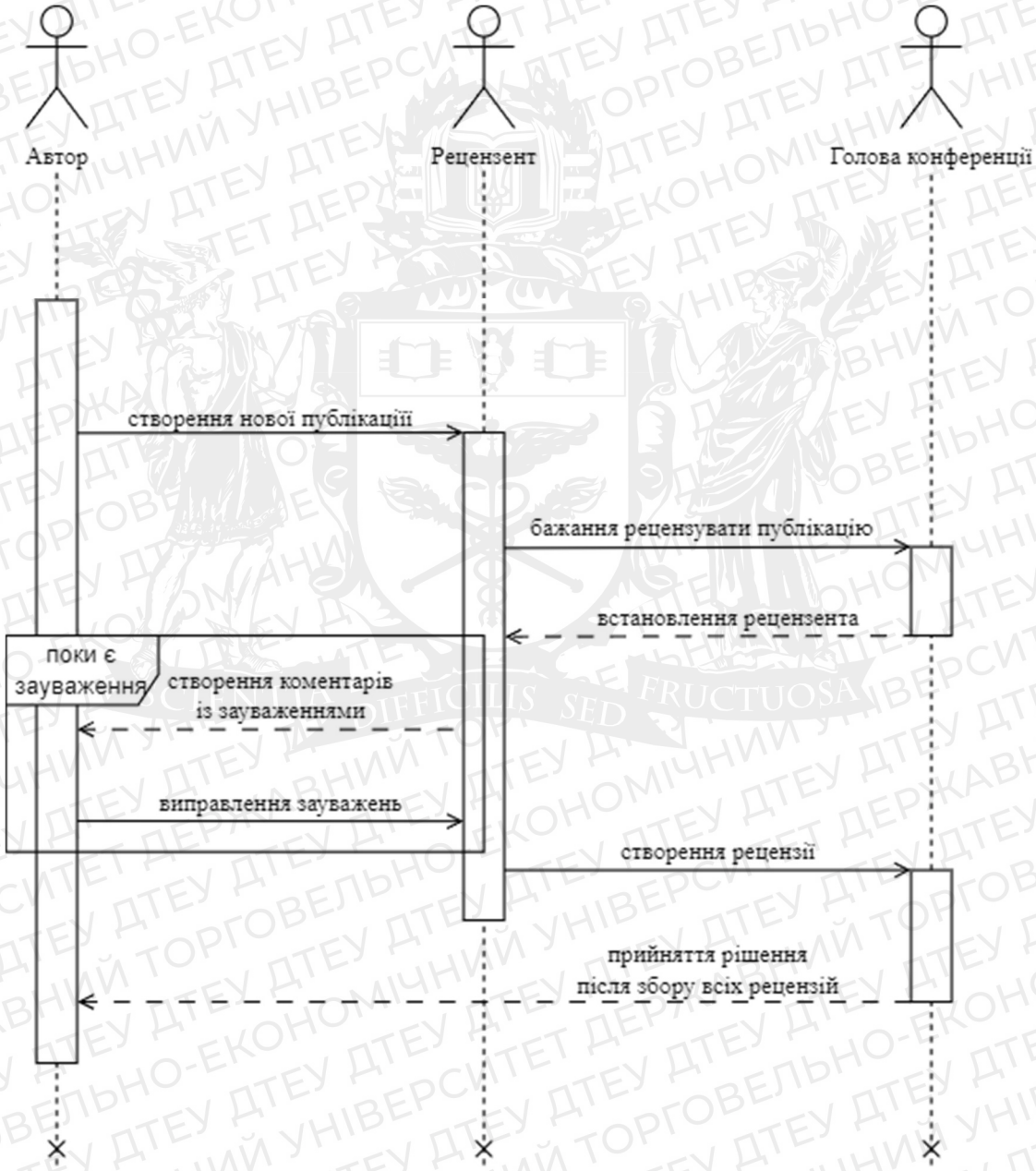


*Рис. 2.2. Діаграма послідовності опрацювання публікації*

### 2.2.3. Діаграма активності

Діаграма активності – це графічне зображення робочого процесу (workflow), що складається з покрокових дій, з підтримкою логічних розгалужень, паралельних та циклічних операцій.

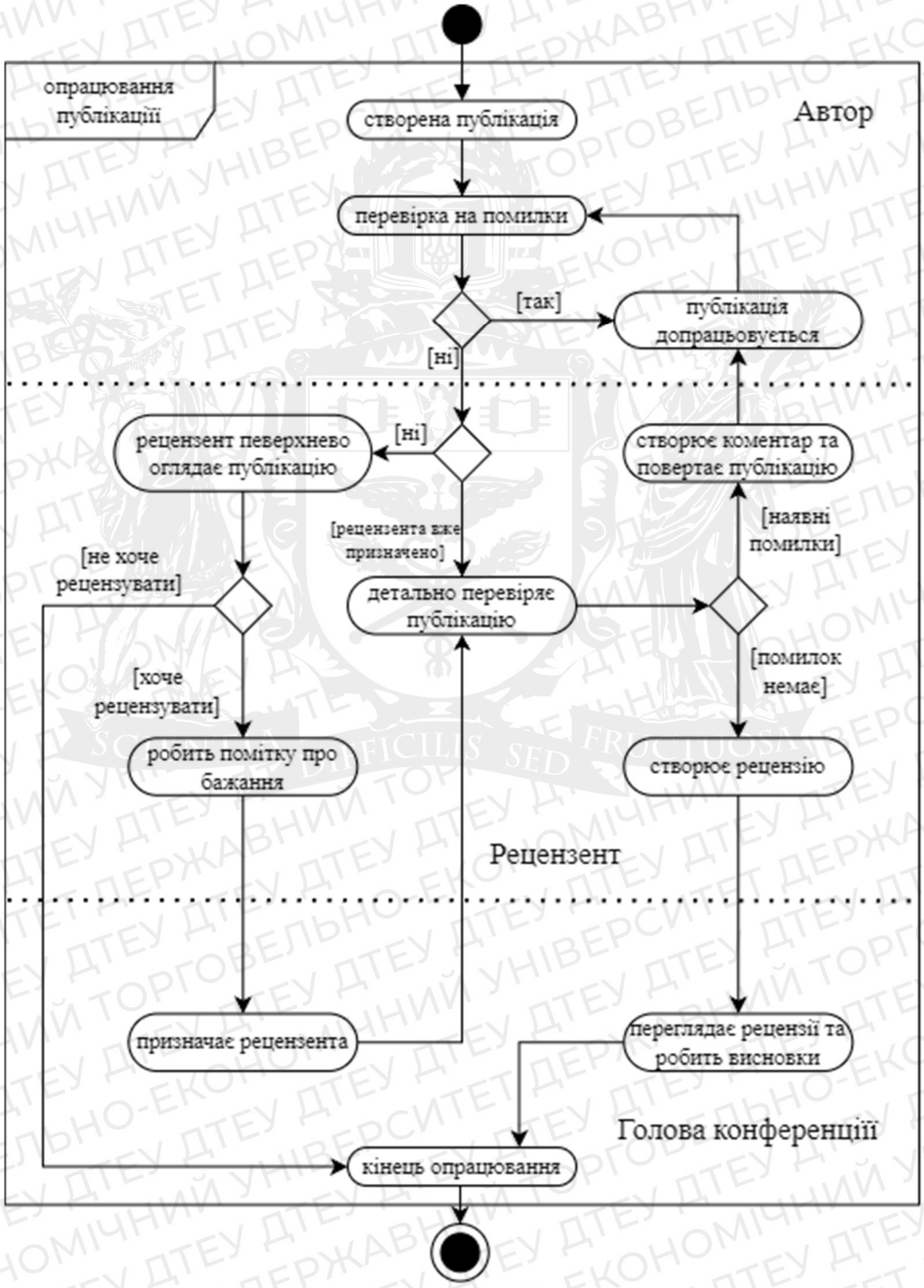


*Рис. 2.3. Діаграма активності опрацювання публікації*

## 2.3. Проєктування моделі додатку

### 2.3.1. Визначення основних сутностей

Моделювання даних – це процес створення візуального представлення всієї інформаційної системи або її частин для передачі зв'язків між точками даних і структурами. Мета моделювання – проілюструвати типи та структури даних, що використовуються та зберігаються в системі, взаємозв'язки між цими типами даних, способи групування та організації даних, а також їхні формати та атрибути [8].

Етапи проєктування моделі даних:

1. Аналіз бізнес вимог.
2. Ідентифікація сутностей.
3. Створення концептуальної моделі.
4. Створення логічної моделі.
5. Створення фізичної моделі.
6. Створення бази даних.

Спираючись на визначені раніше вимоги, можна відилити наступні основні сутності додатку:

- Користувач (User)
- Конференція (Conference)
- Публікація (Submission)
- Файл (Paper)
- Рецензія (Review)
- Коментар (Comment)

Додатковими сутностями будуть:

- Роль (Role)
- Роль користувача у конференції (UserConferenceRole)
- Призначення рецензента (SubmissionReviewer)

- Вподобання рецензента (ReviewPreference)

### 2.3.2. Концептуальна модель бази даних

Концептуальна модель даних (іноді модель предметної області) - дає загальне уявлення про те, що буде містити система, як вона буде організована і які бізнес-правила будуть задіяні. Концептуальна модель зазвичай створюються як частина процесу збору початкових вимог до проекту. Зазвичай вони включають класи сутностей (що визначають типи речей, які важливо представляти в моделі даних), їхні характеристики та обмеження, взаємозв'язки між ними, а також відповідні вимоги до безпеки та цілісності даних. Будь-яка нотація зазвичай проста.

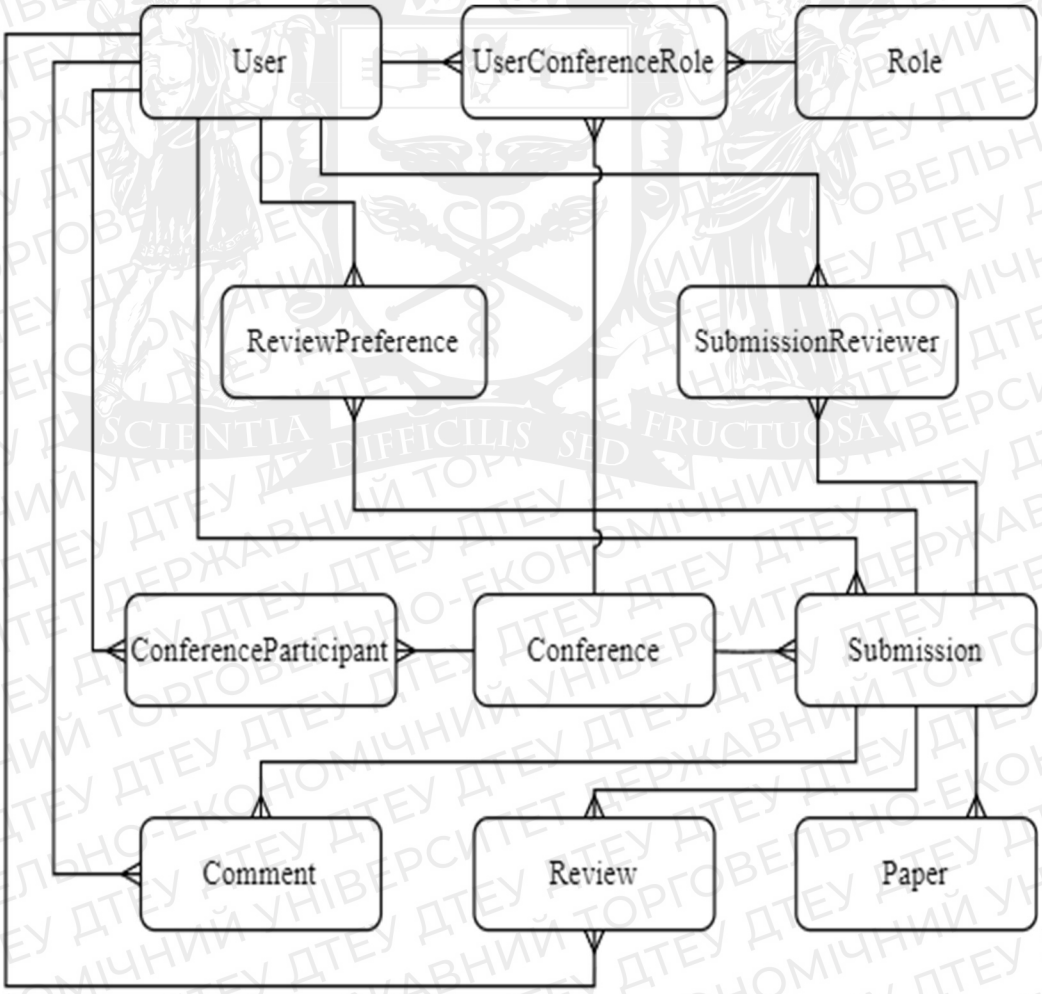


*Рис. 2.4. Концептуальна модель додатку*

### 2.3.3. Логічна модель бази даних

Логічна модель даних – менш абстрактна та надає більше деталей реалізації структури даних. Логічна модель вказує атрибути сутностей, проте не вдається у деталі реалізації.

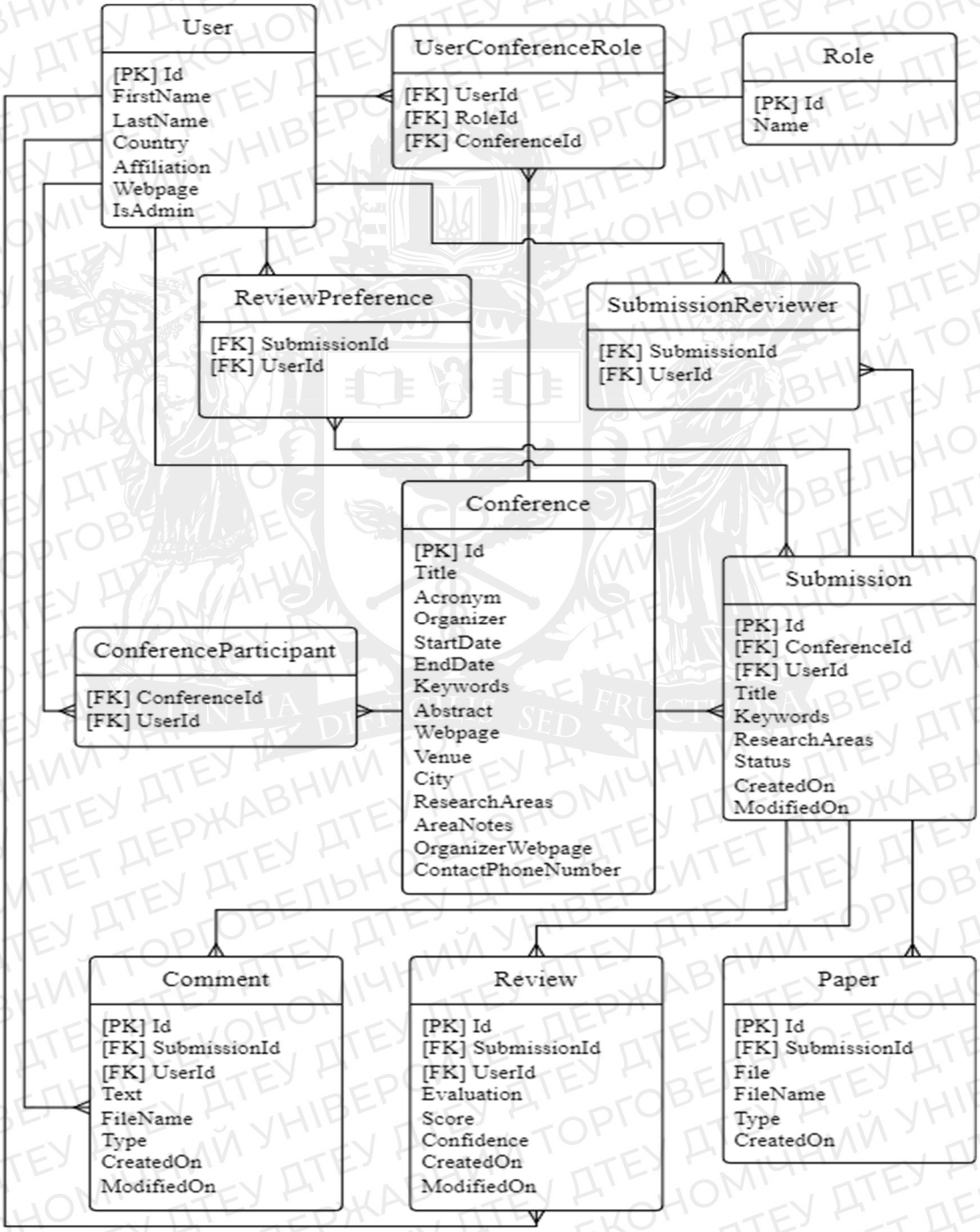


*Рис. 2.5. Логічна модель додатку*

### 2.3.4. Фізична модель бази даних

Фізична модель бази даних вказує на конкретну реалізацію концептуальної моделі у контексті обраної СКБД. Фізична модель зазначає типи даних атрибутів сутності та може включати додаткову інформацію, що властива обраній СКБД. Для реалізації додатку було обрано СКБД MS SQL Server.

*Таблиця 2.1.*

**ФІЗИЧНА МОДЕЛЬ БАЗИ ДАНИХ**

| Таблиця | Атрибут | Тип даних | Підтримує NULL | Ключ |
|---|---|---|---|---|
| User | Id | integer | NOT NULL | PK |
| | FirstName | nvarchar (50) | NOT NULL | - |
| | LastName | nvarchar (50) | NOT NULL | - |
| | Country | nvarchar (50) | NOT NULL | - |
| | Affiliation | nvarchar (50) | NOT NULL | - |
| | Webpage | nvarchar (100) | NULL | - |
| | IsAdmin | bit | NOT NULL | - |
| Conference | Id | integer | NOT NULL | PK |
| | Title | nvarchar (100) | NOT NULL | - |
| | Acronym | nvarchar (20) | NOT NULL | - |
| | Organizer | nvarchar (100) | NOT NULL | - |
| | StartDate | datetime2 | NOT NULL | - |
| | EndDate | datetime2 | NOT NULL | - |
| | Keywords | nvarchar (100) | NULL | - |
| | Abstract | nvarchar (1000) | NULL | - |
| | Webpage | nvarchar (100) | NULL | - |
| | Venue | nvarchar (100) | NULL | - |
| | City | nvarchar (50) | NULL | - |
| | ResearchAreas | nvarchar (500) | NOT NULL | - |
| | AreaNotes | nvarchar (1000) | NULL | - |
| | OrganizerWebpage | nvarchar (100) | NULL | - |
| | ContactPhoneNumber | nvarchar (20) | NULL | - |

*Продовження таблиці 2.1*

| | | | | |
|---|---|---|---|---|
| Submission | Id | integer | NOT NULL | PK |
| | UserId | integer | NOT NULL | FK |
| | ConferenceId | integer | NOT NULL | FK |
| | Title | nvarchar (100) | NOT NULL | - |
| | Keywords | nvarchar (100) | NOT NULL | - |
| | Abstract | nvarchar (1000) | NOT NULL | - |
| | ResearchAreas | nvarchar (500) | NOT NULL | - |
| | Status | tinyint | NOT NULL | - |
| | CreatedOn | datetime2 | NOT NULL | - |
| | ModifiedOn | datetime2 | NOT NULL | - |
| Paper | Id | integer | NOT NULL | PK |
| | SubmissionId | integer | NOT NULL | FK |
| | File | varbinary (MAX) | NOT NULL | - |
| | FileName | nvarchar (100) | NOT NULL | - |
| | Type | tinyint | NOT NULL | - |
| | CreatedOn | datetime2 | NOT NULL | - |
| Review | Id | integer | NOT NULL | PK |
| | SubmissionId | integer | NOT NULL | FK |
| | UserId | integer | NOT NULL | FK |
| | Evaluation | nvarchar (1000) | NOT NULL | - |
| | Score | smallint | NOT NULL | - |
| | Confidence | tinyint | NOT NULL | - |
| | CreatedOn | datetime2 | NOT NULL | - |
| | ModifiedOn | datetime2 | NOT NULL | - |
| Comment | Id | integer | NOT NULL | PK |
| | SubmissionId | integer | NOT NULL | FK |
| | UserId | integer | NOT NULL | FK |
| | Text | nvarchar (1000) | NOT NULL | - |
| | CreatedOn | datetime2 | NOT NULL | - |
| | ModifiedOn | datetime2 | NOT NULL | - |

Фізичні моделі таблиць-перетинів UserConferenceRole, SubmissionReviewer та ReviewPreference пропущено, оскільки вони складаються лише з композитного ключа, а таблиця Role складається з первинного ключа та стовпця з назвою.

### 2.4. Визначення предметних контекстів та сервісів додатку

Проаналізувавши вимоги та створену модель, можна помітити, що модель додатку логічно складається із трьох частин, що відповідно і будуть контекстами: контекст користувача, контекст конференції та контекст публікації.

Контекст користувача буде відповідальним за такі дії як: реєстрація та аутентифікація, оновлення, видалення та читання користувачів. Контекст користувача складається лише з одної сутності – User.

Контекст конференції буде відповідальним за операції над конференціями, а саме за створення, оновлення, видалення та читання конференцій, додавання нових користувачей та керування їх ролями у конференції.

Контекст конференції складається з наступних об’єктів:

- Сутності Conference
- Об’єкту-значення UserConferenceRole
- Об’єкту-значення ConferenceParticipant

Контекст публікації є найважливішим та охоплює найбілішу кількість об’єктів. Контекст публікації буде керувати операціями над публікацією та такими спорідненими об’єктами як файли, рецензії та коментарі, оскільки вони не можуть існувати окремо від публікації.

Контекст публікації складається з:

- Агрегату Submission, що включає у себе
  - Сутність Paper
  - Сутність Review

- Сутність Comment
- Об’єкт-значення SubmissionReviewer
- Об’єкт-значення ReviewPreference

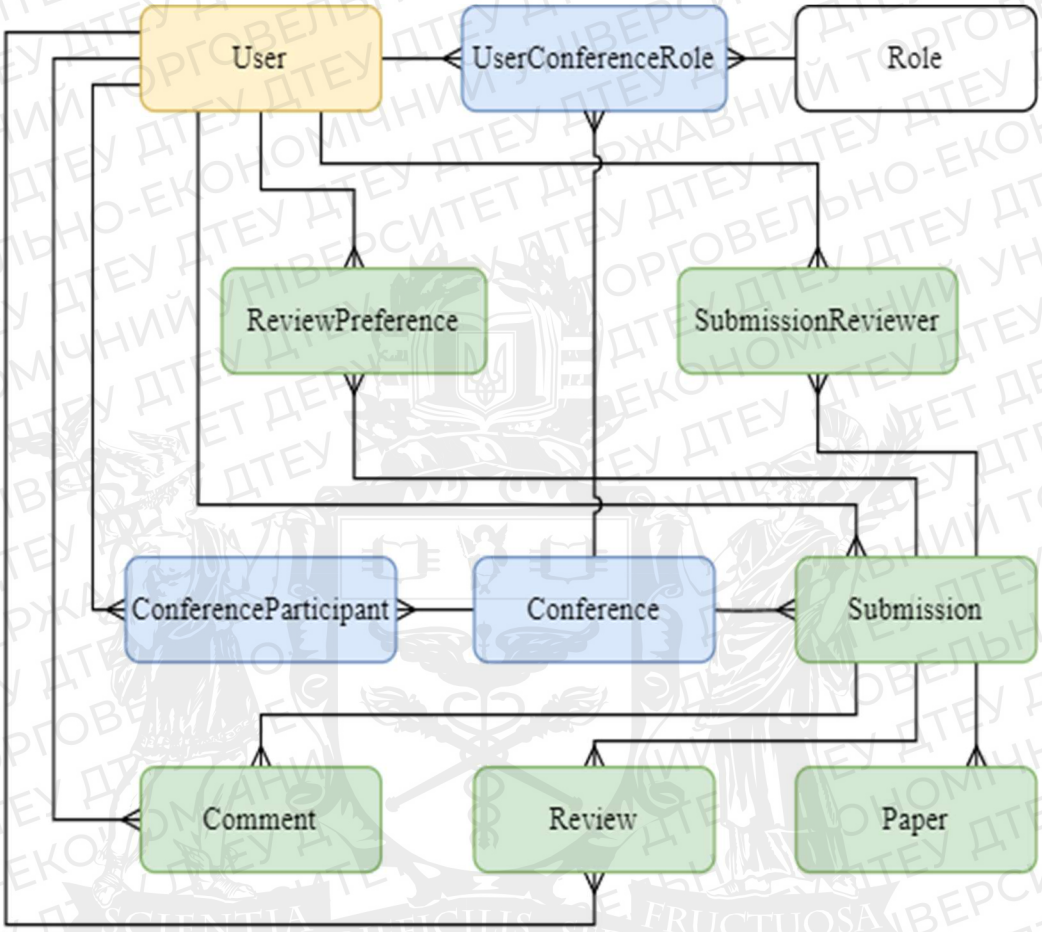


*Рис. 2.6. Концептуальна модель із визначеними предметними контекстами*

Можна помітити, що сутність Role не є частиною жодного контексту. Таке рішення біло прийняте, оскільки ролі у додатку зафіксовані, динамічне створення або видалення нових ролей не планується.

Оскільки предметна модель складається с 3 контекстів, додаток має мати мінімум 3 відповідних сервіси: User Service, Conference Service та Submission Service. Для більшої модульності та задля уникнення скупчення великої кількості операцій в Submission Service, можна відилити окремі Review Service та Commen tService, проте це не є обов’язковим.

Таким чином, додаток буде складатися із наступним сервісів:

- User Service
- Conference Service
- Submission Service

## 2.5. Висновки до розділу 2

У даному розділі було проведено аналіз вимог, побудовано діаграми прецедентів, послідовності та активності на базі зібраних вимог, виділено основні сутності додатку, спроектовано концептуальну, логічну та фізичну моделі бази даних, розділено предметну область на контексти та обрано відповідні сервіси.

В результаті проектування додатку, було відділено такі основні сутності як: User, Conference, Submission, Paper, Review, Comment. Предметну область було розділено на 3 контексти: User context, Conference context, Submission context. На базі утворених контекстів, було вирішено, що архітектура додатку буде складатися із 3 сервісів: User Service, Conference Service, Submission Service.

# РОЗДІЛ 3
# РОЗРОБКА ПРОГРАМНОГО ДОДАТКУ ІЗ ВИКОРИСТАННЯМ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

## 3.1. Розробка back-end частини

### 3.1.1. Реалізація доменної моделі

Створимо інтерфейс IEntity, що буде використовуватись як основа для всіх сутностей додатку [Додаток А]. Даний інтерфейс встановлює обмеження на тип даних первинного ключа, що буде використовуватись у всіх таблицях бази даних – int. Також, даний інтерфейс зазначає список доменних подій та методи для керування ними. Ці доменні події можуть використовуватись для розширення існуючої логіки додатку без прямого втручання до неї, шляхом реєстрації обробників подій. Доменні події викликаються при створенні або оновленні записів відповідної сутності.

Шаблонною реалізацією інтерфейсу **IEntity** буде абстрактний клас **BaseEntity** [Додаток Б]. Клас **BaseEntity** можна розширити класом **BaseAuditableEntity**, що додає нові поля для збереження інформації про автора запису та останнього користувача, що редагував запис [Додаток В].

Всі основні сутності, за виключенням **User**, будуть наслідуватись від класу **BaseEntity** або **BaseAuditableEntity**. Сутність **User** наслідується від класу **IdentityUser** для підтримки функціоналу ASP.NET Identity, пов’язаного із обліковими записами.

Розглянемо приклад сутності **Submission** [Додаток Г, Додаток Д]. Призначення більшості полів зрозуміле без пояснень, оскільки вони реалізують фізичну модель таблиці. Особливої уваги потребують віртуальні поля Conference, Reviews, Comments тощо.

Дані поля називаються полями «навігації», вони використовуються для відображення реляційних зв'язків між таблицями для Entity Framework.

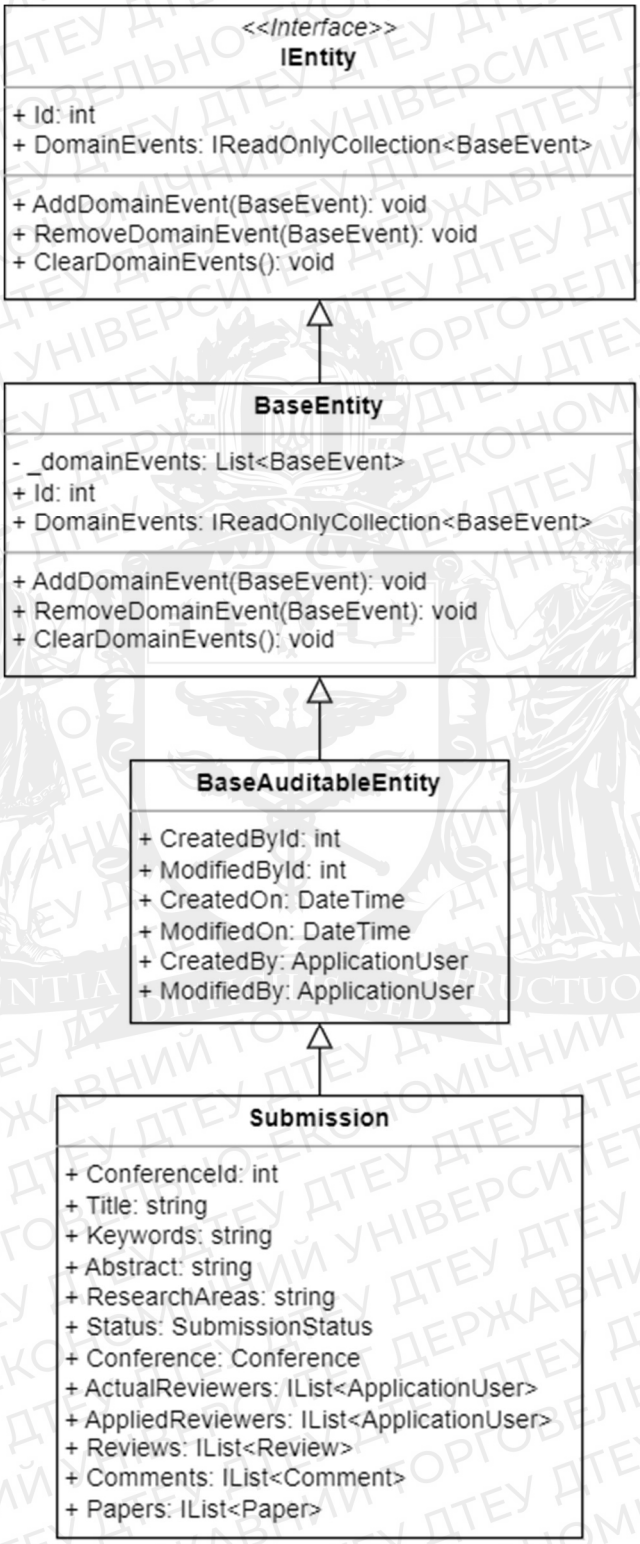


*Рис. 3.1. Діаграма класів сутності Submission*

Поле типу Conference у купі із зовнішнім ключем ConferenceId вказує на зв'язок N:1 між таблицями Submission та Conference, а поля типу IList вказують на зв'язок 1:N між Submission та відповідною таблицею. Діаграма класів сутності Submission вказано на рисунку 3.1.

Інші сутності додатку реалізовані аналогічним чином.

### 3.1.2. Реалізація взаємодії з базою даних

Головною умовою комунікації додатку із базою даних шляхом використання Entity Framework є наявність класу, що наслідується від стандартного **DbContext**. Бібліотека ASP.NET Identity надає власну реалізацію такого класу – **IdentityDbContext**, тому створимо свій клас **ApplicationDbContext** та унаслідуємо його від **IdentityDbContext** [Додаток Е].

Можна побачити, що всі таблиці, що використовуються у додатку, перелічено як властивості класу **ApplicationDbContext** із типом **DbSet**. Таблиці User, Role та інші допоміжні, що є частиною модуля Identity, вже включено в класі **IdentityDbContext**. Клас **DbContext** є готовою реалізацією шаблону Unit of Work, оскільки він дозволяє проводити транзакції, а DbSet – готова реалізація шаблону Repository, оскільки він підтримую стандартні CRUD операції над даними.

Іншим важливим аспектом використання Entity Framework є застосування налаштувань таблиць та реєстрація перехоплювачів.

Класи налаштування використовуються для зміни таких параметрів таблиць як: назва таблиці, назви та типи даних стовпців, обов'язковість стовпців, додаткове налаштування зв'язків між таблицями та каскадні правила тощо [Додаток И, Додаток К]. Класи налаштувань повинні обов'язково реалізовувати інтерфейс **IEntityTypeConfiguration**, що надається Entity Framework.

Класи перехоплювачі використовуються для реєстрації обробників для таких подій бази даних як: створення, оновлення, видалення записів тощо. За допомогою перехоплювачів було реалізовано функціонал відправлення доменних подій та уніфіковане збереження полів CreatedBy, ModifiedBy для сутностей що їх підтримують [Додаток Ж]. Класи перехоплювачі повинні обов'язково реалізовувати інтерфейс **IInterceptor**. Налаштування підключення до бази даних та реєстрація всіх сервісів у контейнері впровадження залежностей відбувається стандартним для ASP.NET застосунків засобом [Додаток Л].

### 3.1.3. Реалізація бізнес логіки

Головним архітектурним стилем, що використовувався при написанні бізнес логіки, було обрано CQRS (Command-Query Responsibility Segregation). Основним принципом даного стилю є суворий розподіл моделі та всіх операцій у системі на операції читання даних (Query) та операції зміни даних (Command). Операції читання завжди виконуються без побічних ефектів та завжди повертають дані, операції зміни мають побічні ефекти та повертають або не повертають дані в залежності від вимог, наприклад повернення ідентифікатору створеного запису.

Відповідно до цього стилю, класи що його реалізують можна поділити на наступні категорії:

- Запити (Query) – уособлюють запит до бази даних з метою повернення деякої інформації. Дані класи описують параметри, що необхідні для реалізації запиту та тип значення, що повертається запитом. Для повернення даних із запиту зазвичай використовуються DTO або ViewModel типи даних, запит не може використовувати доменні типи даних для повернення інформації.
- Обробники запитів (Query handler) – містять у собі логіку щодо реалізації запиту. Дані класи використовують параметри, що надав

запит та обов’язково повертають той тип даних, що було вказано у запиті.

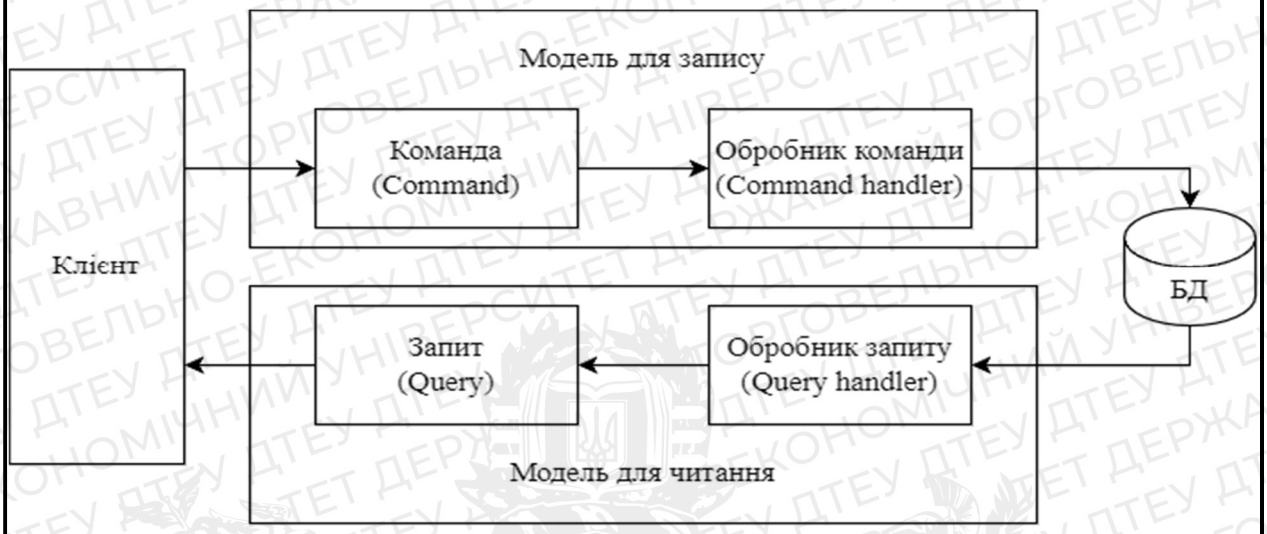


*Рис. 3.2. Архітектурний стиль CQRS*

- Класи моделі для читання – використовуються запитами та обробниками для визначення типів даних, якими вони оперують. Зазвичай такі класи відповідають архітектурному стилю DTO або ViewModel.
- Команди (Command) – уособлюють певну операцію, метою якої є зміна даних у БД або іншому джерелі. Як і запити, команди описують набір параметрів, що необхідний для її виконання, а також тип даних що потрібно повернути, якщо такий є.
- Обробники команд – містять у собі логіку щодо реалізації команди. Дані класи використовують параметри, що надала команда та можуть повернути той тип даних, що було вказано у запиті. Оскільки основною метою команд є зміна інформації у БД, обробники команд можуть користуватися доменною моделлю заради реалізації логіки.
- Класи моделі для запису – використовуються командами та обробниками.

Найпопулярнішою open-source бібліотекою, що допомагає реалізовувати шаблон CQRS у мові C# є MediatR. Дана бібліотека надає наступні інтерфейси для використання: **IRequest<TResponse>, IRequestHandler<TRequest, TResponse>** , що будуть описувати команди, запити та обробники відповідно; **ISender та Mediator**, що будуть використовуватись для передачі команди або запиту до відповідного обробника.

Створимо власний тип даних, **DbContextRequestHandler**, що буде описувати базовий обробник для запитів та команд [Додаток М]. Даний клас буде використовувати наступні сервіси:

- **IApplicationDbContext** – контекст доступу до бази даних.
- **IMappingHost** – сервіс, відповідальний за пошук відповідного маперу для команди, запиту або доменної моделі.
- **ICurrentUserService** – сервіс, що надає інформацію про користувача, який ініціював операцію.

Іншим корисним функціоналом бібліотеки MediatR є можливість налаштування конвеєру обробки запитів. За допомогою цієї функції можна додавати логіку, що виконується до або після всіх обробників, при цьому не змінюючи логіку самих обробників. Таким чином дуже зручно додавати функціонал, який стосується всіх аспектів додатку, наприклад глобальне логування [Додаток Н], валідація [Додаток П] тощо. Для реалізації валідації команд та запитів було використано популярну бібліотеку Fluent Validation.

Таким чином, розробка певного функціоналу складається з наступних основних кроків:

1. Визначення типу операції, створення відповідного запиту або команди. Визначення параметрів, що потрібні для виконання операції, визначення типу даних, що повертаються.

2. Створення валідатору, якщо параметри запиту або команди потребують додаткових перевірок перед виконанням основної логіки.
3. Створення маперу, якщо команду треба перетворити у певний доменний об’єкт, або якщо доменний об’єкт потрібно перетворити на певний DTO об’єкт для повернення із запиту.
4. Створення обробника команди або запиту, що містить у собі основну логіки щодо виконання операції.

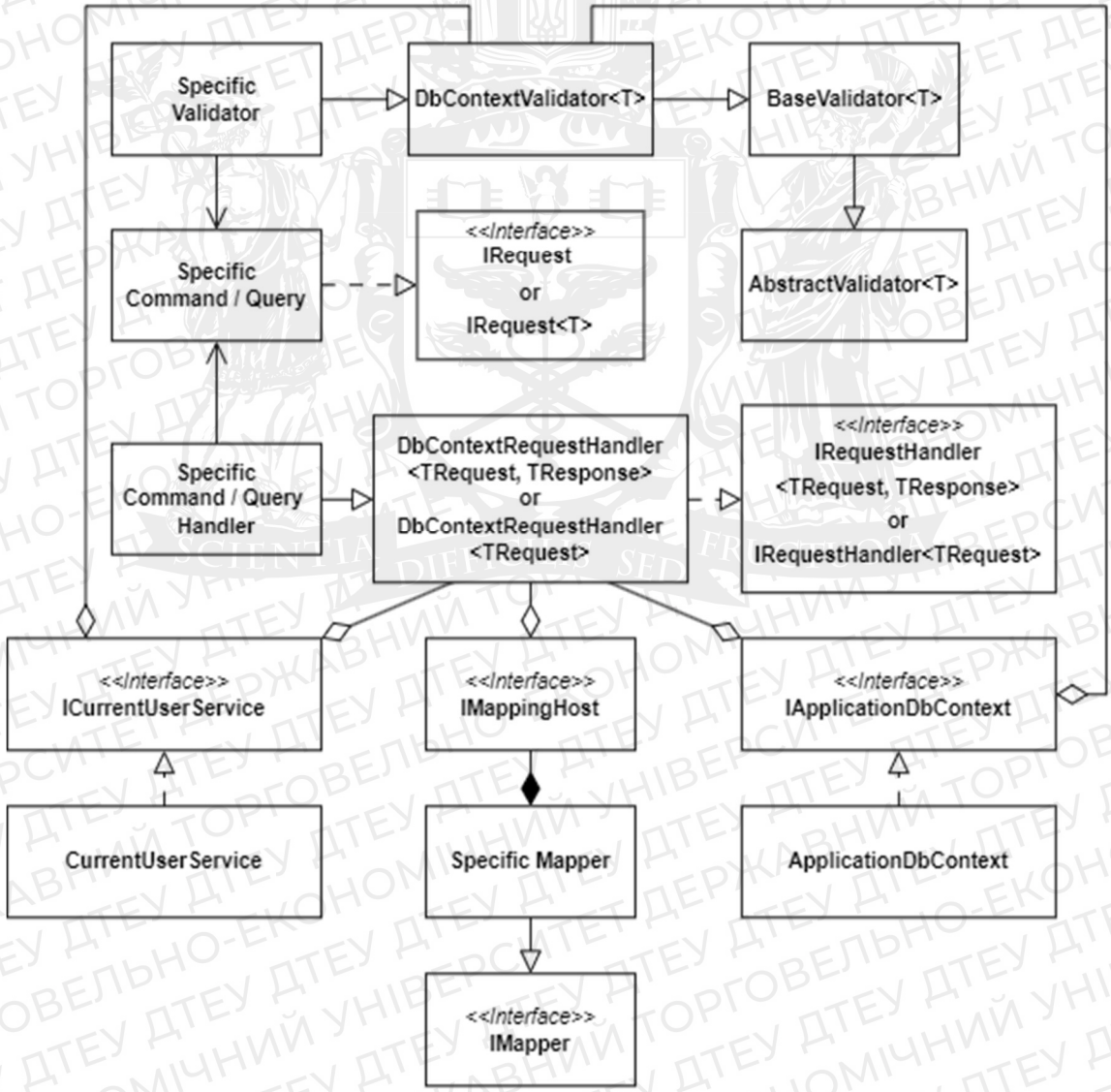


*Рис. 3.3. Діаграма основних класів шару бізнес логіки додатку*

### 3.1.4. Реалізація API сервісів

Для кожного мікросервісу було створено окремий проєкт ASP.NET Web API, спільну логіку було винесено до бібліотеки класів.

Для обробки HTTP запитів використовуються класи контролери, що повинні обов'язково наслідуються від стандартного **ControllerBase** та використовувати атрибут **ApiController**. Створимо власний базовий клас **ApiControllerBase**, що буде використовуватись у якості бази для всіх майбутніх контролерів [Додаток Р]. Даний клас зазначає наступні атрибути, що будуть застосовані до всіх контролерів, які наслідуються від **ApiControllerBase**:

- ApiController – обов'язковий стандартний атрибут, що необхідний для реєстрації класу в інфраструктурі ASP.NET.
- ApiExceptionFilter – власний атрибут, що використовується для глобальної обробки помилок.
- Produces – стандартний атрибут, що використовується в цілях документації. Вказує на те, що всі методи контролерів будуть повертати дані у форматі JSON.
- SwaggerResponse – атрибут, що використовується в цілях документації. Вказує на те, які HTTP статус коди може повертати контролер, та на типи даних, пов'язані із цими кодами.
- Route – стандартний атрибут, що визначає за якою відносною адресою будуть доступні методи контролеру. Параметр шляху [controller] вказує на те, що адреса буде містити назву класу контролеру, тобто класу UserController буде відповідати адреса «/api/user».

За своєю суттю контролери відіграють роль зв'язуючої ланки, що поєднують певний HTTP запит із певною логікою його обробки. Самі контролери не повинні містити логіку обробки запиту, лише передавати

його далі по стеку. Саме задля цього в базовий клас **ApiControllerBase** було поміщено медіатор **ISender**, що буде використовуватись для виклику відповідного обробника запиту.

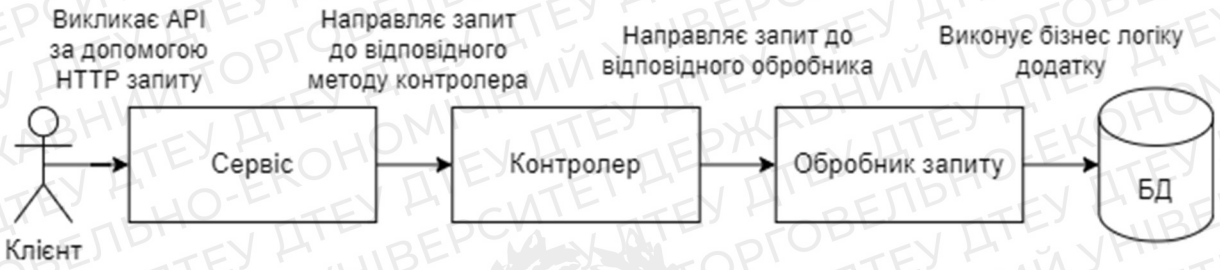


*Рис. 3.4. Послідовність обробки запиту сервісом*

Розглянемо частину класу **UserController** [Додаток С]. Даний приклад складається з двох методів, що використовуються для реєстрації та аутентифікації користувачів. Для того щоб метод класу мав можливість відповідати на HTTP запити, для нього обов'язково потрібно вказати тип HTTP методу, якому він відповідає. Для цього використовуються атрибути HttpGet, HttpPost, HttpPut, HttpDelete тощо. Атрибут Route повторно використовується для розширення адреси, на яку відповідає метод; даний атрибут не є обов'язковим, у випадку його відсутності метод буде мати адресу вказану у самому контролері. Декілька різних методів класу можуть бути зареєстровані на одну й ту саму адресу, але вони обов'язково повинні відповідати різним HTTP методам.

Механізми аутентифікації та авторизації користувачів реалізовано за допомогою стандартного функціоналу ASP.NET Identity у поєднанні із JWT токенами, що зберігаються у куках.

JWT – це відкритий стандарт для створення токенів доступу у форматі JSON. JWT токени створюються сервером на базі інформації про користувача, підписуються та відправляються користувачу. При створені запитів до захищених ресурсів, користувач повинен додати отриманий токен до запиту, а задача сервера його перевірити.

Для генерації та інтеграції JWT токенів в інфраструктуру ASP.NET було використано пакети System.IdentityModel.Tokens.Jwt та Microsoft.AspNetCore.Authentication.JwtBearer.

Розглянемо процес генерації токену [Додаток Т]. Даний алгоритм збирає певну інформацію про користувача, а саме список його конференцій, його ролі тощо, а також інформацію про самий токен: час його придатності, ключ, алгоритм підпису – та записує в об'єкт класу **SecurityTokenDescriptor**. Дана інформація буде використовуватись для перевірки валідності токену. Далі, власне значення токену створюється за допомогою класу **JwtSecurityToken** та записується у куки, що повернуться із запитом до користувача. Для забезпечення безпеки куків, вони створюються із параметром httpOnly, що уникає можливість їх читання за допомогою JavaScript у клієнтській частині. Такий підхід до збереження токену використовується для запобігання XSS атак.

Оскільки куки були створені із параметром httpOnly, це означає що їх неможливо використовувати напряму у клієнтській частині додатку, вони будуть надіслані браузером із кожним запитом автоматично. Проте, відповідно до стандарту JWT, токен повинен передаватися у Authorization хедері. Для вирішення даної проблеми було використано іншу надзвичайно потужну функціональність ASP.NET – middleware.

Middleware (проміжне програмне забезпечення) у ASP.NET – це спосіб додавання допоміжної логіки бля обробки всіх HTTP запитів та відповідей, що оброблює та створює сервер. Так само як pipeline behavior у MediatR, middleware допомагають розширювати основну логіку контролеру, не змінюючи його внутрішню реалізацію. Middleware створюються шляхом реалізації інтерфейсу **IMiddleware**.

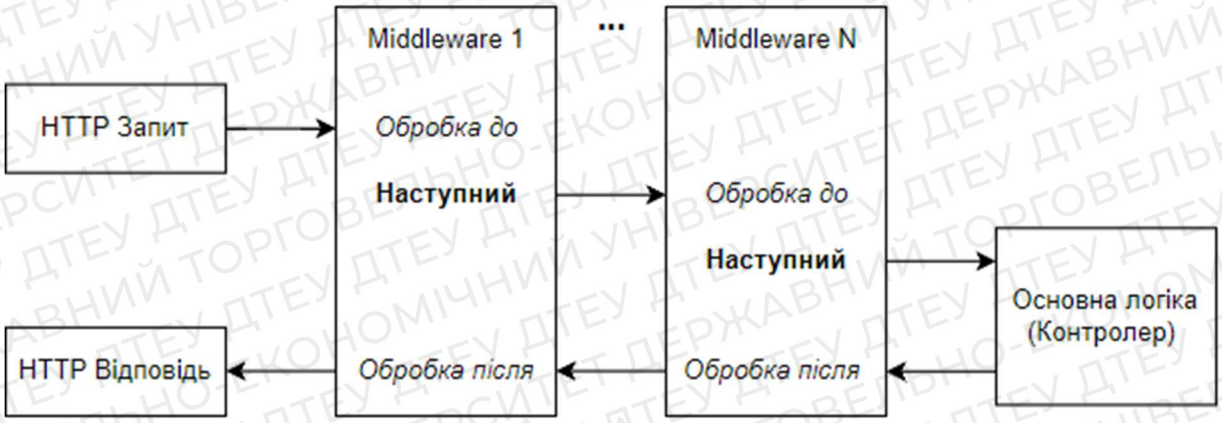


*Рис. 3.5. Middleware архітектура*

За допомогою класу **JwtCookieMiddleware** [Додаток У], відправлені клієнтом куки перевіряються на наявність токену. Якщо токен присутній, він копіюється в Authorization хедер та продовжується хід обробки запиту. Далі спрацьовує вбудоване middleware, що відповідальне за валідацію токену та авторизацію запиту.

Для захисту контролерів та методів від несанкціонованого доступу та для увімкнення механізму авторизації використовується атрибут **Authorize**, що приймає перелік ролей як параметр. Хоча даний механізм і перевіряє, чи має користувач необхідні ролі для доступу до ресурсу, він робить це без урахування конференцій, тому для розширення стандартного механізму необхідно реалізувати власний фільтр авторизації. Для цього створимо новий клас **ConferenceAuthorizationFilter**, що реалузіє **IAuthorizationFilter** [Додаток Ф]. Даний клас отримує список ролей у якості параметру, шукає хедер **x-conference-id** у запиті, що визначає у контексті якої конференції відбувається запит та перевіряє, чи є користувач учасником конференції та чи має відповідну роль.

Для збереження всіх налаштувань сервісу використовується стандартний для ASP.NET додатків засіб із використанням appsettings.json файлу [Додаток Х]. Даний файл містить налаштування, що стосуються:

- Логування
- Підключення до бази даних
- Генерації JWT токену
- Аутентифікації для рутового користувача
- CORS

Для локального запуску у режимі Debug використовується файл appsettings.Development.json, що зберігає параметри налаштування напряму, даний файл не зберігається у системі контролю версій, розробники створюють його відповідно до конфігурації власної системи. Для запуску у режиму Release – appsettings.json. Можна помітити, що даний файл не містить ніяких реальних налаштувань, тільки константи вигляду DB_SERVER, DB_USER тощо. Дані константи будуть підмінені реальними значеннями у процесі розгортання за допомогою Docker.

Запуск сервісу локально здійснюється за допомогою команди dotnet run --project .\ConferenceManager\ConferenceManager.Api.<назва сервісу>. Після запуску сервісу можна побачити стандартний вивід: інформацію про застосовані міграції БД та адресу.

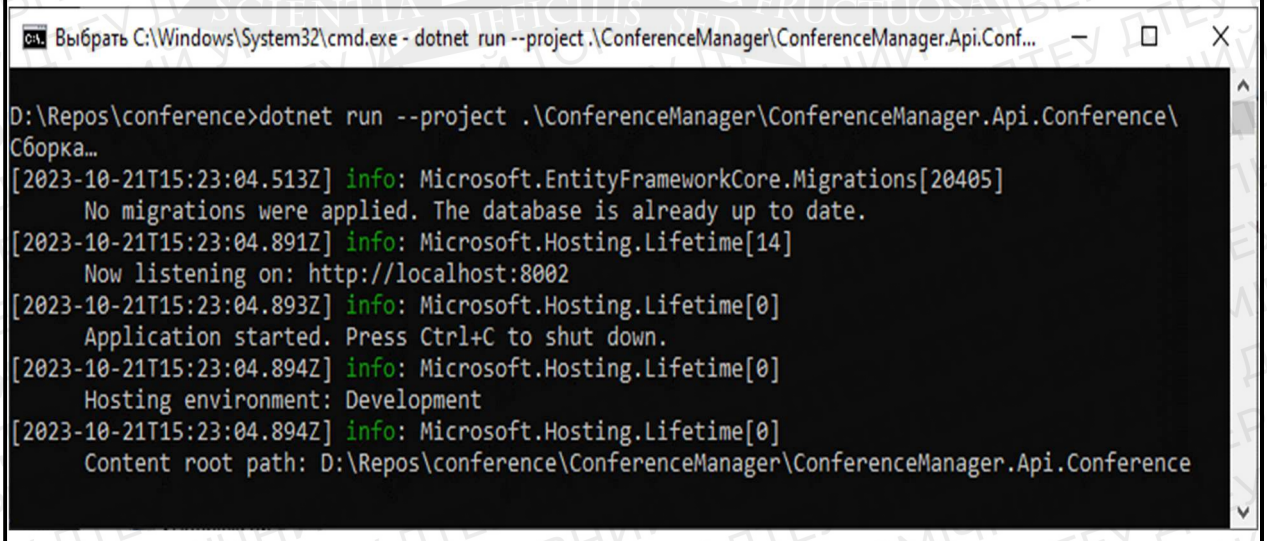


*Рис. 3.6. Приклад запуску Conference сервісу*

### 3.2. Розробка front-end частини

Клієнтська частина була виконана у вигляді SPA додатку із використанням мікрофреймворку React та мови програмування TypeScript.

Головним принципом React є створення додатку як сукупності малих незалежних компонентів, що містять у собі певну частину інтерфейсу та логіку її обробки. Розбиття додатку на такі компоненти допомагає легше вирішувати великі та складні задачі шляхом розбиття не менші та також забезпечує краще повторне використання коду. React заохочує до використання такої парадигми як декларативне програмування, основним принципом якої є опис поведінки програми та бажаного кінцевого результату, проте не конкретного алгоритму дій.

Для реалізації посторінкової навігації було обрано бібліотеку React Router. За своєю суттю SPA додатки – це додатки що складаються лише з однією кореневої HTML сторінки. React Router використовується для імітації навігації між посиланнями так, як посилання працювали би між різними HTML сторінками, проте зі збереженням парадигми та всіх переваг SPA.

У якості HTTP клієнта для надсилання запитів до сервісів було обрано бібліотеку Axios. Axios працює на базі XMLHttpRequest у браузері та вбудованого модуля http у серверному середовищі node.js. Головними функціональними особливостями Axios є підтримка Promise, просунута обробка помилок та наявність перехоплювачів запитів, що допомагають впроваджувати додаткову логіку для всіх запитів та відповідей від серверу.

Для створення користувацького інтерфейсу було обрано бібліотеку компонентів Mui. Для керування та валідації форм було використано бібліотеки Formik та Yup відповідно.

Розглянемо приклад реалізації форми для аутентифікації користувача [Додаток Ц]. Інтерфейс даної форми є дуже простим, він складається з двох

компонентів Mui для вводу - TextField, компоненту Mui Button та власного компоненту FormErrorAlert, що відображає помилку запиту до сервісу, якщо така сталася. Можна помітити, що компонент не має власного стану, а користується різноманітними хуками для реалізації логіки та збереження введених даних:

- useNavigate – повертає функцію React Router, що використовується для навігації по «сторінкам».
- usePostLoginApi – власний хук, що використовує Axios для відправлення запиту та React.useState для збереження поточного стану запиту.
- useFormik – повертає об’єкт formik, що представляє собою поточний стан форми, разом із логікою його оновлення. Валідація форми впроваджується за допомогою параметру validationSchema, що описується за допомогою бібліотеки Yup.
- useEffect – один із найважливіших вбудованих хуків React, використовується тоді, коли необхідно виконати дії з «побічними ефектами», наприклад: HTTP запит, робота із зовнішніми ресурсами, маніпуляція DOM деревом тощо. Оскільки навігація до інших компонентів є зміною DOM дерева, вона повинна здійснюватися за допомогою useEffect.

Налаштування для додатку, а саме адреси мікросервісів, зберігається у файлі .env.

```
1  VITE_USER_API_URL=http://localhost:8001/api
2  VITE_CONFERENCE_API_URL=http://localhost:8002/api
3  VITE_SUBMISSION_API_URL=http://localhost:8003/api
```

*Рис. 3.7. Файл конфігурації для фронт-енд частини*

Даний файл не можна використовувати для збереження конфіденційної інформації, наприклад ключів доступу, оскільки весь його

зміст завантажується разом із самим сайтом. Так само як і з серверною частиною додатку, зміст даного файлу оброблюється під час процесу розгортання додатку за допомогою Docker. Локальний запуск фронт-енду здійснюється за допомогою команди npm run start.

### 3.3. Розгортання додатку за допомогою Docker

Головним засобом для розгортання додатку було обрано платформу Docker. Docker – це програмне забезпечення, що використовується для створення, керування та запуску додатку у якості контейнерів. Docker характеризується такими основними поняттями:

- Docker Engine – ядро платформи, відповідальне за безпосереднє створення та керування роботою контейнерів. Головними компонентами движку Docker є Docker Daemon, фоновий процес, що керує контейнерами, образами, томами тощо та Docker Client, що використовується для комунікації з користувачем.
- Dockerfile – набір інструкцій, що виконуються послідовно з метою пакування ПЗ та створення образу контейнеру.
- Образ – запаковане програмне забезпечення разом із залежностями необхідними для його роботи.
- Контейнер – середовище виконання, що було створено на базі образу. Містить в собі все необхідне для запуску додатку. Контейнери – лише засіб розгортання додатку, вони створюються та видаляються за вимогами і не можуть використовуватись для довготривалого збереження даних, оскільки всі дані контейнеру видаляються разом із ним.
- Том – використовуються у випадках, коли контейнеру потрібно мати доступ до довготривалого сховища даних. Один том може використовуватись декількома контейнерами.

- Docker Hub – найбільший відкритий реєстр для збереження, завантаження та поширення образів, використовується Docker за замовчуванням.
- Docker Compose – інструмент, що використовується для розгортання додатку, що складається з декількох контейнерів.

Будь яка робота із Docker починається зі створення Dockerfile. Почнемо із опису Dockerfile для сервісної частини додатку. Для кожного сервісу буде створено окремий Dockerfile, проте вони будуть відрізнятись тільки точкою входу та портами, тому розглянемо приклад файлу для User сервісу [Додаток Ш]:

Даний файл складається із 4 етапів:

1. Етап base – вказує який образ буде використовуватись у якості основного для роботи сервісу. Для запуску та роботи сервісів використовувався офіційний образ ASP.NET версії .NET 7. За допомогою команди ENV вказується змінна середовища ASPNETCORE_URLS, що використовується для зазначення порту.
2. Етап build – вказує який образ буде використовуватись для зборки сервісу. Головним призначенням даного етапу є підстановка аргументів збірки (ARG, RUN sed) у файл конфігурації appsettings.json та власне збірка програмного коду (COPY, RUN dotnet).
3. Етап publish – створений на основі попереднього етапу, використовується для генерації кінцевих файлів, що будуть використовуватись для запуску сервісу.
4. Етап final – створений на основі етапу base, копіює кінцеві файли з етапу publish та запускає сервіс.

Файл для фронт-енду є схожим за своєю суттю, проте використовує різні засоби для збірки та запуску коду [Додаток Щ]:

1. Етап build – використовує середовище node.js для збірки коду. Так само використовує аргументи збірки для оновлення файлу створення коректного файлу конфігурації .env.
2. Етап final – використовує Nginx у якості фронт-енд серверу.

Для створення образів за даним файлами необхідно використовувати команду docker build із обов'язковим переліком всіх аргументів зборки, що були зазначені. Після створення образів, можна власноруч запустити контейнери за допомогою команди docker run, із вказанням налаштування портів, томів тощо. Окремо необхідно здійснювати налаштування мережі, щоб контейнери сервісів мали доступ до контейнеру бази даних. Всі вказані процеси можна полегшити та автоматизувати за допомогою інструменту Docker Compose, що використовується саме у подібних випадках.

Для застосування функціоналу docker compose необхідно створити додатковий файл docker-compose.yml [Додаток Ю]. Даний файл збирає в одному місці всі налаштування, що необхідні для запуску додатку, а саме:

- назви контейнерів та образів для їх створення,
- налаштування портів та мережі,
- список томів та їх підключення до контейнерів,
- аргументи збірки та змінні середовища.

Також в цьому файлі можна вказувати залежності між контейнерами, для випадків коли порядок їх запуску має значення та наявність одного контейнеру необхідна для роботи іншого.

Можна помітити, що даний файл також не містить у собі кінцевих значень змінних так аргументів. Оскільки даний файл зберігається у системі контролю версій, для збереження та захищення змінних використовується додатковий .env файл.

Окрім опису одного фронт-енд контейнеру та трьох контейнерів для сервісів, даних файл також описує контейнер для серверу бази даних SQL

Server, що заснований на офіційному образу SQL Server 2022, а також контейнер, що використовується для створення бекапів бази даних. Для збереження власне бази даних використовується том db-data-vol.

Запуск додатку здійснюється за допомогою команди docker compose up -d. Вимкнення та видалення контейнерів здійснюється за допомогою docker compose down.

## 3.4. Висновки до розділу 3

У даному розділі було детально розглянуто процес створення додатку із використанням мікросервісної архітектури. Були описані основні інструменти, архітектурні стилі та шаблони, проблеми та способи їх вирішення, що були зустрінуті у процесі розробки.

Під розробки сервісної частини додатку було використано на практиці такі інструменті як: C#, .NET, ASP.NET Web API, Entity Framework, MediatR. Було застосовано архітектурні стилі Unit of Work, Repository, Command Query Responsibility Segregation, шаблони посередник, описано основні класи, що використовувались для реалізації логіки.

Для реалізації клієнтської частини було застосовано бібліотеку React у поєднанні з бібліотеками React Router, Axios, Formik та Yup. Було розглянуто приклад реалізації одного із компонентів додатку, розглянуто на практиці приклади застосування різноманітних хуків.

Були описані основні поняття платформи Docker, як створювати образи та контейнери, розгортати їх поодинці та як єдиний багатоконтейнерний додаток за допомогою Docker Compose.

# ВИСНОВКИ ТА ПРОПОЗИЦІЇ

В ході роботи над магістерським кваліфікаційним проєктом було проаналізовано поняття мікросевісної архітектури програмного забезпечення, визначено її основні поняття та характеристики. Отримані навички було успішно застосовано у процесі розробки додатку для кафедри інженерії програмного забезпечення та кібербезпеки. Розроблений додаток можна використовувати для проведення наукових конференцій у якості сховища даних та інструмента для збору рецензій. Додаток є доступним до використання за інтернет адресою http://cm.knute.edu.ua.

У першому розділі роботи було проведено ретельний аналіз програмного забезпечення, розробленого з використанням мікросервісної архітектури. Зокрема, були вивчені передумови її виникнення, порівняння із монолітним видом архітектури, основні принципи, переваги та недоліки цієї архітектурної парадигми, проблеми зв'язності та пов'язаності, та засоби їх вирішення.

На основі аналізу було виявлено, що мікросервісна архітектура дозволяє розділити складну систему на невеликі, незалежні компоненти, що спрощує розробку, тестування та розгортання програмного забезпечення. Крім того, вона забезпечує гнучкість та масштабованість системи, що дозволяє ефективно відповідати на зростаючі потреби користувачів.

Загалом, дослідження підтверджує, що розробка програмного забезпечення з використанням мікросервісної архітектури є перспективним підходом, особливо для великих та розподілених систем.

Другий розділ роботи було присвячено аналізу поставленої задачі, виділенню специфічних вимог та проєктуванню архітектури додатку на базі проведеного дослідження. У результаті аналізу вимог було створено діаграми прецедентів, послідовності та активності, що описують основні варіанти використання програми, спроектовано концептуальну, логічну та фізичні моделі бази даних, виділено основні сутності та предметні контексти додатку.

У третьому розділі даної роботи було ретельно задукоментовано процес розробки кінцевого програмного забезпечення із використанням всіх отриманих навичок, зібраних вимог та спроєктованих моделей додатку. Було проведено ретельний опис всіх найважливіших використаних технологій, описано важливі архітектурні проблеми та шляхи їх рішення.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Fowler M. Microservices: a definition of this new architectural term [Електронний ресурс] / Martin Fowler // martinfowler.com. – 2014. – Режим доступу до ресурсу: https://martinfowler.com/articles/microservices.html.

2. Richards M. Fundamentals of Software Architecture / M. Richards, F. Neal. – Sebastopol: O'Reilly Media, 2020. – 265 с.

3. Newman S. Building Microservices / Sam Newman. – Sebastopol: O'Reilly Media, 2015. – 265 с.

4. Newman S. Monolith to Microservices / Sam Newman. – Sebastopol: O'Reilly Media, 2020. – 273 с.

5. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software / Eric Evans. – Boston: Addison Wesley, 2003. – 560 с.

6. Yourdon E. Structured design / E. Yourdon, L. Constantine. – Hoboken: Prentice Hall, 1975. – 473 с.

7. Hamilton K. Learning UML 2.0 / K. Hamilton, R. Miles. – Sebastopol: O'Reilly Media, 2006. – 286 с.

8. What is data modeling? [Електронний ресурс] // IBM Cloud Education – Режим доступу до ресурсу: https://www.ibm.com/topics/data-modeling.

9. Bogard J. MediatR [Електронний ресурс] / Jimmy Bogard – Режим доступу до ресурсу: https://github.com/jbogard/MediatR/wiki.

10. CQRS pattern [Електронний ресурс] // Microsoft Learn – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs.

11. Milett S. Patterns, Principles, and Practices of Domain-Driven Design / S. Milett, N. Tune. – Birmingham: Wrox, 2015. – 800 с.

12. Nickoloff J. Docker in Action / J. Nickolof – Shelter Island: Manning, 2019 – 336

# ДОДАТКИ

*Додаток А*

```
namespace ConferenceManager.Domain.Common
{
    public interface IEntity
    {
        public int Id { set; get; }

        public IReadOnlyCollection<BaseEvent> DomainEvents { get; }

        public void AddDomainEvent(BaseEvent domainEvent);

        public void RemoveDomainEvent(BaseEvent domainEvent);

        public void ClearDomainEvents();
    }
}
```

```
using System.ComponentModel.DataAnnotations.Schema;

namespace ConferenceManager.Domain.Common;

public abstract class BaseEntity : IEntity
{
    private readonly List<BaseEvent> _domainEvents = new();

    [NotMapped]
    public IReadOnlyCollection<BaseEvent> DomainEvents =>
_domainEvents.AsReadOnly();

    public int Id { get; set; }

    public void AddDomainEvent(BaseEvent domainEvent)
    {
        _domainEvents.Add(domainEvent);
    }

    public void RemoveDomainEvent(BaseEvent domainEvent)
    {
        _domainEvents.Remove(domainEvent);
    }

    public void ClearDomainEvents()
    {
        _domainEvents.Clear();
    }
}
```

```
using ConferenceManager.Domain.Entities;

namespace ConferenceManager.Domain.Common;

public abstract class BaseAuditableEntity : BaseEntity
{
    public DateTime CreatedOn { get; set; }

    public int CreatedById { get; set; }

    public virtual ApplicationUser CreatedBy { get; set; } = null!;

    public DateTime ModifiedOn { get; set; }

    public int ModifiedById { get; set; }

    public virtual ApplicationUser ModifiedBy { get; set; } = null!;
}
```

```
using ConferenceManager.Domain.Common;
using ConferenceManager.Domain.Enums;

namespace ConferenceManager.Domain.Entities
{
    public class Submission : BaseAuditableEntity
    {
        public int ConferenceId { set; get; }

        public required string Title { set; get; }

        public required string Keywords { set; get; }

        public required string Abstract { set; get; }

        public required string ResearchAreas { set; get; }

        public required SubmissionStatus Status { set; get; }

        public virtual Conference Conference { set; get; } = null!;

        public virtual IList<ApplicationUser> ActualReviewers { set; get; } = null!;

        public virtual IList<ApplicationUser> AppliedReviewers { set; get; } =
null!;

        public virtual IList<Review> Reviews { set; get; } = null!;

        public virtual IList<Comment> Comments { set; get; } = null!;

        public virtual IList<Paper> Papers { set; get; } = null!;
    }
}
```

```
using ConferenceManager.Core.Common.Interfaces;
using ConferenceManager.Domain.Entities;
using ConferenceManager.Infrastructure.Persistence.Interceptors;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using System.Reflection;

namespace ConferenceManager.Infrastructure.Persistence
{
    public class ApplicationDbContext :
        IdentityDbContext<
            ApplicationUser, ApplicationRole, int,
            IdentityUserClaim<int>, UserConferenceRole, IdentityUserLogin<int>,
            IdentityRoleClaim<int>, IdentityUserToken<int>>,
        IApplicationDbContext
    {
        private readonly DomainEventsSaveChangesInterceptor
_domainEventsInterceptor;
        private readonly AuditableEntitySaveChangesInterceptor
_auditableEntityInterceptor;
        private readonly ForeignKeysSaveChangesInterceptor _foreignKeysInterceptor;

        public DbSet<Conference> Conferences => Set<Conference>();

        public DbSet<Paper> Papers => Set<Paper>();

        public DbSet<Comment> Comments => Set<Comment>();

        public DbSet<Review> Reviews => Set<Review>();

        public DbSet<Submission> Submissions => Set<Submission>();

        public DbSet<SubmissionReviewer> SubmissionReviewers =>
Set<SubmissionReviewer>();

        public DbSet<ReviewPreference> ReviewPreferences => Set<ReviewPreference>();

        public DbSet<ConferenceParticipant> ConferenceParticipants =>
Set<ConferenceParticipant>();

        public DbSet<InviteCode> InviteCodes => Set<InviteCode>();

        public ApplicationDbContext(
            DbContextOptions<ApplicationDbContext> options,
            DomainEventsSaveChangesInterceptor domainEventsInterceptor,
            AuditableEntitySaveChangesInterceptor auditableEntityInterceptor,
            ForeignKeysSaveChangesInterceptor foreignKeysInterceptor
            ) : base(options)
        {
            _domainEventsInterceptor = domainEventsInterceptor;
            _auditableEntityInterceptor = auditableEntityInterceptor;
            _foreignKeysInterceptor = foreignKeysInterceptor;
        }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);


builder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());
        }
```

```
        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
            optionsBuilder.AddInterceptors(
                _domainEventsInterceptor,
                _auditableEntityInterceptor,
                _foreignKeysInterceptor);
        }
    }
}
```

```
using MediatR;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;

namespace ConferenceManager.Infrastructure.Persistence.Interceptors
{
    public class DomainEventsSaveChangesInterceptor : SaveChangesInterceptor
    {
        private readonly IMediator _mediator;

        public DomainEventsSaveChangesInterceptor(IMediator mediator)
        {
            _mediator = mediator;
        }

        public override int SavedChanges(SaveChangesCompletedEventData eventData,
int result)
        {
            DispatchEvents(eventData.Context).GetAwaiter();

            return base.SavedChanges(eventData, result);
        }

        public override async ValueTask<int>
SavedChangesAsync(SaveChangesCompletedEventData eventData, int result,
CancellationToken cancellationToken = default)
        {
            await DispatchEvents(eventData.Context);

            return await base.SavedChangesAsync(eventData, result,
cancellationToken);
        }

        private async Task DispatchEvents(DbContext? context)
        {
            if (context == null)
            {
                return;
            }

            await _mediator.DispatchDomainEvents(context);
        }
    }
}
```

```
using ConferenceManager.Domain.Common;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace ConferenceManager.Infrastructure.Persistence.Configurations
{
    public abstract class BaseAuditableEntityConfiguration<TEntity> :
IEntityTypeConfiguration<TEntity> where TEntity : BaseAuditableEntity
    {
        public void Configure(EntityTypeBuilder<TEntity> builder)
        {
            builder.HasKey(x => x.Id);

            builder.Property(x => x.Id)
                .ValueGeneratedOnAdd();

            builder.Property(e => e.CreatedOn)
                .HasConversion(v => v, v => DateTime.SpecifyKind(v,
DateTimeKind.Utc));

            builder.Property(e => e.ModifiedOn)
                .HasConversion(v => v, v => DateTime.SpecifyKind(v,
DateTimeKind.Utc));

            ConfigureInner(builder);
        }

        protected abstract void ConfigureInner(EntityTypeBuilder<TEntity> builder);
    }
}
```

```
using ConferenceManager.Domain.Entities;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace ConferenceManager.Infrastructure.Persistence.Configurations
{
    public class SubmissionConfiguration :
BaseAuditableEntityConfiguration<Submission>
    {
        protected override void ConfigureInner(EntityTypeBuilder<Submission>
builder)
        {
            builder.Property(x => x.Title)
                .IsRequired()
                .HasMaxLength(100);

            builder.Property(x => x.Keywords)
                .IsRequired()
                .HasMaxLength(100);

            builder.Property(x => x.Abstract)
                .IsRequired()
                .HasMaxLength(1000);

            builder.Property(x => x.ResearchAreas)
                .IsRequired()
                .HasMaxLength(500);

            builder.Property(x => x.Status)
                .IsRequired();
        }
    }
}
```

```
using ConferenceManager.Core.Common.Interfaces;
using ConferenceManager.Domain.Entities;
using ConferenceManager.Infrastructure.Persistence;
using ConferenceManager.Infrastructure.Persistence.Interceptors;
using ConferenceManager.Infrastructure.Services;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace ConferenceManager.Infrastructure
{
    public static class ConfigureServices
    {
        public static IServiceCollection AddInfrastructureServices(this
IServiceCollection services, IConfiguration configuration)
        {
            services.AddScoped<AuditableEntitySaveChangesInterceptor>();
            services.AddScoped<ForeignKeysSaveChangesInterceptor>();
            services.AddScoped<DomainEventsSaveChangesInterceptor>();

            services.AddDbContext<ApplicationDbContext>(options =>

options.UseLazyLoadingProxies().UseSqlServer(configuration.GetConnectionString("Defa
ultConnection"),
                    builder =>
builder.MigrationsAssembly(typeof(ApplicationDbContext).Assembly.FullName)));

            services.AddScoped<IApplicationDbContext>(provider =>
provider.GetRequiredService<ApplicationDbContext>());

            services.AddScoped<ApplicationDbContextInitializer>();

            services.AddIdentityCore<ApplicationUser>()
                .AddRoles<ApplicationRole>()
                .AddEntityFrameworkStores<ApplicationDbContext>();

            services.Configure<IdentityOptions>(options =>
            {
                options.User.RequireUniqueEmail = true;
                options.Password.RequireDigit = true;
                options.Password.RequireLowercase = true;
                options.Password.RequireNonAlphanumeric = true;
                options.Password.RequireUppercase = true;
                options.Password.RequiredLength = 8;
            });

            services.AddTransient<IDateTimeService, DateTimeService>();

            return services;
        }
    }
}
```

```
using ConferenceManager.Core.Common.Interfaces;
using MediatR;

namespace ConferenceManager.Core.Common
{
    public abstract class DbContextRequestHandler<TRequest, TResponse> :
IRequestHandler<TRequest, TResponse> where TRequest : IRequest<TResponse>
    {
        protected IApplicationDbContext Context { get; }
        protected ICurrentUserService CurrentUser { get; }
        protected IMappingHost Mapper { get; }

        protected DbContextRequestHandler(IApplicationDbContext context,
ICurrentUserService currentUser, IMappingHost mapper)
        {
            Context = context;
            CurrentUser = currentUser;
            Mapper = mapper;
        }

        public abstract Task<TResponse> Handle(TRequest request, CancellationToken
cancellationToken);
    }

    public abstract class DbContextRequestHandler<TRequest> :
IRequestHandler<TRequest> where TRequest : IRequest
    {
        protected IApplicationDbContext Context { get; }
        protected ICurrentUserService CurrentUser { get; }
        protected IMappingHost Mapper { get; }

        protected DbContextRequestHandler(IApplicationDbContext context,
ICurrentUserService currentUser, IMappingHost mapper)
        {
            Context = context;
            CurrentUser = currentUser;
            Mapper = mapper;
        }

        public abstract Task Handle(TRequest request, CancellationToken
cancellationToken);
    }
}
```

```
using ConferenceManager.Core.Common.Interfaces;
using MediatR.Pipeline;
using Microsoft.Extensions.Logging;

namespace CleanArchitecture.Application.Common.Behaviors
{
    public class LoggingBehavior<TRequest> : IRequestPreProcessor<TRequest> where
TRequest : notnull
    {
        private readonly ILogger _logger;
        private readonly ICurrentUserService _currentUser;

        public LoggingBehavior(ILogger<TRequest> logger, ICurrentUserService
currentUserService)
        {
            _logger = logger;
            _currentUser = currentUserService;
        }

        public Task Process(TRequest request, CancellationToken cancellationToken)
        {
            _logger.LogInformation($"Initiated {typeof(TRequest).Name} by user id
{_currentUser.Id}");

            return Task.CompletedTask;
        }
    }
}
```

```
using FluentValidation;
using MediatR;
using ValidationException =
ConferenceManager.Core.Common.Exceptions.ValidationException;

namespace CleanArchitecture.Application.Common.Behaviors
{
    public class ValidationBehavior<TRequest, TResponse> :
IPipelineBehavior<TRequest, TResponse> where TRequest : notnull
    {
        private readonly IEnumerable<IValidator<TRequest>> _validators;

        public ValidationBehavior(IEnumerable<IValidator<TRequest>> validators)
        {
            _validators = validators;
        }

        public async Task<TResponse> Handle(TRequest request,
RequestHandlerDelegate<TResponse> next, CancellationToken cancellationToken)
        {
            if (_validators.Any())
            {
                var context = new ValidationContext<TRequest>(request);

                var validationResults = await Task.WhenAll(
                    _validators.Select(validator => validator.ValidateAsync(context,
cancellationToken)));

                var failures = validationResults
                    .Where(r => r.Errors.Any())
                    .SelectMany(r => r.Errors)
                    .ToList();

                var exceptionFailures = failures.Where(f => f.CustomState is
Exception);

                if (exceptionFailures.Any())
                {
                    throw (Exception)exceptionFailures.First().CustomState;
                }

                if (failures.Any())
                {
                    throw new ValidationException(failures);
                }
            }

            return await next();
        }
    }
}
```

```
using MediatR;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.DependencyInjection;
using Swashbuckle.AspNetCore.Annotations;

namespace ConferenceManager.Api.Abstract
{
    [ApiController]
    [ApiExceptionFilter]
    [Produces("application/json")]
    [SwaggerResponse(StatusCodes.Status400BadRequest, Type =
typeof(ValidationProblemDetails))]
    [SwaggerResponse(StatusCodes.Status403Forbidden, Type = typeof(ProblemDetails))]
    [SwaggerResponse(StatusCodes.Status404NotFound, Type = typeof(ProblemDetails))]
    [Route("api/[controller]")]
    public class ApiControllerBase : ControllerBase
    {
        private ISender? _mediator;

        protected ISender Mediator => _mediator ??=
HttpContext.RequestServices.GetRequiredService<ISender>();
    }
}
```

```
namespace ConferenceManager.Api.Controllers
{
    [Route("api/[controller]")]
    public class UserController : ApiControllerBase
    {
        [HttpPost]
        [Route("register")]
        [SwaggerResponse(StatusCodes.Status200OK, Type = typeof(TokenResponse))]
        public async Task<IActionResult> Register(RegisterUserCommand command,
CancellationToken cancellation)
        {
            var token = await Mediator.Send(command, cancellation);

            return Ok(token);
        }

        [HttpPost]
        [Route("login")]
        [SwaggerResponse(StatusCodes.Status200OK, Type = typeof(TokenResponse))]
        public async Task<IActionResult> Login(LoginUserCommand command,
CancellationToken cancellation)
        {
            var token = await Mediator.Send(command, cancellation);

            return Ok(token);
        }
    }
}
```

```
public AuthResponse GenerateToken(ApplicationUser user)
{
    byte[] key = Encoding.ASCII.GetBytes(_settings.Key);

    var descriptor = new SecurityTokenDescriptor
    {
        Expires = _dateTime.Now.AddMinutes(_settings.ExpiresMinutes),
        SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key),
SecurityAlgorithms.HmacSha256Signature),
        Subject = new ClaimsIdentity(new Claim[]
        {
            new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
        })
    };

    var roles = user.ConferenceRoles
        ?.Select(r => new { r.ConferenceId, Role = r.Role.Name! })
        ?.GroupBy(r => r.ConferenceId)
        ?.ToDictionary(key => key.Key, value => value.Select(v => v.Role).ToArray())
        ?? new Dictionary<int, string[]>();

    var participations = user.ConferenceParticipations
        ?.Select(part => part.Id)
        ?.ToArray() ?? Array.Empty<int>();

    var uniqueRoles = roles.SelectMany(r => r.Value).Distinct();

    foreach (var role in uniqueRoles)
        descriptor.Subject.AddClaim(new Claim(ClaimTypes.Role, role));


    if (user.IsAdmin)
        descriptor.Subject.AddClaim(new Claim(ClaimTypes.Role,
ApplicationRole.Admin));


    descriptor.Subject.AddClaim(new Claim(Claims.ConferenceRoles,
JsonConvert.SerializeObject(roles)));
    descriptor.Subject.AddClaim(new Claim(Claims.ConferenceParticipations,
JsonConvert.SerializeObject(participations)));

    var handler = new JwtSecurityTokenHandler();
    var token = handler.CreateToken(descriptor);
    var tokenValue = handler.WriteToken(token);

    _httpContextAccessor.HttpContext?.Response.Cookies.Append(Cookies.Token,
tokenValue, new CookieOptions()
    {
        MaxAge = TimeSpan.FromMinutes(_settings.ExpiresMinutes),
        IsEssential = true,
    });

    return new AuthResponse()
    {
        Id = user.Id,
        Roles = roles,
        Admin = user.IsAdmin,
        ValidTo = token.ValidTo,
    };
}
```

```
using ConferenceManager.Api.Shared.Util;
using Microsoft.AspNetCore.Http;

namespace ConferenceManager.Api.Shared.Middleware
{
    public class JwtCookieMiddleware : IMiddleware
    {
        public async Task InvokeAsync(HttpContext context, RequestDelegate next)
        {
            var token = context.Request.Cookies[Cookies.Token];

            if (!string.IsNullOrEmpty(token))
            {
                context.Request.Headers.Authorization = $"Bearer {token}";
            }

            context.Response.Headers.Add("X-Content-Type-Options", "nosniff");
            context.Response.Headers.Add("X-Xss-Protection", "1");
            context.Response.Headers.Add("X-Frame-Options", "DENY");

            await next(context);
        }
    }
}
```

```
public class ConferenceAuthorizationFilter : IAuthorizationFilter
{
    private readonly IApplicationDbContext _dbContext;
    private readonly ICurrentUserService _currentUser;
    private readonly string[] _roles;

    public ConferenceAuthorizationFilter(
        IApplicationDbContext dbContext,
        ICurrentUserService currentUser,
        string[] roles)
    {
        _dbContext = dbContext;
        _currentUser = currentUser;
        _roles = roles;
    }

    public void OnAuthorization(AuthorizationFilterContext context)
    {
        var hasHeader =
context.HttpContext.Request.Headers.TryGetValue(Headers.ConferenceId, out var
header);

        if (!hasHeader)
        {
            SetMissingHeaderResult(context);
            return;
        }

        if (!int.TryParse(header.FirstOrDefault(), out var conferenceId);)
        {
            SetInvalidHeaderResult(context);
            return;
        }

        var conference = _dbContext.Conferences.AsNoTracking()
            .FirstOrDefault(c => c.Id == conferenceId);

        if (conference == null)
        {
            SetConferenceNotFoundResult(context);
            return;
        }

        if (_currentUser.IsAdmin)
            return;

        if (!_currentUser.Participations.Contains(conferenceId))
        {
            SetNotPartOfConferenceResult(context);
            return;
        }

        var userConferenceRoles = _currentUser.Roles
            .Where(kv => kv.Key == conferenceId)
            .SelectMany(kv => kv.Value);

        bool hasRole = userConferenceRoles.Any(role => _roles.Contains(role));
        if (!hasRole)
           SetInsufficientPermissionsResult(context);
     }}
```

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.EntityFrameworkCore.Database.Command": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DefaultConnection": "Server=DB_SERVER;Database=ConferenceManager;User
ID=DB_USER;Password=DB_PASSWORD;Trusted_Connection=False;TrustServerCertificate=True
;MultipleActiveResultSets=true"
  },
  "TokenSettings": {
    "Issuer": "TOKEN_ISSUER",
    "Audience": "TOKEN_AUDIENCE",
    "Key": "TOKEN_KEY",
    "ExpiresMinutes": 2880
  },
  "SeedSettings": {
    "AdminPassword": "ADMIN_PASSWORD",
    "AdminEmail": "ADMIN_EMAIL"
  },
  "CorsSettings": {
    "FrontUrl": "FRONT_URL"
  }
}
```

```
export const LoginForm: React.FC<{}> = () => {
  const { response, post } = usePostLoginApi();
  const navigate = useNavigate();

  useEffect(() => {
    if (Auth.isAuthed()) {
      navigate("/");
    }
  }, [navigate]);

  useEffect(() => {
    if (response.data) {
      Auth.login(response.data);
      navigate("/");
    }
  }, [response, navigate]);

  const formik = useFormik({
    initialValues: {
      email: "",
      password: "",
    },
    validationSchema: validationSchema,
    onSubmit: (values) => {
      post(values);
    },
  });

  return (
    <form onSubmit={formik.handleSubmit}>
      <TextField
        name="email"
        label="Email"
        value={formik.values.email}
        onChange={formik.handleChange}
        error={formik.touched.email && Boolean(formik.errors.email)}
        helperText={formik.touched.email && formik.errors.email}
      />
      <TextField
        name="password"
        label="Password"
        type="password"
        value={formik.values.password}
        onChange={formik.handleChange}
        error={formik.touched.password && Boolean(formik.errors.password)}
        helperText={formik.touched.password && formik.errors.password}
      />
      <FormErrorAlert response={response} />
      <Button
        disabled={response.status === "loading"}
        color="primary"
        variant="contained"
        type="submit">
        Submit
      </Button>
    </form>
  );
};
```

```
FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS base
WORKDIR /app
ENV ASPNETCORE_URLS=http://+:8001

FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build
WORKDIR /src
ARG DB_SERVER
ARG DB_USER
ARG DB_PASSWORD
ARG TOKEN_ISSUER
ARG TOKEN_AUDIENCE
ARG TOKEN_KEY
ARG ADMIN_PASSWORD
ARG ADMIN_EMAIL
ARG FRONT_URL
COPY ConferenceManager.sln ./ConferenceManager.sln
COPY ConferenceManager.Api.User/*.csproj ./ConferenceManager.Api.User/
COPY ConferenceManager.Api.Submission/*.csproj ./ConferenceManager.Api.Submission/
COPY ConferenceManager.Api.Conference/*.csproj ./ConferenceManager.Api.Conference/
COPY ConferenceManager.Api.Shared/*.csproj ./ConferenceManager.Api.Shared/
COPY ConferenceManager.Core/*.csproj ./ConferenceManager.Core/
COPY ConferenceManager.Domain/*.csproj ./ConferenceManager.Domain/
COPY ConferenceManager.Infrastructure/*.csproj ./ConferenceManager.Infrastructure/
RUN dotnet restore
COPY . .
WORKDIR /src/ConferenceManager.Api.User
RUN sed -i "s;DB_SERVER;${DB_SERVER};" appsettings.json
RUN sed -i "s;DB_USER;${DB_USER};" appsettings.json
RUN sed -i "s;DB_PASSWORD;${DB_PASSWORD};" appsettings.json
RUN sed -i "s;TOKEN_ISSUER;${TOKEN_ISSUER};" appsettings.json
RUN sed -i "s;TOKEN_AUDIENCE;${TOKEN_AUDIENCE};" appsettings.json
RUN sed -i "s;TOKEN_KEY;${TOKEN_KEY};" appsettings.json
RUN sed -i "s;ADMIN_PASSWORD;${ADMIN_PASSWORD};" appsettings.json
RUN sed -i "s;ADMIN_EMAIL;${ADMIN_EMAIL};" appsettings.json
RUN sed -i "s;FRONT_URL;${FRONT_URL};" appsettings.json
RUN dotnet build "ConferenceManager.Api.User.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "ConferenceManager.Api.User.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "ConferenceManager.Api.User.dll"]
```

```
FROM node:alpine as build
ARG VITE_USER_API_URL
ARG VITE_CONFERENCE_API_URL
ARG VITE_SUBMISSION_API_URL
WORKDIR /app
COPY . .
RUN rm .env
RUN echo "VITE_USER_API_URL=${VITE_USER_API_URL}" >> .env
RUN echo "VITE_CONFERENCE_API_URL=${VITE_CONFERENCE_API_URL}" >> .env
RUN echo "VITE_SUBMISSION_API_URL=${VITE_SUBMISSION_API_URL}" >> .env
RUN npm install
RUN npm run build

FROM nginx:stable-alpine as final
COPY --from=build /app/dist /usr/share/nginx/html
COPY --from=build /app/nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

```yaml
version: "3.8"
services:
  ms-front:
    container_name: ms-front
    build:
      context: ./conference
      args:
        VITE_USER_API_URL: ${VITE_USER_API_URL}
        VITE_CONFERENCE_API_URL: ${VITE_CONFERENCE_API_URL}
        VITE_SUBMISSION_API_URL: ${VITE_SUBMISSION_API_URL}
    ports:
      - 80:80
    networks:
      - conference-net
    environment:
      - NODE_ENV=production
    restart: on-failure:2
  user-service:
    container_name: user-service
    build:
      context: ./ConferenceManager/
      dockerfile: Dockerfile.user
      args:
        DB_SERVER: ${DB_SERVER}
        DB_USER: ${DB_USER}
        DB_PASSWORD: ${MSSQL_SA_PASSWORD}
        TOKEN_ISSUER: ${TOKEN_ISSUER}
        TOKEN_AUDIENCE: ${TOKEN_AUDIENCE}
        TOKEN_KEY: ${TOKEN_KEY}
        ADMIN_EMAIL: ${ADMIN_EMAIL}
        ADMIN_PASSWORD: ${ADMIN_PASSWORD}
        FRONT_URL: ${TOKEN_AUDIENCE}
    depends_on:
      - db
    ports:
      - 8001:8001
    networks:
      - conference-net
    restart: on-failure:2
  conference-service:
    container_name: conference-service
    build:
      context: ./ConferenceManager/
      dockerfile: Dockerfile.conference
      args:
        DB_SERVER: ${DB_SERVER}
        DB_USER: ${DB_USER}
        DB_PASSWORD: ${MSSQL_SA_PASSWORD}
        TOKEN_ISSUER: ${TOKEN_ISSUER}
        TOKEN_AUDIENCE: ${TOKEN_AUDIENCE}
        TOKEN_KEY: ${TOKEN_KEY}
        ADMIN_EMAIL: ${ADMIN_EMAIL}
        ADMIN_PASSWORD: ${ADMIN_PASSWORD}
        FRONT_URL: ${TOKEN_AUDIENCE}
    depends_on:
      - db
    ports:
      - 8002:8002
    networks:
      - conference-net
    restart: on-failure:2
  submission-service:
    container_name: submission-service
```

```
    build:
      context: ./ConferenceManager/
      dockerfile: Dockerfile.submission
      args:
        DB_SERVER: ${DB_SERVER}
        DB_USER: ${DB_USER}
        DB_PASSWORD: ${MSSQL_SA_PASSWORD}
        TOKEN_ISSUER: ${TOKEN_ISSUER}
        TOKEN_AUDIENCE: ${TOKEN_AUDIENCE}
        TOKEN_KEY: ${TOKEN_KEY}
        ADMIN_EMAIL: ${ADMIN_EMAIL}
        ADMIN_PASSWORD: ${ADMIN_PASSWORD}
        FRONT_URL: ${TOKEN_AUDIENCE}
    depends_on:
      - db
    ports:
      - 8003:8003
    networks:
      - conference-net
    restart: on-failure:2
  db:
    container_name: db
    image: mcr.microsoft.com/mssql/server:2022-latest
    ports:
      - 1433:1433
    networks:
      - conference-net
    volumes:
      - db-data-vol:/var/opt/mssql
      - db-backup-vol:/backup
    user: root
    environment:
      - ACCEPT_EULA=Y
      - MSSQL_SA_PASSWORD=${MSSQL_SA_PASSWORD}
      - MSSQL_PID=Express
    restart: on-failure:2
  backup:
    container_name: backup
    image: bbtsoftwareag/mssql-backup
    depends_on:
      - db
    volumes:
      - db-backup-vol:/backup
    environment:
      - TZ=Europe/Kiev
      - DB_SERVER=${DB_SERVER}
      - DB_USER=${DB_USER}
      - DB_PASSWORD=${MSSQL_SA_PASSWORD}
      - DB_NAMES=ConferenceManager
      - BACKUP_CLEANUP=true
      - BACKUP_AGE=10
      - CRON_SCHEDULE=0 0 * * *
    networks:
      - conference-net
volumes:
  db-data-vol:
  db-backup-vol:
networks:
  conference-net:
```