Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

РОЗРОБКА ВЕБ-ЗАСТОСУНКІВ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ НА ОСНОВІ RESTFUL API

Текстова частина до курсової роботи за спеціальністю «Комп'ютерні науки» - 122

Керівник курсової роботи
к.т.н., доц. *Олецький О. В.*
_____
(підпис)
"_—_" _____ 2021 р.

Виконав студент *Волошко М. В.*
"_—_" _____ 2021 р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики,
д.т.н, доцент
_____А. М. Глибовець
(підпис)
„____”_____2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту_____ факультету _____ _____курсу
ТЕМА _____

Зміст ТЧ до курсової роботи:

1. Індивідуальне завдання
2. Календарний план
3. Анотація
4. Вступ
5. Аналіз поняття мікросервісної архітектури
6. Важливі складові мікросервісної архітектури
7. Опис технологій та розробка застосунку
8. Висновки
9. Список використаних джерел

Додатки (за необхідністю)

Дата видачі „___” _____ 2021 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Календарний план виконання курсової роботи

Тема: Розробка веб-застосунків з мікросервісною архітектурою на основі RESTful API

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	15.11.2020	
2.	Огляд технічної літератури за темою роботи.	5.12.2020	
3.	Дослідження поняття мікросервісної архітектури	25.12.2020	
3.	Аналіз принципів та підходів до побудови мікросервісної архітектури	10.01.2021	
4.	Аналіз доступних технологій та бібліотек для виконання завдання	25.01.2021	
5.	Створення моделі застосунку	15.02.2021	
6.	Виконання практичної частини курсової роботи	15.04.2021	
7.	Написання текстової частини курсової роботи	5.05.2021	
8.	Створення слайдів для доповіді та написання доповіді.	6.05.2021	
8.	Аналіз отриманих результатів з керівником, написання доповіді та попередній захист курсової роботи.	10.05.2021	
10.	Корегування роботи за результатами попереднього захисту.	15.05.2021	
11.	Остаточне оформлення пояснювальної роботи та слайдів.	17.05.2021	
12.	Захист курсової роботи	24.05.2021	

Студент *Волошко М. В.*

Керівник *Олецький О. В.*

“_____” _____

ЗМІСТ

Анотація

У цій роботі виконано дослідження мікросервісної архітектури, її особливостей, переваг та недоліків у порівнянні з монолітної архітектурою. Описані методи міграції від монолітної архітектури до мікросервісів. Розглянуто механізм комунікації мікросервісів та проаналізовано REST архітектуру. Наведено перелік використаних технологій та їх опис. Розроблено веб-додаток з мікросервісною архітектурою на основі RESTful API.

Вступ

Популярність мікросервісної архітектури серед розробників програмного забезпечення стрімко зростала з моменту її виникнення, в результаті чого у 2021 році вона досягла рівня одного із найбільш широко вживаних стилів проектування. Являючись альтернативою традиційному підходу монолітної архітектури, вона надає змогу розв’язувати класичні проблеми, які виникають внаслідок розробки монолітного застосунку.

Однак впровадження мікросервісної архітектури несе в собі певні випробування, оскільки її реалізація включає в себе додаткові складнощі при розгортанні застосунку на етапі виробництва, налаштовуванні автоматизованого тестування, а також вимагає більших зусиль при розробці та координуванні мікросервісів. Ключовим аспектом мікросервісної архітектури є комунікація між мікросервісами і має два основних напрямки –синхронний та асинхронний – та різними способами їх реалізації – REST та віддалений виклик процедури для синхронного методу і керованість повідомленнями та керованість подіями для асинхронного.

Завдяки популярності мікросервісної архітектури, широкому вибору способів її реалізації та поширеності використання методу комунікації за допомогою REST за мету даної роботи були поставлені розробка веб-застосунку з мікросервісною архітектурою на основі REST, визначення особливостей REST та розгляд інших методів комунікації між мікросервісами.

Перший розділ присвячено аналізу мікросервісної архітектури як одного з найбільш популярних сучасних рішень, його характеристика, переваги та недоліки перед монолітною архітектурою, а також стратегії переходу від монолітної архітектури до мікросервісної.

У другому розділі було досліджено важливі складові мікросервісної архітектури, такі як API шлюз, реєстр сервісів та сервісне виявлення, а також механізм комунікації між мікросервісами та різні методи його реалізації.

Третій розділ було присвячено опису використаних технологій, бібліотек та принципів дизайну програмного забезпечення, а також їх застосування при розробці веб-застосунку з мікросервісною архітектурою на основі REST.

Створено програмний продукт із мікросервісною архітектурою на основі REST, який призначений для навчальних цілей, але здатний до застосування у реальних бізнес-задачах у майбутньому.

Постановка задачі

1. Провести аналіз поняття мікросервісної архітектури, її характеристики, особливості та недоліки перед монолітною архітектурою, стратегії переходу від монолітної архітектури до мікросервісної.

2. Дослідити важливі складові мікросервісної архітектури, їх характеристики та призначення. Розглянути різновиди механізму комунікації між мікросервісами.

3. Проаналізувати особливості обраних технологій, бібліотек та принципів дизайну програмного забезпечення. Розробити веб-застосунок з мікросервісною архітектурою на основі REST.

Перелік прийнятих скорочень

REST – репрезентативна передача стану

API – програмний інтерфейс прикладної програми

HTTP – протокол передачі гіпертекстових документів

AMQP – розширений протокол черги повідомлень

MERN – стек технологій

SPA – застосунок єдиної сторінки

SQL – структурована мова запитів

NoSQL – не лише структурована мова запитів

DOM – об’єктна модель документа

JWT – JSON веб-токен

ODM – моделювання даних об’єктів

РОЗДІЛ 1: АНАЛІЗ ПОНЯТТЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

1.1 Поняття мікросервісної архітектури

На початок 2010-х років стрімкий розвиток і поширення мережевих хмарних сервісів призвели до розчарування в класичному, так званому монолітному способу побудови архітектури застосунків. Через складність окремих модулів, які зазвичай призводили до побудови цілих програмних систем, а також через необхідність забезпечувати сумісність між ними за допомогою стандартних протоколів, внесення будь-яких змін і доповнень стало нетривіальною задачею, що віднімало занадто багато часу і ресурсів.

Як відповідь на це було запропоновано мікросервісну архітектуру як розподілену систему простих і легко замінних модулів, що виконують по можливості одну елементарну функцію. При цьому мікросервісна система має симетричну, однорангову, а не ієрархічну структуру, що знімає необхідність у складній організації взаємозв'язків між модулями. Сервіси зв'язуються між собою і клієнтами за допомогою використання легких протоколів. В результаті створюється система, проста в розгортанні і модернізації з функціями автоматичної розробки та оновлення.

Немає точного визначення мікросервісної архітектури, але мікросервісом може бути будь-який операційний блок або підрозділ, який дуже ефективно справляється зі своєю єдиною відповідальністю. Мікросервіси – це різновид сервіс-орієнтовної архітектури і слугують для побудови автономних, самопідтримуваних та слабопов’язаних модулів, які разом утворюють цілу систему. Отже, мікросервісну архітектуру можна визначити як метод побудови архітектури програмного застосунку, що являє собою систему, розподілену на взаємопов’язані мікросервіси, кожен з яких виконує певну бізнес-функцію.

1.2 Характеристики мікросервісної архітектури

Мартін Фаулер у своїй статті [1] зазначав, що «попри відсутність точного визначення мікросервісної архітектури, існують певні характеристики, за допомогою яких можна визначити чи є архітектура програмного застосунку мікросервісною».

1.2.1 Розподіленість системи за бізнес-функціями

Мікросервісна архітектура буде використовувати бібліотеки, але розподілення системи головним чином відбувається за допомогою розбиття на окремі сервіси. Бібліотеки, будучи прив’язаними до програми і викликаними за допомогою викликів функцій у пам’яті, відрізняються від сервісів тим, що сервіси - це компоненти, що існують незалежно і взаємодіють за допомогою такого механізму, як запит веб-служби або виклик віддаленої процедури.

Однією з головних причин використання сервісів у якості компонентів, а не бібліотек, полягає у тому, що розгортання сервісів відбувається незалежно. Якщо застосунок використовує декілька бібліотек в одному процесі, то будь-які зміни вимагатимуть чергового розгортання усього застосунку, але якщо застосунок розподілений на декілька сервісів, то зазвичай зміни будуть ізольовані всередині конкретного сервісу, не впливаючи на інші.

Черговою перевагою використання сервісів як компонентів полягаю у визначенні більш чіткого інтерфейсу. Більшість мов програмування не мають належного механізму для визначення явного публічного інтерфейсу, що призводить до порушення інкапсуляції компонента, в той час як сервіси допомагають цього уникнути, надаючи змогу використовувати механізми віддаленого виклику.

1.2.2 Організованість навколо бізнес-можливостей

Підхід мікросервісної архітектури полягає у розподіленні системи на сервіси, організовані навколо бізнес-можливостей, і передбачають широке впровадження програмного забезпечення для конкретної галузі бізнесу, включаючи інтерфейс користувача, постійного сховища та будь-які інші зовнішні послуги.

1.2.3 Розумні кінцеві точки та прості (dumb) канали передачі даних

Альтернативні підходи до побудови архітектури зазвичай вкладають надмірну логіку у механізм комунікації при його створенні. Хорошим прикладом цього є шина обслуговування сервісів, де продукти часто містять складні засоби для маршрутизації повідомлень, оркестровці, трансформації та застосування бізнес-правил. Додатки, побудовані на основі мікросервісів, прагнуть бути максимально розподіленими - вони володіють власною логікою домену та виконують роль, отримуючи запит, застосовуючи за необхідністю логіку та в результаті продукуючи відповідь.

У мікросервісній архітектурі найбільш вживаними способами комунікації є HTTP протоколи з API та полегшені повідомлення. Перший спосіб являє собою метод комунікації “запит-відповідь”, який полягає у надсиланні запиту до служби, яка у подальшому його оброблює та надсилає відповідь. Часто використані ресурси можуть бути легко занесені до кешу. Другий підхід часто використовується у ролі шини полегшених повідомлень. Така інфраструктура є зазвичай німою у ролі маршрутизатора повідомлень – прості імплементації на зразок RabbitMQ чи ZeroMQ забезпечують не більше, ніж надійну асинхронну структуру, в той час як вся бізнес-логіка знаходиться на кінцевих точках сервісів, яка полягає у відсиланні повідомлень, їх обробці та надсиланні повідомлень.

1.2.4 Децентралізоване управління даними

Існує вірогідність того, що концептуальна модель світу буде відрізнятися поміж систем. Наприклад, деякі сутності, які у сервісі продажів є клієнтами, можуть взагалі бути відсутніми у сервісі підтримки. Іноді атрибути наявних сутностей можуть приймати різний вигляд або навіть містити різну семантику.

Паралельно з розподіленням системи на сервіси, керуючись бізнес-можливостями, також децентралізується система управління даними. Мікросервісна архітектура зазвичай використовує підхід, який полягає у наданні кожному мікросервісу своєї власної бази даних, яка може бути або екземпляром уже використовуваної технології бази даних, або цілком іншою системою баз даних.

Проте розподілення відповідальності за збереження та маніпулювання даними має певні наслідки на проведенні оновлень. Ця проблема зазвичай у монолітній архітектурі вирішується за допомогою транзакцій, забезпечуючи послідовність при оновленні кількох ресурсів.

Проте згідно з Мартіном Фаулером [1], «використання транзакцій таким чином допомагає забезпечити послідовність, але накладає значний тимчасовий взаємозв'язок, що є проблематичним для багатьох сервісів. Розподілені транзакції, як відомо, важко реалізувати, і як наслідок, архітектури мікросервісів акцентують на координації між сервісами без транзакцій, з чітким визнанням того, що узгодженість може бути лише можливою послідовністю, а проблеми вирішуються компенсуючими операціями».

1.2.5 Проектування на збій

При розподіленні системи на мікросервіси, виникає необхідність у витримуванні застосунками збоїв у мікросервісах. Будь-який мікросервіс може дати збій з різноманітних причин, але клієнтська частина застосунку повинна надавати контрольовану відповідь на таку ситуацію.

Оскільки збій може статися у будь-який момент, важливою практикою у розробці мікросервісної архітектури є прикладення великих зусиль на розробку моніторингової системи у реальному часі, яка матиме змогу перевіряти як архітектурні елементи, наприклад, кількість надісланих до бази даних запитів в секунду, так і бізнес-орієнтовну аналітику, наприклад, кількість зроблених замовлень у хвилину.

Як зазначав Мартін Фаулер у своїй статті [1], «це особливо важливо для архітектури мікросервісів, оскільки їхня перевага у хореографії та співпраці на основі подій веде до виникнення непередбачуваної поведінки. Хоча багато експертів хвалять цінність випадкової несподіваності, правда полягає в тому, що нова поведінка іноді може бути небажаною. Моніторинг життєво важливий для того, щоб швидко виявляти небажану поведінку, надаючи змогу її виправити».

1.3 Переваги та недоліки перед монолітною архітектурою

Мікросервісна архітектура дозволяє вирішити типові проблеми монолітної архітектури, таких як нагромадженість системи, важка зрозумілість структури застосунку, масштабування, внесення змін до наявного функціоналу та прив’язаність застосунку лише до одного набору технологій.

Розглядаючи переваги мікросервісної архітектури, по-перше, варто зазначити, що за допомогою мікросервісів вирішується проблема складності та нагромадженості системи – кожна окрема бізнес-функція буде поміщена у

відповідний сервіс, ізолюючи весь функціонал та семантику в межах мікросервісу, а також полегшуючи зрозумілість структури всього застосунку.

Завдяки розподіленості та ізольованості, мікросервісна архітектура дозволяє працювати над кожним сервісом окремій команді та зменшує необхідний поріг для нових членів команди, оскільки новим розробникам потрібно буде розбиратися лише у семантиці та домені певного сервісу, а не всього застосунку, як це відбувається у монолітній архітектурі.

Децентралізованість системи також дозволяє команді для кожного мікросервісу обирати найбільш зручний та ефективний стек технологій, турбуючись лише про забезпечення відповідності у протоколі комунікації та контракту за API. Згадуючи характеристики мікросервісної архітектури, які полягають у проектуванні на збій та ізольованості сервісів, розподіленість системи дозволяє запобігти каскадності збоїв мікросервісів, замикаючи їх всередині сервісу та надаючи контрольовану поведінку-реакцію на збій.

Мікросервісна архітектура вирішує проблему масштабування застосунку, наявну у монолітному підході. Кожен мікросервіс може бути розширеним, вдосконаленим чи зміненим, незалежно від інших сервісів, що робить весь процес масштабування більш ефективним з-боку затратності по часу чи ресурсам, в той час як моноліт змушує масштабувати увесь застосунку, незалежно від того, чи це є доцільно.

Проте впровадження мікросервісної архітектури тягне за собою певні недоліки та аспекти, на які слід звернути увагу.

Один з найбільших недоліків мікросервісної архітектури полягає в організації та оркестровці мікросервісів. При розробці застосунку розробники змушені детально спроектувати та якісно розробити систему комунікації між мікросервісами. Також при створенні нової команди розробників виникає додаткова складність в наданні ізольованого середовища для роботи таким чином, щоб результати їх роботи були інтегровані в систему без тісної зв’язності з іншими командами.

Черговим недоліком у мікросервісній архітектури виступає розподіленість бази даних. Завдяки єдності бази даних, монолітний підхід дозволяє легко використовувати транзакції для оновлення декількох ресурсів, в той час як мікросервіси вимагають власноруч розробляти альтернативний підхід, який полягає у забезпеченні послідовності оновлення, що вимагає більше зусиль від розробників.

Створення якісного середовища для застосунку з мікросервісною архітектурою вимагає відповідної команди, а також забезпеченої від збоїв масштабованої інфраструктури. Додатковим завданням для розробників виступає керування контейнерами та екземплярами мікросервісів, забезпечення масштабованості та високої доступності застосунку, впровадження автоматизації та перехід до підходящого хмарного провайдера.

Мікросервіси із залежностями, будучи повністю ізьолованими, ставлять перед розробниками тяжку задачу, яка полягає у їх тестуванні. При інтегруванні будь-якого нового сервісу до системи, сервіс повинен бути правильно налаштований та повністю протестований, щоб він не став точкою збою у системі. Декілька рівнів тестування повинні бути впровадженні, починаючи від перевірки кешування, логування та бази даних, продовжуючи тестуванням функціоналу, включно із протоколом, який буде використовувати сервіс, і колаборативним тестуванням та закінчуючи перевіркою масштабування і безпечної відмови сервісу.

У розподіленій системі черговим випробуванням слугує механізм знаходження сервісів, оскільки мікросервіси передбачають незалежність команд. Для такого механізму також важливо забезпечити динамічне місцезнаходження сервісів, оскільки воно може змінюватися досить часто впродовж життєвого циклу застосунку, а також він має розрізняти сервіси, у яких стався збій або їхня продуктивність упала.

1.4 Перехід від монолітної архітектури до мікросервісної

Кріс Річардсон у своїй статті [2] писав, що «процес трансформації монолітного застосунку у мікросервісний є формою модернізації застосунку. Це те, чим розробники займалися протягом десятиліть. Як результат, існують певні ідеї, які ми можемо використати при рефакторингу застосунку у мікросервіси».

Існує стратегія переходу від монолітної архітектури до мікросервісної, яка полягає у повній переробці існуючого застосунку, тобто створення застосунку з нуля, але із застосуванням мікросервісів. Але такий підхід не є рекомендованим, оскільки скоріш за все призведе до невдачі.

Замість повної переробки існуючого застосунку, слід використовувати поступовий перехід від моноліту до мікросервісів. Додавання нового функціоналу або розширення вже існуючого, слід виконувати у формі мікросервісів, поступово змінюючи монолітну частину застосунку та підтримуючи сумісну роботу мікросервісів та моноліту.

1.4.1 Перша стратегія

Перша стратегія як раз полягає у тому, що на момент, коли моноліт стає важкокерованим, слід зупинитися розвивати його ще більше, роблячи його ще складнішим, а новий функціонал додати у формі окремого мікросервісу. Поміж моноліта та новоствореного мікросервісу у застосунку має бути присутній також маршрутизатор запитів, який прийматиме вхідні запити та надсилаючи їх до відповідного компоненту – запити, стосовні нової функціональності будуть надсилатися мікросервісу, а застарілі запити будуть прийняті монолітом. Також застосунок повинен мати компонент-зв’язник, який інтегруватиме новий сервіс у моноліт, оскільки перший зазвичай буде потребувати мати доступ до даних моноліту. Існує три способи для сервісу це отримати: робити виклики віддаленого API, який надаватиме моноліт,

напряму робити запити до бази даних моноліту або мати власну копію даних, підтримуючи синхронізацію із монолітом. Така стратегія дозволяє уникнути подальших ускладнень моноліту, створюючи мікросервіс, який може незалежно від монолітної частини розвиватися, масштабуватися та розгортатися, проте це не є вирішенням завдання переходу від моноліту до мікросервісів, тому існують іще дві стратегії.

1.4.2 Друга стратегія

Зазвичай існує щонайменше, як три типи компонентів у корпоративному застосунку: презентаційний шар, шар бізнес-логіки та шар доступу до даних. Друга стратегія переходу від монолітної архітектури до мікросервісної полягає в відокремленні презентаційного шару в окремий застосунок. В результаті такого розподілу вийде два сервіси: один з них буде складений з компонентів бізнес-логіки та доступу до даних, поширюючи власний API, до якого буде звертатися інший сервіс, який складатиметься з презентаційного шару і буде надсилати запити на отримання даних з першого сервісу. Такий підхід дасть змогу розділити застосунок на два мікросервіси, кожен з яких можна розробляти, масштабувати та розгортати незалежно один від одного, а також дозволяє пришвидшити роботу розробників інтерфейсу користувача та полегшити застосування тестування. Також згідно з цією стратегією, створюється віддалений API, до якого у подальшому можна буде звертатися з будь-якого іншого створеного мікросервісу. Але у подальшому ці два розділені сервіси переростуть у два важкокерованих моноліта, тому для вирішення цієї проблеми існує третя стратегія, яка допоможе додатково провести розподіл кожного з монолітів.

1.4.3 Третя стратегія

Третя стратегія переходу полягає у перетворенні існуючих модулів у моноліті в окремі мікросервіси. Кожного разу, коли ви витягуєте модуль і перетворюєте його на послугу, моноліт зменшується. Після розподілення достатньої кількості модулів моноліт перестане бути нагромадженим або ж може повністю зникнути чи зменшитись настільки, що перетвориться у черговий мікросервіс.

Складний та нагромаджений монолітний застосунок складається з великої кількості модулів, тому визначення порядку розподілення модулів часто є складним завданням. Хорошим підходом слугує витягання модулів, які можна легко переробити у мікросервіси. Це забезпечить розробника певним досвідом розподілення моноліту та роботи з мікросервісами в цілому. Після цього слід надавати пріоритет тим модулям, які принесуть найбільшу користь після свого витягання.

Перетворення модуля у мікросервіс часто є часозатратним, тому слід правильно класифікувати модулі за їх вигодою. Також вигідно витягувати модулі, чиї запити щодо ресурсів суттєво відрізняються від запитів решти модулів моноліту. Наприклад, корисно перетворити модуль, що має базу даних в пам'яті, на службу, яка потім може бути розгорнута на хостах, представлені серверами, віртуальними машинами або хмарними екземплярами, з великим обсягом пам'яті. Подібним чином, варто витягати модулі, що реалізують складні обчислювальні операції, оскільки сервіс потім може бути розгорнутим на хостах з великою обчислювальною потужністю. Перетворюючи модулі з певними вимогами до ресурсів на мікросервіси, можна зробити застосунок більш легким менш ресурсозатратним при масштабуванні.

Першим кроком вилучення модуля з моноліту є визначення спільного API, оскільки як моноліт потребуватиме дані від сервісу, так і сервіс від моноліту. Проте створення цього інтерфейсу може бути досить важким завданням через надмірні взаємозв’язки модуля із іншими частинами монолітної програми. Як

результат, розробникам доводиться робити значні зміни у коді, аби відокремити модуль в окремий мікросервіс. Як тільки API для нашого модуля буде розроблено, наступним кроком у розробці мікросервісу є налагодження механізму комунікації, реалізуючи синхронний чи асинхронний методи. Останнім кроком у формуванні мікросервісу є його інтегрування у систему, налаштовуючи моніторинг, виявлення сервісів, логування тощо.

РОЗДІЛ 2: ВАЖЛИВІ СКЛАДОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1 API шлюз

Споживач або веб-клієнт працюють у браузері. На клієнтській частині застосунку не повинно бути механізму, який відкрито для клієнта ідентифікував місцезнаходження мікросервісу, або виконував би роль балансувальника навантаження.

Це є головною причиною створення API шлюзу, який з’єднував би усі мікросервіси та абстрагував би це від клієнта. Також за допомогою нього можна централізувати сервіси трансформування або фільтрації запитів та відповідей, налагодження аутентифікації, а також впровадження версійності. Шлюз слугує центральною точкою доступу до системи, застосовуючи певну логіку до всіх мікросервісів без дублювання функціоналу.

Крім того, у випадку, коли один клієнт очікує відповідь у форматі XML (Extensible Markup Language – розширювана мова розмітки), а інший у JSON (JavaScript Object Notation – запис об’єктів JavaScript), то за допомогою шлюзу застосунок може приймати такі запити, перетворювати відповідь відповідно до протоколу або навіть компонувати декілька різних відповідей.

Мікросервісна архітектура передбачає масштабування сервісів, тому API шлюз також здійснює постійну комунікацію з усіма мікросервісами та підтримує реєстр сервісів. Таким чином, у разі несправності якогось мікросервісу, шлюз повідомить про це клієнта і розірве зв’язок для того, щоб уникнути поширення збою по всій системі. Шлюз також може виступати у ролі централізованого управління кешуванням.

2.2 Реєстр сервісів та їх виявлення

Розподіл сервісів поміж усіх доступних хостів є повністю динамічним та залежить від бізнес-можливостей, тому цей розподіл може змінюватися в будь-який момент часу. Для цього існує реєстр сервісів та їх виявлення, які надають змогу подолати обмеження типового веб-серверу, стежить за усіма сервісами у будь-який момент часу та слугує місцем збереження та підтримки комбінацій IP та порту, надаючи клієнтам можливість отримувати доступ до конкретних провайдерів сервісу.

2.2.1 Виявлення сервісів

Згідно з роботою Гія Парти[3], виявлення сервісів є протилежним до реєстру підходом з точки зору клієнтів, оскільки самі клієнти, при бажанні отримати доступ до сервісу, ініціюють зі свого боку пошук інформації про цей сервіс, його місцезнаходження та іншу необхідну інформацію. Виявлення сервісів характеризується тим, що мікросервіси нічого не знають про фізичне розташування інших сервісів, їхній стан чи кількість вузлів. Кожен із сервісів поширює інформацію про своє існування та зникнення, тому при будь-якому збої всередині сервісу, запити до нього перестають надсилатися. Виявлення сервісів має два підходи: виявлення з клієнтської частини або з серверної частини.

При використанні виявлення з клієнтської частини, клієнт або шлюз мають у своїх обов’язках визначення місцезнаходження екземплярів сервісів, а також балансувати навантаження поміж них. Клієнт надсилає запит до реєстру сервісів та в результаті маршрутизує запит згідно з адресою, яка міститься у відповіді реєстру. Фізична локація мережі екземпляру сервісу реєструється при запуску сервісу і видаляється з реєстру при завершенні роботи сервісу. Перевагами такого підходу є легкість підтримки, оскільки сам спосіб, окрім реєстру сервісів, є статичним та завдяки обізнаності клієнта про екземпляр сервісу, це надає йому можливість робити ефективні рішення щодо розподілення навантаження,

основуючись на ситуацію. Проте у цьому підході є недолік, який полягає у тому, що клієнт має досить велику зв’язність із реєстром сервісів

Виявлення з боку серверу полягає у наявності додаткового компоненту для реєстру, який би приймав запити клієнта через балансувальник навантаження та перенаправляв би їх до доступних екземплярів сервісів. Одним із варіантів реалізації цього слугує використання проксі, яке бере на себе відповідальність за перенаправлення запитів до доступних екзмеплярів. Перевагами такого підходу є відсутність необхідності створювати механізми виявлення сервісів у кожному мікросервісі, а також надає певний рівень абстрагованості від клієнту та дозволяє маніпулювати балансуванням навантаження Однак новий компонент для реєстру потребує керування на сервері, а також підтримці більшості протоколів.

2.2.2 Реєстр сервісів

У книзі [3] також вказано, що реєстр сервісів є ключовою складовою механізму виявлення сервісів у розподіленій системі. Він являє собою базу даних, у якій зберігаються мережеві місцезнаходження усім екземплярів сервісів. Оскільки така інформація є суттєвою у застосунку, реєстр сервісів має забезпечувати високу доступність та актуальність даних. Метадані про місцезнаходження мікросервісів, протоколи комунікації, порти та інша необхідна інформація зберігаються у реєстрі сервісів.

Для правильного функціонування системи, необхідно забезпечити реєстрацію та видалення з реєстру усіх екземплярів сервісів. Одним із підходів до реєстрації є власна реєстрація сервісів, яка полягає у тому, що кожен сервіс несе відповідальність за свою реєстрацію та видалення з реєстру. Реєстр повинен постійно знати статус сервісу, тому у цьому підході сервіси постійно надсилають інформацію про свій статус до реєстру. У разі тривалого відсутності нової інформації про статус сервісу, реєстр може зробити висновок про його відмову.

Недоліками такого підходу є зв’язність сервісу із реєстром та децентралізованість логіки, що змушує реалізовувати її у кожному сервісі.

Альтернативним підходом є стороння реєстрація, яка полягає у розробці нового компоненту – реєстратора сервісів – який буде нести відповідальність за керуванням реєстру сервісів. Такий підхід вирішить проблему зв’язності мікросервісів із реєстром. Реєстратор сервісів проводить реєстрацію тоді, коли отримує повідомлення про запуск сервісу, та видаляє з реєстру тоді, коли сервіс припиняє свою роботу або певний час не надсилає інформацію про свій стан. Перевагами цього підходу є зменшення нагромадженості кожного з сервісів та уникнення необхідності реалізовувати механізм реєстрації у кожному мікросервісі. Проте оскільки до системи додається черговий компонент, розробник зобов’язується його підтримувати та забезпечувати високу доступність.

2.3 Механізми комунікації

2.3.1 Співпраця мікросервісів

Мікросервісна архітектура полягає у розподіленні системи на мікросервіси згідно з бізнес-функціями, ізолюючи функціонал всередині. З точки зору користувача, буде надіслано лише один запит до серверу, однак для надання повноцінної відповіді, обробка запиту зазвичай залучатиме декілька інших мікросервісів до роботи. Саме тому для надання повноцінної відповіді кінцевому користувачу мікросервіси повинні вміти співпрацювати між собою, для забезпечення чого існує механізм комунікації.

2.3.2 Типи співпраці

Оскільки кожен мікросервіс відповідає за конкретну бізнес-функцію, існує необхідність поширювати стан, залежності або комунікувати з іншими сервісами. Вирізняють три основні типи співпраці між мікросервісами – команди, запити та події.

У випадку, коли мікросервіс потребує від іншого виконання якоїсь операції, використовуються команди для співпраці сервісів. Команди своєю природою є синхронними та зазвичай реалізовуються за допомогою методів POST та PUT протоколу HTTP. У разі невдачі надсилання команди, сервіс-ініціатор не буде мати інформації про виконання команди в іншому сервісі.

Будучи схожими на команди, запити використовуються мікросервісами, коли вони потребують певну інформацію від іншого сервісу. Є синхронними та зазвичай реалізовуються за допомогою методу GET протоколу HTTP. У разі невдачі з боку відправника інформації, ініціатор запиту не отримує бажану інформацію.

Відрізняючись від стандартних підходів, цей являє собою більше реактивний підхід. Він полягає у використанні подій щоразу, коли виникає необхідність мікросервісу реагувати на те, що виникає в інших мікросервісах. Коли підписник опитує стрічку про наявність подій, то при їх невдачах наслідки цього є повністю прийнятними, оскільки опитування стрічки може бути виконано пізніше, а завдяки асинхронності підходу, затримка у подіях не буде проблемою.

2.3.3 Формати даних

Ключовим аспектом комунікації мікросервісів є обмін повідомленнями будь-яких форматів. Оскільки більшість повідомлень містять в собі якісь дані, то важливим кроком у налаштуванні комунікації є вибір формату даних, так як в результаті це матиме вплив на ефективність комунікації та на розвиток всього застосунку. Існує два типи форматів повідомлень – текстовий та бінарний.

Текстовий формат повідомлень часто представлений форматами JSON та XML, оскільки вони є самовираженими та надають змогу людині зрозуміти їх значення. Ці формати дозволяють обирати значення, які цікавлять споживача, та відкидати решту. Будь-які зміни у форматі схеми є легко зворотними, проте використання текстових форматів має надто багатослівний характер і

призводить до певних накладних витрат при синтаксичному розборі всього тексту. Для забезпечення більшої ефективності часто використовують бінарні формати.

Для визначення структури повідомлення, бінарні формати забезпечують типізовану мову ідентифікації, після чого відбувається серіалізація та десеріалізація повідомлення. Компілятор виконує перевірку правильності використання API, якщо клієнт має статично типізовану мову.

2.3.4 Стилі комунікації

Комунікація мікросервісів може відбуватися за допомогою багатьох різновидів стилів комунікації, кожен з яких націлений ефективно виконувати своє завдання при певних обставинах. Типи комунікації можуть бути поділені на дві різні групи.

Перша група полягає у відповідності до типу протоколу, а саме якої природи він є – синхронної чи асинхронної. Комунікація, реалізована за допомогою команд та запитів, основана на протоколі HTTP, тому він несе синхронний характер. Найчастіше синхронний механізм комунікації реалізовується за допомогою підходу REST. Синхронний механізм полягає у тому, що клієнт після надсилання запиту змушений чекати на відповідь, хоча це очікування по-різному представлене залежно мови програмування. Важливим моментом є те, що клієнт зможе продовжити роботу лише після отримання відповіді від серверу. Асинхронними по природі є протоколи на зразок AMQP чи сокетів. Клієнт зазвичай не чекає на відповідь від серверу, а лише надсилає повідомлення до черги.

Друга група визначається кількістю отримувачів запиту. У разі єдиного отримувача зазвичай використовується тип команди та запиту. Взаємодія з одним отримувачем включає також такі моделі, як запит-відповідь, односторонні запити, на зразок сповіщень, а також запит-асинхронна відповідь. Якщо запит може оброблюватися від нуля до декількох отримувачів, то такий тип повинен

бути асинхронного типу комунікації, наприклад як механізм публікації-підписки, який використовує керовану подіями архітектуру. За допомогою шини сервісів або брокеру повідомлень реалізується пропаговане через події оновлення даних між декількома мікросервісами.

2.3.5 REST

Поняттю REST згідно зі статтею [4] можна визначити як «… архітектурний стиль забезпечення стандартів між комп'ютерними системами в Інтернеті, полегшуючи їх взаємодію між собою. Системи, сумісні з REST, які також часто називають RESTful, характеризуються відсутністю стану та розподілом клієнтської частини та серверної».

В архітектурному стилі REST, реалізації клієнтської частини та серверної можуть бути виконані незалежно від того, чи знають вони одне про одного, що дозволяє ізолювати зміни кожної частини всередині них. Зв'язок між цими компонентами підтримується на основі кінцевих точок REST, надаючи змогу різним клієнтам отримувати однакові відповіді та виконувати схожі дії.

REST передбачає відсутність стану, яке проявляться у тому, що клієнтська частина та серверна не знають про стан один одного, надаючи змогу розуміти повідомлення без переглядання чи запам'ятовування попереднього. Це досягається за допомогою використання замість команд ресурсів, які надають змогу описувати будь-який об'єкт чи сутність, яку потрібно надіслати іншому сервісу. Оскільки REST покладається на стандартні операції над ресурсами, для нього немає необхідності у визначенні інтерфейсів, що надає RESTful застосункам надійності при швидкій роботі та масштабованості, оскільки клієнт та сервер розділені між собою на самостійні компоненти.

Комунікація у RESTful системах відбувається за допомогою протоколу HTTP та чотирьом головним методам цього протоколу:

- GET – для отримання колекції ресурсів («/ресурс») або якогось конкретного ресурсу за його ідентифікаційним кодом («/ресурс/:код»).
- POST – для створення ресурсу («/ресурс).
- PUT – для проведення оновлення конкретного ресурсу за його ідентифікаційним кодом («/ресурс/:код»).
- DELETE – видалення ресурсу за його ідентифікаційним кодом («/ресурс/:код»).

У переліку заголовків запиту клієнт надсилає тип вмісту відповіді від серверу, який може прийняти. Це відбувається за допомогою вказування заголовку «Accept» та типу вмісту. Існує 5 основних типів: text, audio, image, video, application.

РОЗДІЛ 3: ОПИС ОБРАНИХ ТЕХНОЛОГІЙ, БІБЛІОТЕК, ПРИНЦИПІВ ДИЗАЙНУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. ЇХ ЗАСТОСУВАННЯ ПРИ РОЗРОБЦІ ВЕБ-ЗАСТОСУНКУ

3.1 Опис обраних технологій, бібліотек, принципів дизайну програмного забезпечення

3.1.1 Опис технологій

3.1.1.1 MERN

У результаті розвитку веб-розробки виникло поняття SPA, що полягає у наявності на клієнтській частині лише однієї сторінки, яка оновлює свій стан за допомогою легких запитів на сервер. В результаті появи нової парадигми, почали з’являтися нові веб-фреймворки, такі як Angular, React, Vue, але для побудови повноцінного веб-застосунку необхідні додаткові технології.

MERN стек технологій, назва якого є акронімом від переліку використаних технологій MongoDB, Express, React та Node.js, є одним із найпопулярніших способів розробки веб-застосунків на сьогоднішній день. Оскільки кожна з технологій використовує мову програмування Javascript, при використанні цього стеку створюється певна екосистема, яка сприяє швидкості, зручності та якості розробки, а також включає в себе також менеджер пакетів npm з величезною бібліотекою пакетів, які дуже легко та доступно підключаються до застосунку. Також при використанні MERN стеку, застосунок обмінюється інформацією у форматі JSON скрізь – у базі даних, у клієнтській частині і сервері. Це допомагає економити час, який в іншому випадку витратився б на трансформації даних.

3.1.1.1.1 MongoDB

MongoDB є прикладом бази даних, побудованої на основі горизонтально масштабованої архітектури. Вона має відкритий код, а також є дуже популярний рішенням серед розробників.

371 systems in ranking, May 2021

Rank			DBMS	Database Model	Score		
May 2021	Apr 2021	May 2020			May 2021	Apr 2021	May 2020
1.	1.	1.	Oracle	Relational, Multi-model	1269.94	-4.98	-75.50
2.	2.	2.	MySQL	Relational, Multi-model	1236.38	+15.69	-46.26
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	992.66	-15.30	-85.64
4.	4.	4.	PostgreSQL	Relational, Multi-model	559.25	+5.73	+44.45
5.	5.	5.	MongoDB	Document, Multi-model	481.01	+11.04	+42.02
6.	6.	6.	IBM Db2	Relational, Multi-model	166.66	+8.88	+4.02
7.	7.	↑ 8.	Redis	Key-value, Multi-model	162.17	+6.28	+18.69
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model	155.35	+3.18	+6.23
9.	9.	9.	SQLite	Relational	126.69	+1.64	+3.66
10.	10.	10.	Microsoft Access	Relational	115.40	-1.33	-4.50

Рисунок 3.1 – Статистика популярності баз даних (https://db-engines.com/en/ranking)

MongoDB є різновидом NoSQL баз даних, тому на відміну від SQL баз даних представляє дані у вигляді документів у форматі JSON, а не у вигляді таблиць, стовпчиків та рядків, як це відбувається в мовах SQL. Бази даних документів надають розробникам досить потужної гнучкості у визначенні структури, а також дозволяє мати вбудовані документи. Поля документів виконують роль стовпчиків, як у SQL, а також можуть бути індексовані для покращення пошуку. Найбільша перевага бази даних документів полягає у можливості змінення структури документів у будь-який момент часу. MongoDB також підтримує транзакції, що надає змогу групувати багато змін та разом виконувати або скасовувати.

3.1.1.1.2 Express

Express.js – це фреймворк Node.js для розробки веб-додатків, який має відкритий код, та дозволяє розробникам швидко та просто проектувати та реалізовувати веб-застосунки.

Express надає розширені можливості розробки механізму маршрутизації, а також дозволяє розробляти проміжні програмні забезпечення для зручної маніпуляції запитами та відповідями, а також керування помилками. Головним призначенням цього фреймворку є полегшення та прискорення розробки серверної частини застосунку на Node.js.

3.1.1.1.3 React

Всесвітньо відома компанія Facebook є засновником бібліотеки React у 2011 році. Спочатку бібліотека використовувалася для корпоративних цілей компанії, але пізніше стала доступною для всіх розробників та на кінець 2020 рік досягла неабиякої популярності [5].

Реакт оперує неабиякою зручністю для розробника, оскільки він доступний для використання на більшості платформах, а також націлений на єдину задачу – створення веб-компонентів. Також він надає досить зручний синтаксис, саме за який він і набув такої популярності. React має сильну підтримку зі сторони Facebook, а також величезну спільноту розробників, що забезпечує розробників великою кількістю доступних для використання пакетів. Швидкість роботи React є черговою перевагою. Вона забезпечена за допомогою роботи віртуального DOM, який порівнює себе з реальним DOM та шукає найбільш ефективні способи його оновлення. Також дизайн технології React полегшує тестування.

3.1.1.1.4 Node.js

Node.js побудований на двигуні Chrome V8 та слугує середовищем виконання програм, написаних мовою Javascript, поза межами браузера. Node.js використовує свою власну модульну систему для збірки декількох файлів Javascript. Також це надає змогу використовувати сторонні пакети, які можна завантажити з величезної бібліотеки пакетів npm, яка постійно поповнюється зростаючою спільнотою розробників. Node.js має асинхронну природу, керовану подіями та модель, яка не блокується введенням чи виведенням. В той час, коли обслуговування виконання паралельних задач в інших мовах передбачає створення додаткових потоків, Node.js справляється з цією задачею за допомогою зворотніх викликів та петлею подій, що являє собою чергу подій, які повинні бути обробленими.

3.1.1.2 Typescript

Typescript є надмножиною Javascript, яка вирішує головну проблему останнього – динамічну типізацію. За допомогою статичної типізації, Typescript, згідно з дослідженням [6], допомагає уникати близько 15% усіх помилок, характерних програмам на Javascript. Typescript можна використовувати як на клієнтській стороні застосунку, так і на стороні серверу. Причини використання Typescript полягають у наданні застосунку більшої безпеки від помилок. Також він допомагає більш явно побачити суть застосунку, за допомогою статичної типізації, а також, надаючи змогу визначати інтерфейси, наслідуваність та абстрактність, Typescript допомагає у більш якісній структуризації всього застосунку. Оскільки в кінці Typescript все одно компілюється до Javascript, перехід від Javascript до Typescript не є складною задачею.

3.1.1.3 JWT

На головному сайті розробників JWT [7] надається визначення: «це відкритий стандарт, який визначає компактний та автономний спосіб безпечної передачі інформації між сторонами як об'єкт JSON. Цю інформацію можна перевірити та довіритися їй, оскільки вона має цифровий підпис. JWT можна підписати, використовуючи секрет (з алгоритмом HMAC) або пару відкритих / приватних ключів, використовуючи RSA або ECDSA».

JWT в основному використовується для авторизації, оскільки містить у собі дані про користувача, надаючи серверу змогу ідентифікувати його та авторизувати, а також для безпечного обміну інформацією, оскільки сервер може перевіряти цифровий підпис.

Структура JWT складається з 3 частин, розділених між собою крапками:

1) Заголовок – зазвичай містить інформацію про тим токена, а також алгоритм, який був використаний для підпису.
2) Корисне навантаження – містить у собі дату і час ініціалізації токену, а також дату і час закінчення терміну його дії. Також сервер може наповнювати цю частину своїми власними даними.
3) Підпис – для створення підпису береться закодовані заголовок, корисне навантаження, а також секрет, який надається при створенні токену. Підпис виконує роль перевірки недоторканості токену, а вірність відправника.

JWT має вигляд: «заголовок.навантаження.підпис»

3.1.2 Опис бібліотек

3.1.2.1 Awilix

Awilix є досить потужним контейнером для ін’єкції залежностей, який використовується у програмах Node.js та надає досить простий API. Для початку роботи, Awilix повинен створити контейнер.

```
import * as awilix from "awilix";
import { createContainer } from "awilix";
import { UserController } from "./controllers/UserController";

const container = createContainer( options: {
    injectionMode: awilix.InjectionMode.PROXY
});

container.register( nameAndRegistrationPair: {
    userController: awilix.asClass(UserController),
})
```

Рисунок 3.3 – Приклад створення контейнеру Awilix

Після створення контейнеру, до нього можна реєструвати значення, сервіси, задаючи пари ключ-значення в об’єкті. Awilix надає змогу керувати тривалістю життя зареєстрованих екземплярів. Існує три типи тривалості життя:

1) Lifetime.Transient – є стандартним значенням. Він полягає у створенні нового екземпляру зареєстрованої одиниці при кожному її виклику із контейнера.

2) Lifetime.Scoped – той самий екземпляр буде використовуватися кожного разу, коли звернення до зареєстрованої одиниці відбувається в тих же межах.

3) Lifetime.Singleton – кожного разу при зверненні до зареєстрованої одиниці, буде використовуватися той самий екземпляр.

Для використання контейнеру на кінцевих точках, створюється проміжне програмне забезпечення, яке назначає властивість «container» із значенням вказаного контейнеру кожному вхідному запиту. Всі зареєстровані одиниці контейнеру знаходяться у властивості «cradle».

```
@Get( route: '/')
async getUsers(@Req() req: ContainerReq): Promise<IUser[]> {
    const { getUsers } = req.container.cradle;
    return await getUsers.execute();
}
```

Рисунок 3.4 – Використання зареєстрованих одиниць контейнеру у контроллері

3.1.2.2 Routing-controllers

Routing-controllers надає змогу розробнику створювати та застосовувати контролери у Node.js у більш об’єктно-орієнтовному вигляді. Бібліотека надає зручний API для витягання тіла, параметрів та іншої інформації із запитів. Надає API для задання проміжного програмного забезпечення та стандартного обробника помилок. Створення контролерів та проміжного програмного забезпечення в бібліотеці реалізовано за допомогою використання анотацій у мові Typescript. Приклад створення контролеру наведено у додатку А.

3.1.2.3 Mongoose

Бібліотека Mongoose являє собою ODM для MongoDB та допомагає в управлінні взаємозв'язками між даними, використовується для відображення документів із MongoDB в об'єкти Javascript, а також надає валідацію схем. Також великою перевагою Mongoose є використання проміжного програмного забезпечення, яке допомагає зручно вирішувати нестандартне внесення даних чи їх витягнення з бази даних.

```
45  AccountSchema.pre<IAccountDocument>(
46      method: 'save',
47      fn: async function (next : HookNextFunction ) {
48          const user = this;
49          user.password = await bcrypt.hash(user.password,  saltOrRounds: 10);
50          next();
51      });
```

Рисунок 3.5 – Приклад використання проміжного програмного забезпечення у Mongoose для шифрування паролю перед внесенням об'єкту до бази даних

3.1.2.4 Axios

Бібліотека axios надає зручний API для надсилання запитів до серверу.

```
(axios.get( url: 'http://localhost:5000/auth'));
```

Рисунок 3.6 – Приклад надсилання GET-запиту за допомогою axios

Вона підтримує усі методи HTTP запитів, дозволяє вказувати певні налаштування запитів, а також надає змогу визначати перехоплювачі запитів чи відповідей.

```
axios.interceptors.request.use(
    onFulfilled: async (config : AxiosRequestConfig ) => {
        config.headers = {
            "Authorization": 'Bearer ' + localStorage.getItem( key: 'token')
        }

        return config;
    }
)
```

Рисунок 3.7 – Приклад роботи перехоплювача запитів у axios

3.1.3 Чиста архітектура

Роберт Мартін навів приклад діаграми у додатку Б, вказуючи у своїй статті [8]: «…усі стилі розробки архітектури застосунків відрізняються між собою у певних деталях, але є дуже схожими. Усі вони мають одну і ту ж мету, яка полягає у розділенні переконань. Всі вони досягають цього розділення, розділяючи програмне забезпечення на шари. Кожен має принаймні один рівень для бізнес-правил, а інший - для інтерфейсів».

Усі підходи до створення архітектури полягають у створенні застосунку, який буде:

1) Незалежним від інтерфейсу користувача, оскільки він характеризується частими змінами і тому не повинен мати впливу на застосунок.
2) Незалежним від виду бази даних, роблячи систему неприв’язаною до конкретної бази даних
3) Придатною до тестування
4) Незалежною від будь-яких фреймворків, позбавляючись необхідності підлаштовуватися під їхні обмеження
5) Незалежною від будь-яких сторонніх сервісів

Чиста архітектура полягає в дотриманні правила залежності, яке полягає у тому, що залежності повинні йти від зовнішнього шару до внутрішнього. Такий

підхід забезпечить мінімальний об’єм коду, який потрібно змінити при будь-якій зміні наявного функціоналу чи внесенні нового.

Центральний шар в архітектурі містить у собі сутності, оскільки вони поширюються не лише на застосунок, а й на концепцію усього конкретного бізнесу чи підприємства, тому сутності є найменш вірогідні бути зміненими.

Наступний шар полягає у визначенні бізнес-логіки застосунку та містить у собі приклади використання сутностей. Вони виконують роль направлення потоку даних від та до сутностей.

Шар інтерфейсів та адаптерів слугують проміжним шаром між випадками використання та зовнішніми сервісами, забезпечуючи з’єднання між внутрішніми сервісами та зовнішніми.

Останній шар фреймворків та зовнішніх сервісів повністю складається із коду, який є найбільш вірогідним та очікуваним до змін. З цієї причини, концепція Чистої архітектури полягає у винесенні таких частин застосунку у зовнішній, що при дотриманні правила залежностей допоможе ізолювати середину застосунку від наслідків змін у коді.

3.2 Розробка веб-застосунку

3.2.1 Предметна область

Ідея веб-застосунку полягає у консультаційному сервісі. Користувачі виступають в ролі звичайного користувача та у ролі спеціаліста, тобто консультанта. Вони мають змогу переглядати публічні пости, які містять у собі певне питання та мають коментарі-відповіді інших користувачів. Також є можливість замовити приватну консультацію у спеціаліста.

3.2.2 Модель застосунку

Архітектура веб-застосунку була розроблена, використовуючи підхід мікросервісної архітектури, а зображення його моделі знаходиться у додатку В. Під час проектування було вирішено розділити застосунок на такі мікросервіси: сервіс користувачів, постів, сфер, консультацій та рекомендацій, а також API шлюз. Кожен з мікросервісів працює на своєму окремому порті. Реєстром сервісів слугує файл середовища, у якому в змінних середовища зберігається номер порту. Усі запити з клієнтської частини надсилаються до API шлюзу, який проксує ці запити до відповідних мікросервісів, залежно від маршруту запиту, використовуючи бібліотеку http-proxy-middleware.

```
7   POST_SERVICE_PORT=5001
8   CONSULTATION_SERVICE_PORT=5002
9   SPHERE_SERVICE_PORT=5003
10  USER_SERVICE_PORT=5004
11  RECOMMENDATION_SERVICE_PORT=5005
```

Рисунок 3.8 – Змінні середовища, які містять порти мікросервісів

```
this.app.use('/posts/*',
    createProxyMiddleware( context: '/posts', options: {
        target: `http://localhost:${process.env.POST_SERVICE_PORT}`,
        changeOrigin: true,
    }));
```

Рисунок 3.9 – Приклад проксування запитів до сервісу постів

3.2.3 Управління даними

Загальна діаграма організації даних наведена у додатку Г.1. Оскільки мікросервісна архітектура передбачає наявність децентралізації управління

даними, при розробці застосунку було вирішено надати кожному мікросервісу окрему базу даних. Для економії ресурсів при надсиланні запитів іншим мікросервісам, а також з метою збереження слабої зв’язності між ними були розроблені дублюючі сутності у певних сервісах, як це можна побачити на прикладах, зображених у додатках Г.2, Г.3 та Г.4. Сервіси постів та консультацій потребують обробки інформації користувачів, тому було прийняте рішення створити дублюючі сутності із мінімально необхідним набором атрибутів. Діаграма сервісу сфер наведена у додатку Г.5.

3.2.4 Комунікація між мікросервісами

Оскільки у застосунку відсутні операції, які потребували б значних обчислювальних можливостей, механізм комунікації між мікросервісами здійснюється за допомогою методу запит-відповідь, використовуючи протокол HTTP та принцип REST. Форматом повідомлень у застосунку виступає JSON, враховуючи використання стеку MERN.

Яскравим прикладом комунікації мікросервісів слугує забезпечення цілісності баз даних системи при створенні нового користувача, оскільки сервіси постів та консультацій також зберігають відповідні дублікати користувачів у власних базах даних.

Це відбувається наступним чином:

1) Клієнтська частина надсилає POST запит до API шлюзу, маючи у тілі повідомлення необхідну інформацію для реєстрації користувача. Приклад запиту наведено у додатку Д.1.
2) API шлюз приймає запит, звіряється з маршрутом запиту та перенаправляє цей запит до сервісу користувачів.
3) Сервіс користувачів приймає запит, звіряє маршрут запиту та направляє його до відповідного контролера. Контролер відповідальний за реєстрацію можна знайти у додатку Д.2.

4) Після створення користувача у сервісі користувачів, поточний сервіс відправляє запити на створення дублюючих сутностей в інших сервісах. Це проілюстровано у додатку Д.3.
5) Сервіс надсилає відповідь, яка містить новостворену сутність користувача.
6) API шлюз приймає відповідь від сервісу та повертає її клієнту.

3.2.5 Дизайн сервісів

У дизайні кожного з мікросервісів було використано принцип Чистої архітектури запропонований Робертом Мартіном. Для детального висвітлення розробленого дизайну буде використано сервіс користувачів як приклад.

Сервіс було поділено на 3 шари: домен, доступ до даних та інфраструктура.

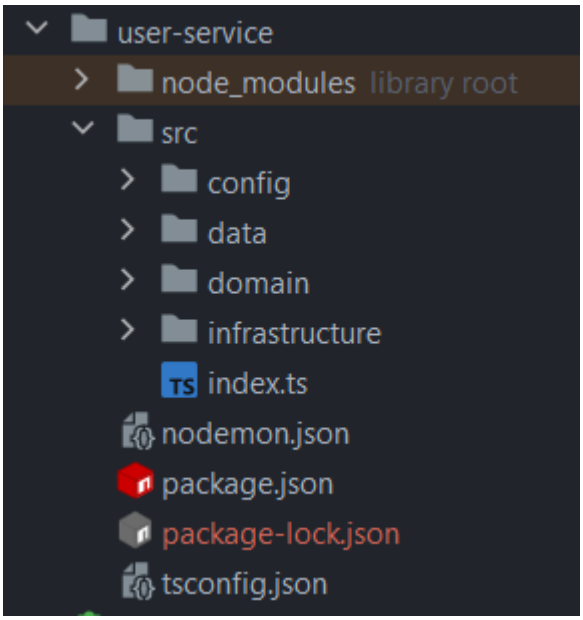


Рисунок 3.10 – Приклад структури сервісу користувачів

Шар домену включає в себе ключові сутності, які задіяні у всьому застосунку. Оскільки це є центральним шаром, компоненти цього шару не мають зовнішніх залежностей.

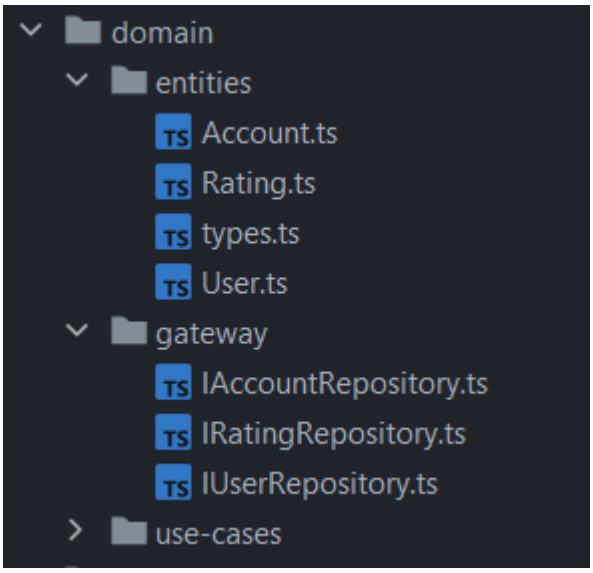


Рисунок 3.11 – Шар домену сервісу користувачів

Оскільки для реалізації випадків використання сутностей потрібне знання функціоналу репозиторію, який через свої залежності на зовнішні служби знаходиться у шарі доступу до даних, то для уникнення порушення правила залежностей Чистої архітектури, було використано принцип інверсії залежностей – у сервісі було визначено так званий шлюз, який слугує зв'язком із репозиторієм і являє собою його інтерфейс. Приклад реалізації випадку використання знаходиться у додатку Е.1.

Шар доступу до даних містить у собі схеми для використання ODM Mongoose, а також реалізації репозиторіїв, які імплементують інтерфейс-шлюз із внутрішнього шару.

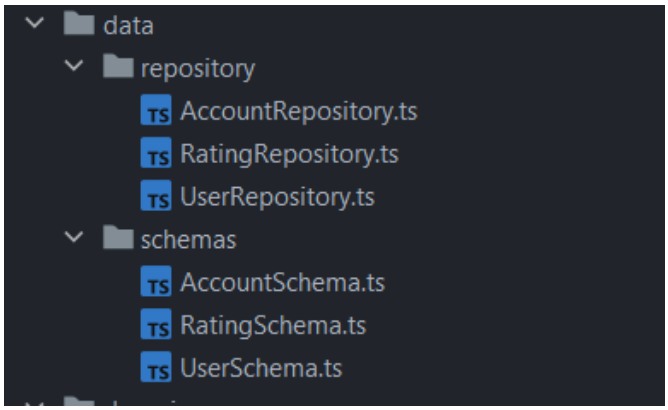


Рисунок 3.12 – Структура шару доступу до даних

Приклад реалізації репозиторію наведений у додатку Е.2.

Шар інфраструктури містить у собі засоби зв’язку внутрішньої частини застосунку із зовнішньою, а також містить деякі допоміжні функції, такі як розсилка запитів на створення дублікатів, перевірка JWT токену тощо.

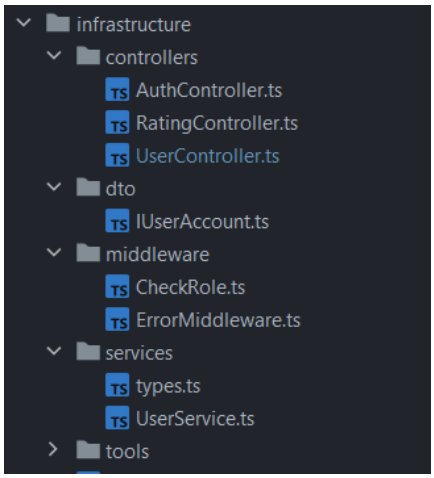


Рисунок 3.13 – Структура шару інфраструктури

Також у кожному сервісі містяться файли для налаштування та запуску мікросервісу, які знаходяться у директорії «configs». Приклад створення екземпляру бібліотеки Express, а також його налаштування наведені у додатку Е.3.

3.2.6 Авторизація

Авторизація у застосунку реалізована за допомогою JWT токенів. При надсиланні клієнтом запитів до сервісу користувачів, а саме до маршруту, який відповідає за реєстрацію, вхід та аутентифікацію, у разі успіху сервер повертає екземпляр користувача, а також JWT токен для подальшого використання у застосунку. Для зручного та ефективного використання цих токенів слугує API шлюз, який при кожному вхідному запиті перевіряє коректність та актуальність токену та оновлює його термін придатності, а також додає заголовок авторизації до запиту. В окремих мікросервісах за допомогою бібліотеки routing-controllers зазначається метод перевірки авторизації, а також анотуються

усі маршрути, які потребують авторизації. Приклад оновлення токену наведено у додатку Ж.1, а перевірка авторизації у додатку Ж.2.

Висновки

У роботі було досліджено та описано стиль розробки архітектури програмного застосунку на основі мікросервісів, його особливості, переваги та недоліки у порівнянні з монолітною архітектурою, а також проблеми і можливі складнощі в її реалізації. На початок 2021 року домінуючою стала думка про те, що мікросервісна архітектура більше підходить для застосунків, для яких очікуються часті зміни у вимогах та реалізації, особливо у випадках, коли передбачається розміщення застосунку у хмарі.

Були описані та охарактеризовані основні складові мікросервісної архітектури, такі як API шлюз, реєстр сервісів та їх виявлення, а також механізми комунікації та методи їх реалізація. Основна увага була приділена RESTful архітектури.

Описані основні підходи до міграції від монолітної архітектури до мікросервісів.

На основі мікросервісного підходу було створено веб-застосунок, який полягає у наданні консультацій користувачам. В ході розробки були випробувані основні підходи до створення застосунку з мікросервісною архітектурою, а також необхідні для цього технології.

Список використаної літератури

1. Fowler M. Microservices [Електронний ресурс] / M. Fowler, J. Lewis. – 2014. – Режим доступу до ресурсу: https://martinfowler.com/articles/microservices.html.
2. Richardson C. Introduction to Microservices [Електронний ресурс] / Chris Richardson. – 2015. – Режим доступу до ресурсу: https://www.nginx.com/blog/introduction-to-microservices/.
3. Ghiya P. Typescript Microservices: Build, deploy, and secure microservices using TypeScript combined with Node.js / Parth Ghiya. – Birmingham: Packt, 2018. – 373 с. – (Packt).
4. What is REST? [Електронний ресурс] // Codecademy. – 2021. – Режим доступу до ресурсу: https://www.codecademy.com/articles/what-is-rest.
5. Number of stars on GitHub projects for Angular, React, and Vue [Електронний ресурс] – Режим доступу до ресурсу: https://iotvnaw69daj.i.optimole.com/AXVzL2w.n2y9~6666f/w:auto/h:auto/q:90/https://www.codeinwp.com/wp-content/uploads/2019/01/2021-star-history.png.
6. Gao Z. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript [Електронний ресурс] / Z. Gao, C. Bird, E. T. Barr – Режим доступу до ресурсу: https://earlbarr.com/publications/typestudy.pdf.
7. JSON Web Token Introduction [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://jwt.io/introduction.
8. Martin R. C. The Clean Architecture [Електронний ресурс] / Robert C. Martin. – 2012. – Режим доступу до ресурсу: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.
9. The Clean Architecture [Електронний ресурс] – Режим доступу до ресурсу: https://blog.cleancoder.com/uncle-bob/images/2012-08-13-the-clean-architecture/CleanArchitecture.jpg.

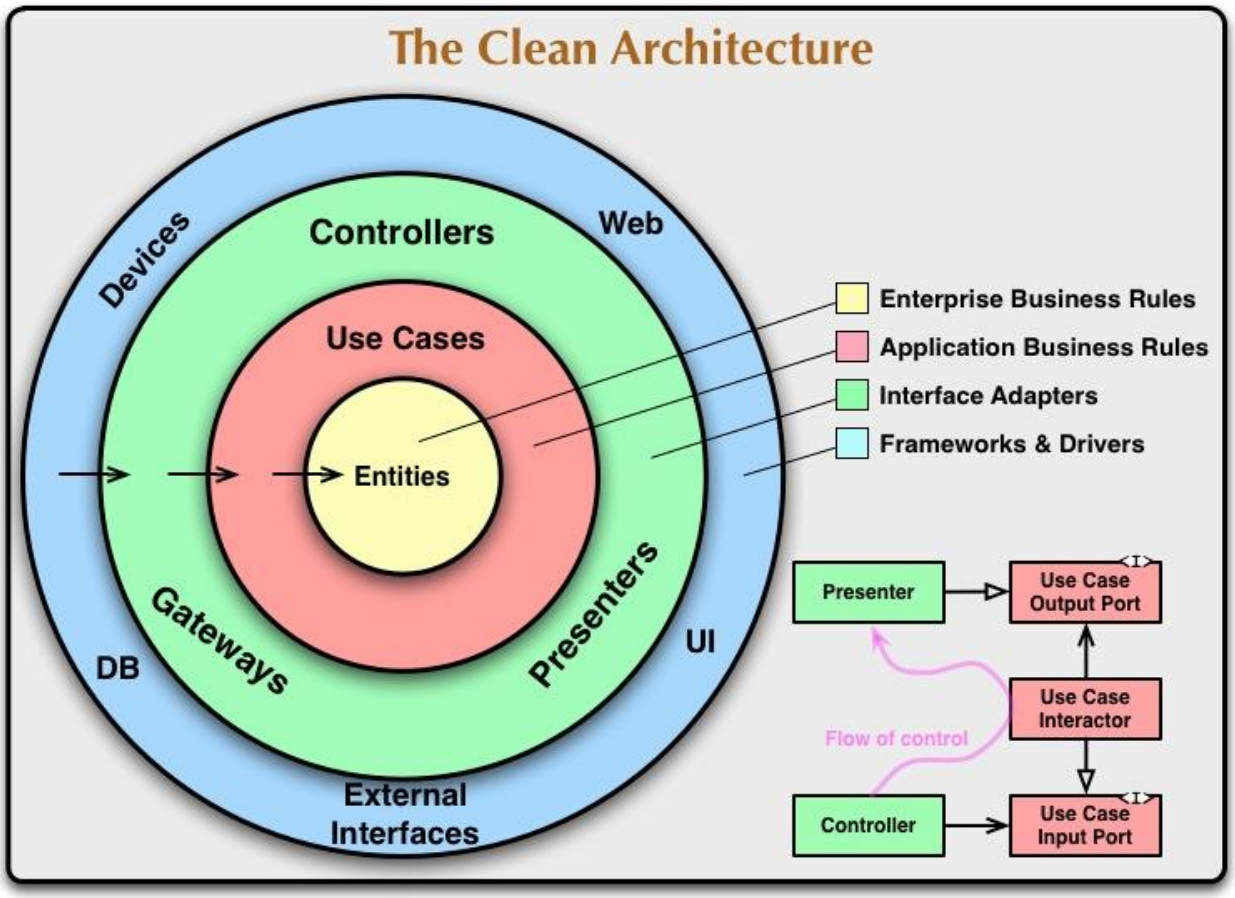
Додаток А

Приклад створення контролеру бібліотекою routing-controllers

```
@Authorized( role: [Role.Consultant])
@JsonController( baseRoute: '/users')
export class UserController {

    @Get( route: '/')
    async getUsers(@Req() req: ContainerReq): Promise<IUser[]> {
        const { getUsers } = req.container.cradle;
        return await getUsers.execute();
    }

    @Get( route: '/:id')
    async getUser(@Req() req: ContainerReq, @Param( name: 'id') id: string): Promise<IUser> {
        const { getUserById } = req.container.cradle;
        return await getUserById.execute(id);
    }

    @Post( route: '/')
    async createUser(@Req() req: ContainerReq, @Body() userProps: IUser): Promise<IUser> {
        const { createUser } = req.container.cradle;
        return await createUser.execute(userProps);
    }
```

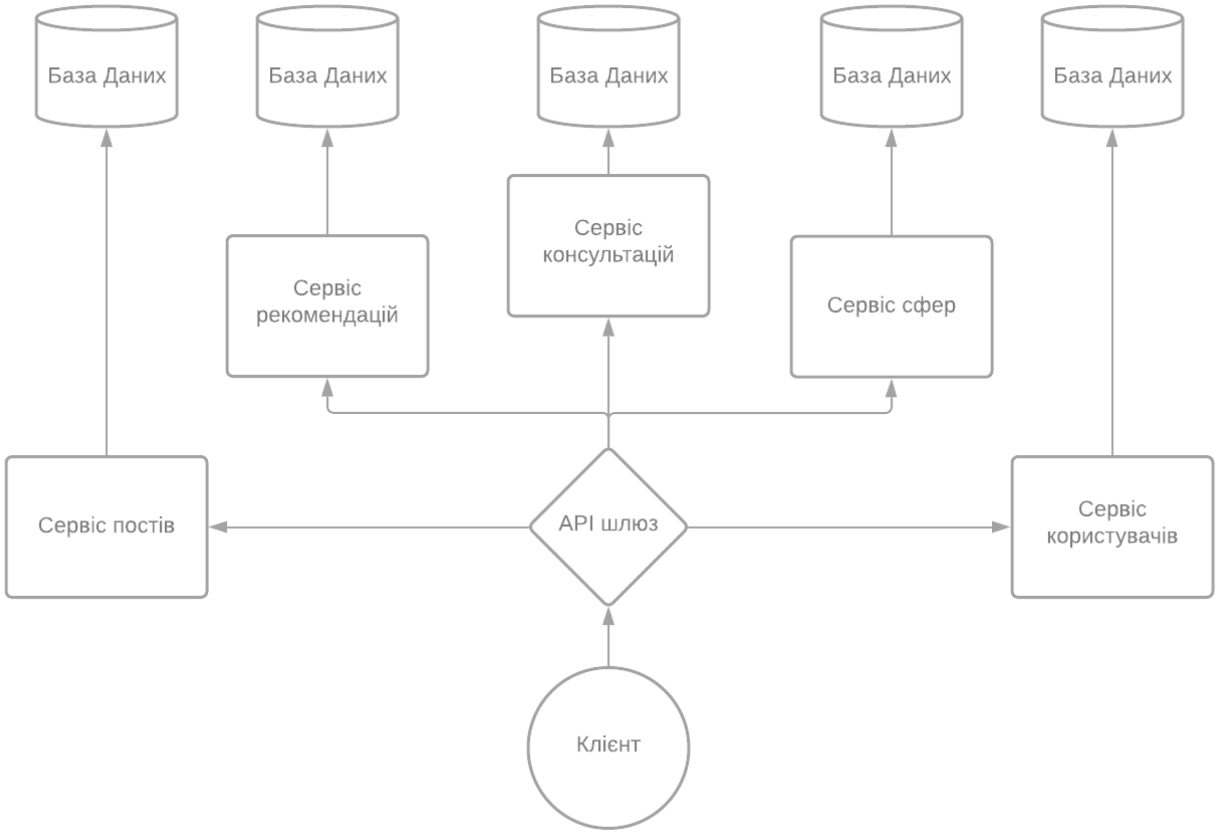
Додаток Б

Діаграма принципу Чистої архітектури за Робертом Мартіном [9]



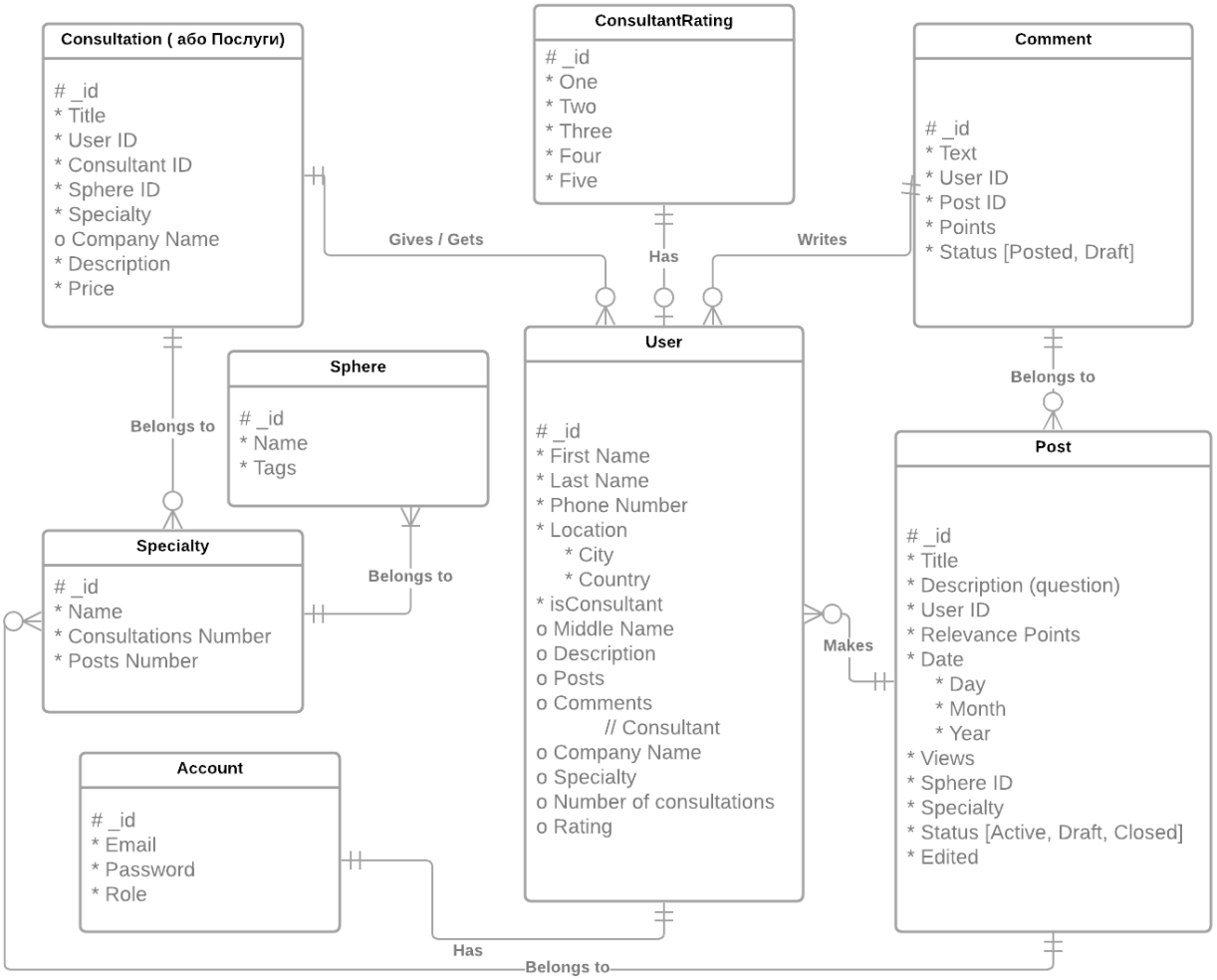
Додаток В

Модель розробленого веб-застосунку



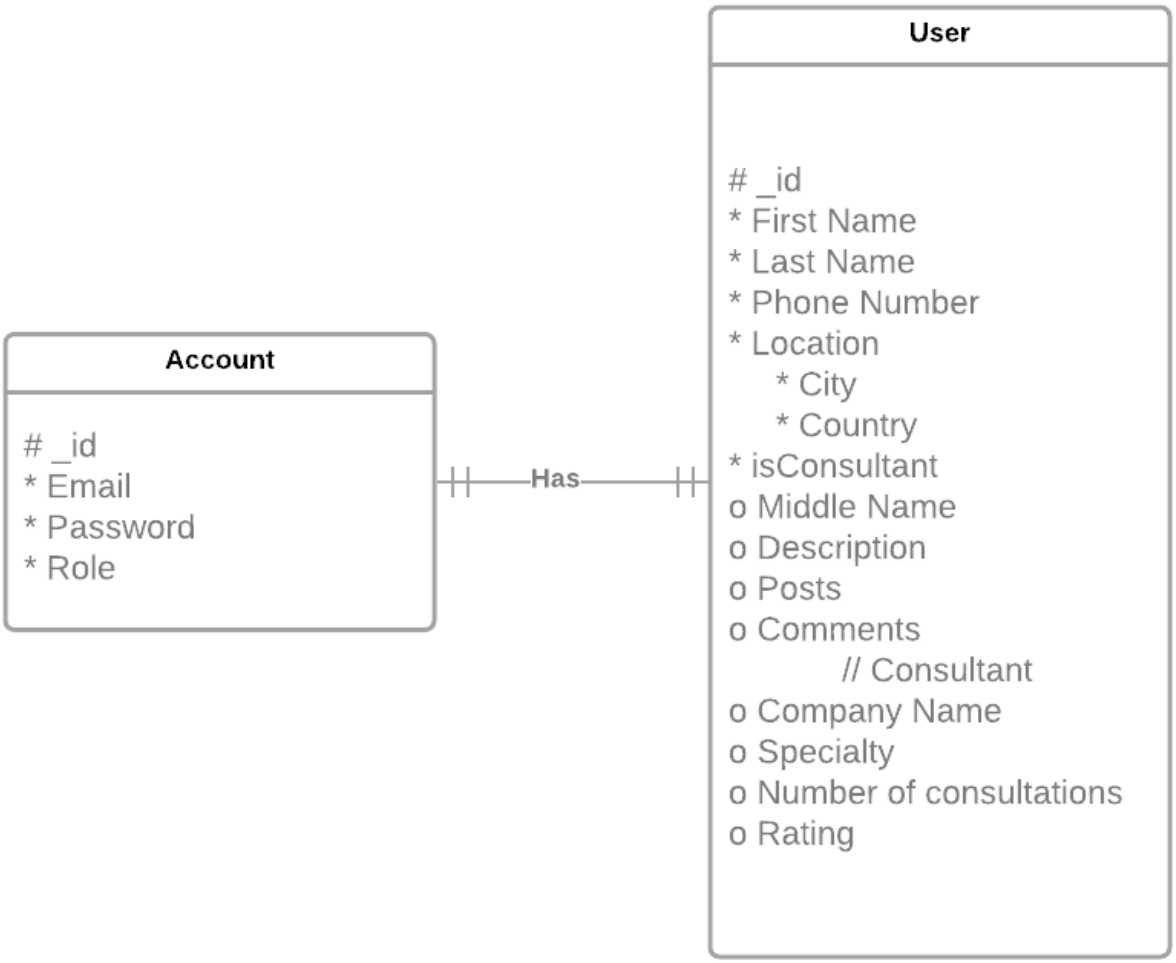
Додаток Г.1

Загальна діаграма організації даних



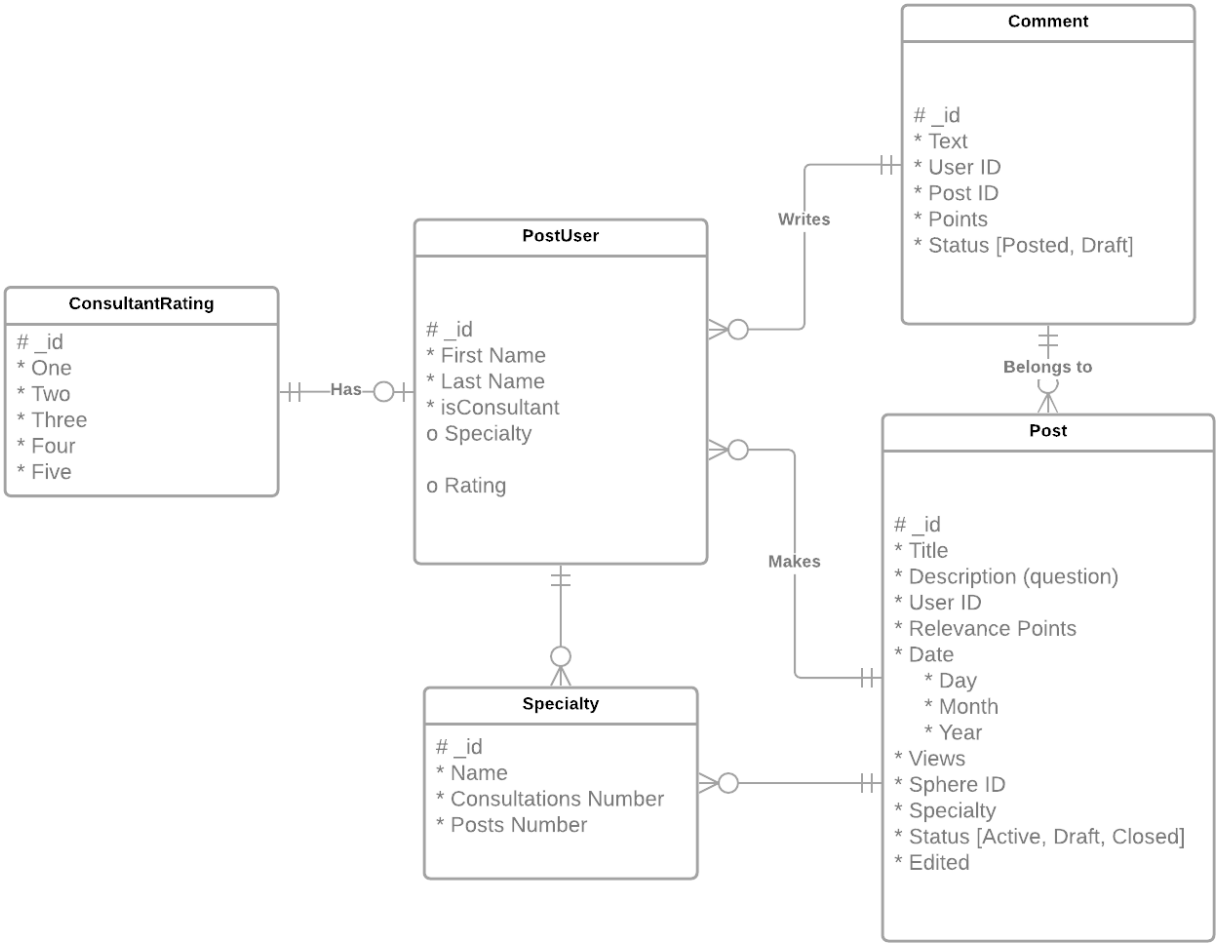
Додаток Г.2

Діаграма організації даних у сервісі користувачів



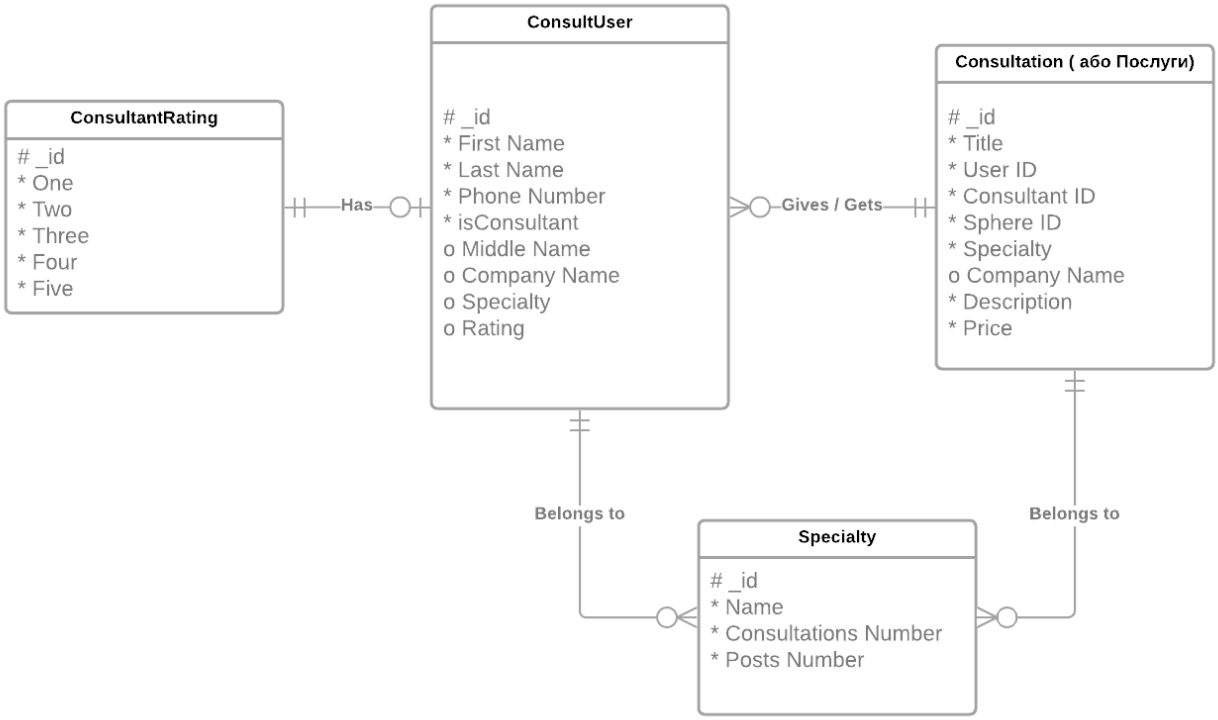
Додаток Г.3

Діаграма організації даних у сервісі постів



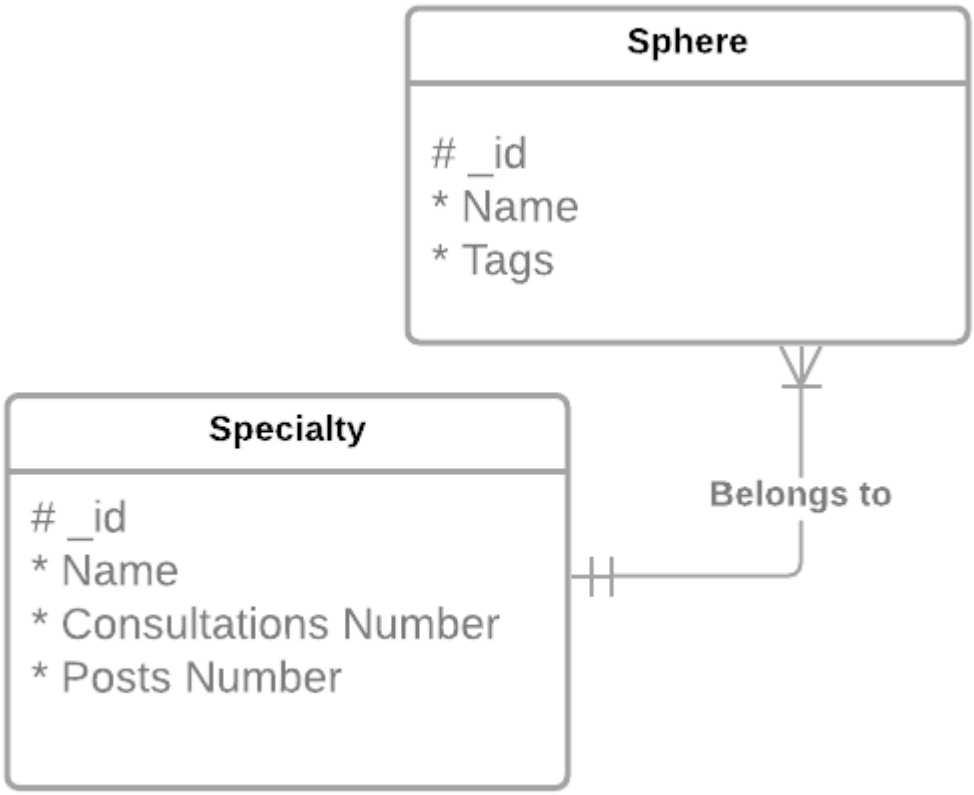
Додаток Г.4

Діаграма організації даних у сервісі консультацій



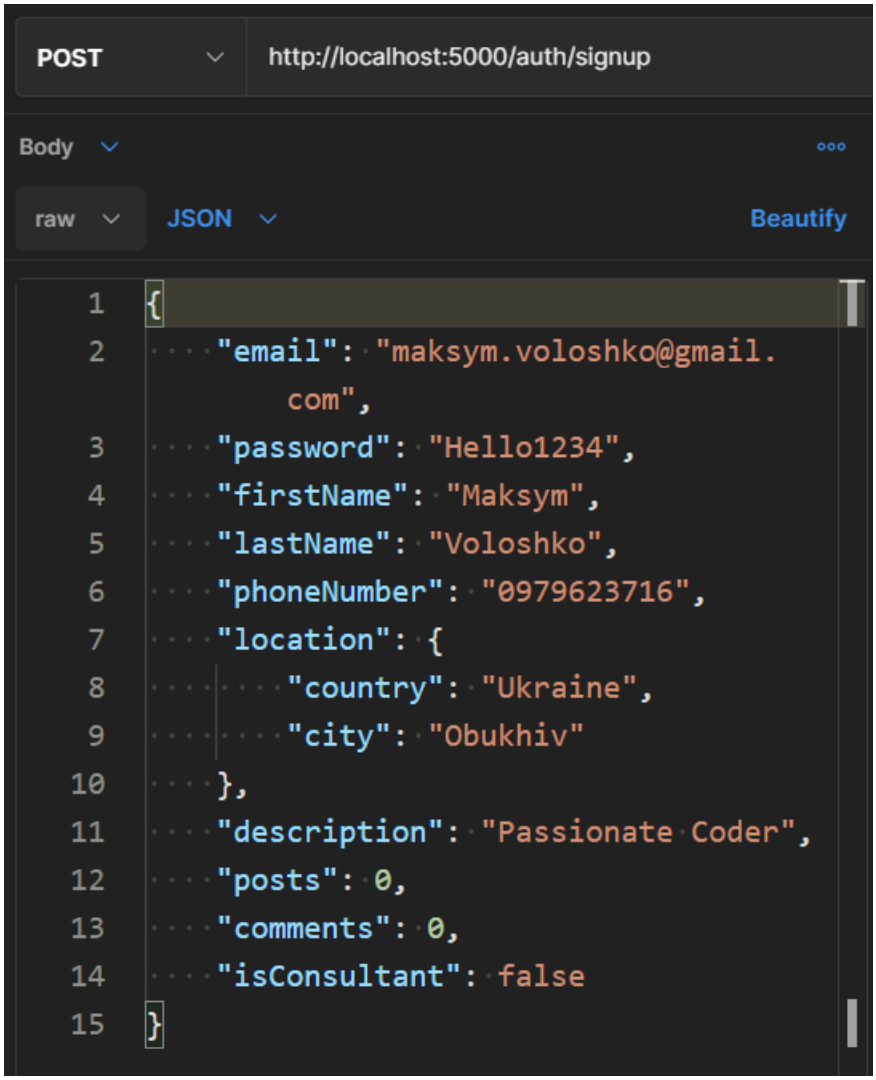
Додаток Г.5

Діаграма організації даних у сервісі сфер



Додаток Д.1

Приклад запиту на реєстрацію



Додаток Д.2

Реалізацію контролеру реєстрації користувача

```
@Post( route: '/signup')
async signup(@Req() req: ContainerReq, @Res() res: Response, @Body() body: any): Promise<any> {
    const { userService } = req.container.cradle;
    const [err, user] = await to<UserAccount>(userService.signup(req.body));

    if (err) throw err;

    const token = generateJwtToken( user: {
        email: user.email,
        userID: user.userID,
        role: user.role
    });

    res.setHeader( name: 'Authorization', value: 'Bearer ' + token);

    return {
        message: 'User has successfully signed up',
        user,
        token
    }
}
```

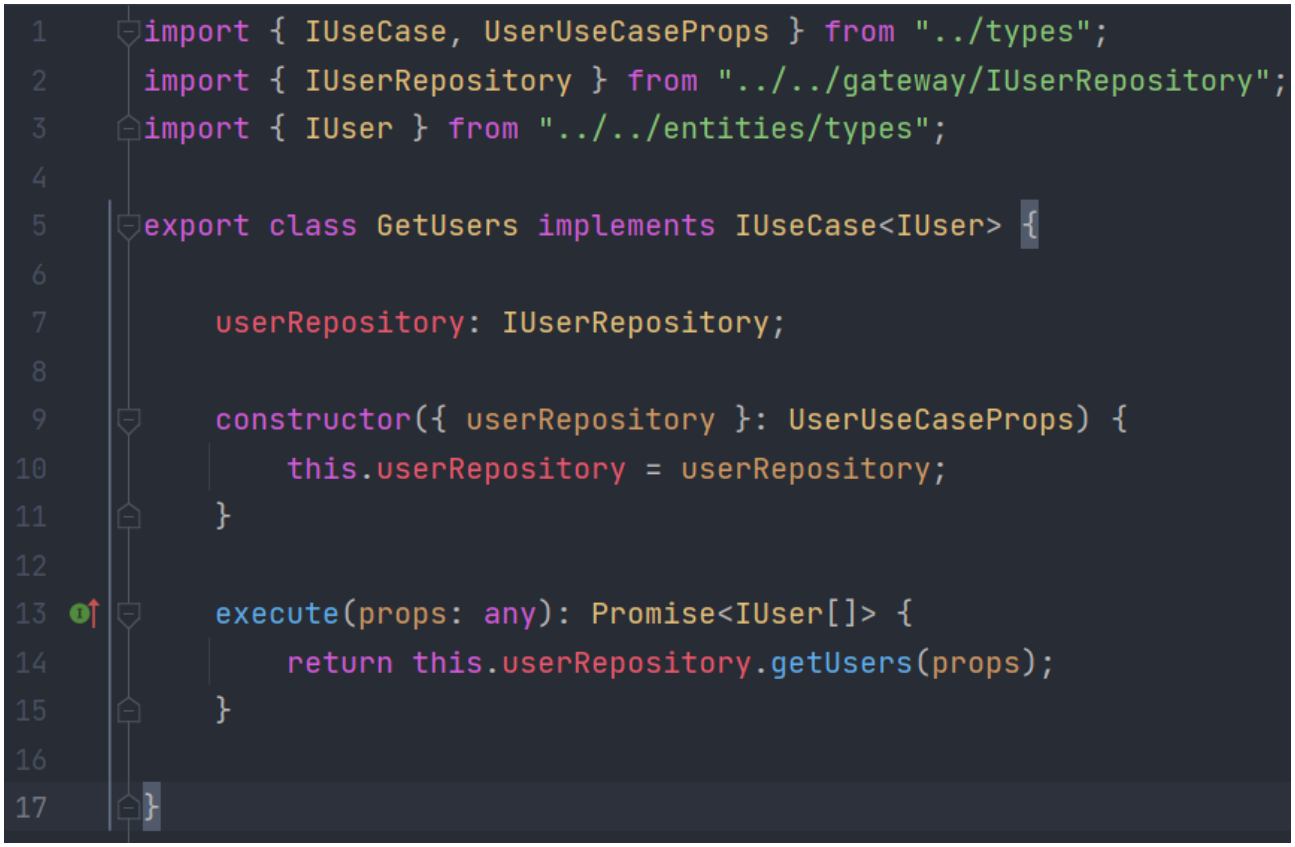
Додаток Д.3

Відправка запитів до інших мікросервісів

```
export async function createUsers(user: IUser) {
    let err;
    [err] = await to<any>(axios.post(
        url: `http://localhost:${process.env.CONSULTATION_SERVICE_PORT}/consultations/users`, data: {
            ...user,
            userID: user._id,
        })
    );
    if (err) throw err;

    [err] = await to<any>(axios.post(
        url: `http://localhost:${process.env.POST_SERVICE_PORT}/posts/users`, data: {
            ...user,
            userID: user._id,
        })
    );
    if (err) throw err;

    [err] = await to<any>(axios.post(
        url: `http://localhost:${process.env.RECOMMENDATION_SERVICE_PORT}/recommendations/user-tags`, da
            userID: user._id,
            tags: [],
        })
    );
    if (err) throw err;
```

Додаток Е.1

Приклад реалізації випадку використання



```
import { IUseCase, UserUseCaseProps } from "../types";
import { IUserRepository } from "../../gateway/IUserRepository";
import { IUser } from "../../entities/types";

export class GetUsers implements IUseCase<IUser> {

    userRepository: IUserRepository;

    constructor({ userRepository }: UserUseCaseProps) {
        this.userRepository = userRepository;
    }

    execute(props: any): Promise<IUser[]> {
        return this.userRepository.getUsers(props);
    }

}
```

Додаток Е.2

Приклад реалізації репозиторію

```
export class RatingRepository implements IRatingRepository {

    RatingModel: IRatingModel;

    constructor({ RatingModel }: RatingRepositoryProps) {
        this.RatingModel = RatingModel;
    }

    async createRating(ratingProps: IRating): Promise<Rating> {
        const [err, rating] = await to<IRating>(new this.RatingModel( doc: {
            ...ratingProps
        }).save());

        if (err) throw err;

        return this.RatingModel.toRating(rating);
    }

    async deleteRating(ratingID: string): Promise<Rating> {
        return this.RatingModel.toRating(await this.RatingModel.findByIdAndRemove(ratingID));
    }

    async getRatingById(ratingID: string): Promise<Rating> {
        return this.RatingModel.toRating(await this.RatingModel.findById(ratingID));
    }

    async getRatings(filter?: any): Promise<Rating[]> {
        const ratings = await this.RatingModel.find(filter);
        return ratings.map(rating => this.RatingModel.toRating(rating));
    }

    async updateRating(ratingID: string, updateProps: any): Promise<Rating> {
        return this.RatingModel.toRating(await this.RatingModel.findByIdAndUpdate(ratingID, updateProps));
    }
```

Додаток Е.3

Приклад створення та налаштування екземпляру Express

```
export class ExpressConfig {
    app: express.Express;
    container: AwilixContainer;

    constructor(connection: mongoose.Connection) {
        this.app = express();

        this.app.use(cors( options: {exposedHeaders: 'Authorization'}));
        this.app.use(bodyParser.json());
        this.app.use(bodyParser.urlencoded( options: {extended: false}));

        this.container = makeContainer(connection);
        this.app.use(scopePerRequest(this.container));

        this.setUpControllers();
    }

    setUpControllers() {
        const env = process.env.ENVIRONMENT;
        const controllerPath =
            env === 'PROD'
                ? path.resolve( pathSegments: 'dist', 'infrastructure', 'controllers')
                : path.resolve( pathSegments: 'src', 'infrastructure', 'controllers');

        const extension = env === 'PROD' ? '/*.js' : '/*.ts';

        useExpressServer(this.app,  options: {
            controllers: [controllerPath + extension],
            middlewares: [ErrorMiddleware],
            authorizationChecker: checkRole,
            // defaultErrorHandler: false
        });
    }
}
```

Додаток Ж.1

Приклад оновлення токену

```
export default class JwtMiddleware {
    static token = '';

    use(req: Request, res: Response, next: NextFunction) {

        const token = JwtMiddleware.extractToken(req);

        let jwtPayload: any;

        try {
            jwtPayload = jwt.verify(token, process.env.JWT_SECRET);
        } catch (e) {
            res.status( code: 401).json( body: {
                msg: 'You should be logged in to access this url'
            })
            return;
        }

        let data;

        if (typeof jwtPayload == 'object') {
            data = jwtPayload.data;
        } else {
            throw new Error('JwtToken has no payload');
        }

        const newToken = jwt.sign( payload: {
            data: { userID: data.userID, email: data.email, role: data.role }
        }, process.env.JWT_SECRET,  options: { expiresIn: process.env.JWT_EXPIRATION });

        req.cookies.token = newToken;

        res.setHeader( name: 'Authorization',  value: 'Bearer ' + newToken);

        next();
    }
```

Додаток Ж.2

Приклад перевірки авторизації

```
13  export async function checkRole(action: Action, roles: string[]) {
14      const req: ContainerReq = action.request;
15
16      const token = extractToken(req);
17
18      if (!token) return false;
19
20      const jwtPayload: any = jwt.verify(token, process.env.JWT_SECRET);
21      const userID = jwtPayload.data.userID;
22
23      const { getAccounts }: checkRoleProps = req.container.cradle;
24      const [err, accounts] = await to<IAccount[]>(getAccounts.execute( props: {userID}));
25
26      if (err || !accounts[0]) return false;
27      const account = accounts[0];
28
29      return roles.indexOf(account.role) > -1;
30  }
```