ДВНЗ «ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ»
Факультет комп'ютерно-інформаційних технологій та автоматизації
Кафедра прикладної математики і інформатики

«До захисту допущено»
Завідувачка кафедри ПМІ
Ольга ДМИТРІЄВА
«___» _____ 2022 р.

Випускна кваліфікаційна робота
бакалавр
(освітній ступень)

на тему:
«Розробка мобільних мікросервісів для контролю персональних даних користувачів»
зі спецчастиною:
«Розробка серверної частини, програмних модулів, баз даних засобами TypeScript, Node.js, PostgreSQL»

Виконав: студент 4-го курсу, групи ІПЗз-18

спеціальності 121 Інженерія програмного забезпечення

Олександр ГРИЦЕНКО _____
(ім'я та прізвище) (підпис)

Керівник зав. каф. ПМІ, д.т.н., проф. Ольга ДМИТРІЄВА _____
(посада, науковий ступінь, вчене звання, ім'я та прізвище) (підпис)

Рецензент зав. каф. КІ, к.т.н., доц. Сергій КОВАЛЬОВ _____
(посада, науковий ступінь, вчене звання, ім'я та прізвище) (підпис)

Нормоконтроль:
_____ Ірина НАЗАРОВА
(підпис)

Дата _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Покровськ, Луцьк – 2022 р.

ДВНЗ «Донецький національний технічний університет»
Факультет комп'ютерно-інформаційних технологій та автоматизації_____
Кафедра прикладної математики та інформатики_____
Освітньо-кваліфікаційний рівень (освітній ступінь) бакалавр_____
Напрям підготовки (спеціальність) 121 Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Завідувачка кафедри
Ольга ДМИТРІЄВА
/_____/
"____" _____2022 р.

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Гриценко Олександр Сергійович

1. Тема роботи «Розробка мобільних мікросервісів для контролю персональних даних користувачів»

Спецчастина «Розробка серверної частини, програмних модулів, баз даних засобами TypeScript, Node.js, PostgreSQL»

Керівник роботи Дмитрієва Ольга Анатоліївна, д.т.н, проф., зав. каф._ПМІ

затверджені наказом від "29" квітня 2022 року № 163

2. Строк подання студентом роботи_____

3. Вихідні дані до роботи результати науково-дослідної роботи, матеріали переддипломної практики

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити: 1) аналіз предметної області і постановка завдання; 2) проєктування програмної системи; 3) розробка серверної частини, спрямованої на збереження даних користувачів під час реєстрації, авторизації з подальшим розширенням функціоналу; 4) реалізація засобів безпеки при функціонуванні системи; 5) тестування програмної системи.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) 1) блок-схеми розроблених сервісів; 2) діаграми таблиць даних; 3) екранні форми.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 07.04.2022

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Об’єктний аналіз предметної області і постановка завдання	07.04.2022	
2	Розробка моделі взаємодії даних	20.04.2022	
3	Проєктування бази даних	01.05.2022	
4	Проєктування програмної системи	03.05.2022	
5	Програмування модулів інтерфейсу	07.05.2022	
6	Розробка програмного засобу	13.05.2022	
7	Тестування програми й аналіз	27.05.2022	
8	Оформлення пояснювальної записки	01.06.2022	
9	Підготовка слайдів презентації	05.06.2022	
10	Підготовка демонстраційної версії розробленого програмного продукту	10.06.2022	
11	Написання доповіді	12.06.2022	

Студент _____ Олександр ГРИЦЕНКО
(підпис) (ім’я, прізвище)

Керівник роботи _____ Ольга ДМИТРІЄВА
(підпис) (ім’я, прізвище)

АНОТАЦІЯ

Олександр ГРИЦЕНКО. Розробка мобільних мікро сервісів для контролю персональних даних користувачів / Випускна кваліфікаційна робота на здобуття освітньо-кваліфікаційного рівня «бакалавр» за спеціальністю 121 Інженерія програмного забезпечення. – ДВНЗ ДонНТУ, Луцьк, 2022.

Об’єктом дослідження є процес збереження персональних даних користувачів при вирішенні завдання побудови серверної частини з використанням мікросервісної архітектури.

Предметом дослідження є серверна частина на основі мікросервісів, орієнтована на безпечне збереження даних користувачів за умови процесу реєстрації та авторизації у системі мобільного додатку.

Мета роботи полягає у розробці серверної частини для безпечного та зручного збереження даних користувачів з подальшим простим розширенням розробленої системи.

Методи розробки базуються на використанні основних положень мікросервісної архітектури, дизайну сервісів, структур і баз даних, проєктування програмного забезпечення.

Практичне значення полягає в полегшенні розширення серверної частини мобільних та веб додатків, підвищенні ефективності збереження даних та покращенні між сервісної комунікації.

Ключові слова: мікросервісна архітектура, API Gateway, Node.js, СКБД PostgreSQL, мова TypeScript

ANNOTATION

Oleksandr HRYTSENKO. Development of mobile microservices for control personal data of users / Final qualification work for obtaining the educational qualification level "bachelor" in the specialty 121 Software Engineering. - SHEI DonNTU, Lutsk, 2022.

The object of research is the process of storing personal data of users when solving the problem of building a server part using microservice architecture.

The subject of the study is a server part based on microservices, focused on secure storage of user data under the process of registration and authorization in the mobile application system.

The purpose of the work is to develop a server part for secure and convenient storage of user data with the subsequent simple expansion of the developed system.

Development methods are based on the use of the basic provisions of microservice architecture, design of services, structures and databases, software design.

Of practical importance is to facilitate the expansion of the server part of mobile and web applications, increase the efficiency of data storage and improve inter-service communication.

Keywords: microservice architecture, Gateway API, Node.js, PostgreSQL database, TypeScript language

ЗМІСТ

ВСТУП

Особисті дані в Інтернеті рідко, якщо взагалі коли-небудь, ефективно контролюються користувачами, яких вони стосуються. Якщо подумати, будь-який контроль, який користувачі можуть мати над своїми особистими даними, часто є занадто грубим (як правило, пропозиція все або нічого), заплутаним (оскільки він залежить від кожної організації, яка зберігає та обробляє дані) і застосовується вручну відповідно до бажання або навіть примха кожної організації.

Збереження даних, особливо персональних, являється не простим завданням. Особливо складною є задача вибору архітектури проєктування, яка буде основою для майбутнього проєкту. У даному випадку було вибрано мікросервісну архітектуру, яка є типом сервісно-орієнтованої архітектури, та є тією архітектурою в якій можливість незалежного розгортання є ключовою та не залежить від технологій.

Незважаючи на те, що мікросервісна архітектура має багато плюсів та стає все більш популярною, я вважаю, що її використання вимагає ретельного обдумування. Потрібно оцінити кожен недолік, персональні та командні навички, технологічні основи і зрозуміти, чого потрібно досягти перш ніж вирішити, чи підходять мікросервіси.

Метою розробки є створення серверної частини для контролю персональних даних користувачів на основі мікросервісної архітектіри.

В процесі розробки необхідно вирішити наступні задачі:

а) зробити огляд мікросервісної архітектури;

б) визначити необхідні функції для реалізації серверної частини;

в) вибрати необхідний технологічний стек для реалізації серверу та реалізувати серверну частину;

г) провести тестування розробленої системи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА СПЕЦИФІКАЦІЇ ВИМОГ МІКРОСЕРВІСІВ ДЛЯ КОНТРОЛЮ ПЕРСОНАЛЬНИХ ДАНИХ КОРИСТУВАЧІВ

У сучасному світі є величезна кількість програм, мобільних та веб додатків, систем тощо, які потребують аутентифікації та збереження даних користувачів. Важливість конфіденційності в цифровому світі змусила цілі країни переоцінити способи, якими організації обробляють персональні дані. Типовим прикладом є Загальний регламент захисту даних (GDPR), складений Європейським Союзом.

У той же час виникає багато складних проблем під час розробки розподілених систем, однією з проблем є те, як реалізувати гнучку, безпечну та ефективну схему аутентифікації, авторизації та яку саме архітектуру слід обрати.

За основу майбутньої серверної частину було вибрану архітектуру, яка базується цілком на мікросервісах. Архітектура мікросервісів дає багато переваг програмним додаткам, включаючи невеликі групи розробників, коротші цикли розробки, гнучкість вибору мови та покращену масштабованість сервісу. Але перед тим, як розробляти проєкт та використовувати той, чи інший архітектурний підхід, слід детальніше розібратися з технологією. В цьому розділі буде обговорена мікросервісна архітектура та її можливі недоліки, основні концепції, переваги.

1.1 Огляд концепції мікросервісної архітектури

Мікросервіси — це сервіси, які можна самостійно випускати, які моделюються навколо бізнесу домену [14]. Сервіс інкапсулює функціональність і робить її доступною для інших послуги через мережі — ви будуєте більш складну систему з цієї будівлі блоків. Один мікросервіс може представляти інвентар, інший – керування замовленнями та ще один

доставкою, але разом вони можуть становити повноцінну систему електронної комерції.

Ідея полягає в тому, щоб розділити вашу програму на набір менших взаємопов’язаних служб замість того, щоб створювати єдиний монолітний додаток. Кожен мікросервіс — це невелика програма, яка має власну гексагональну архітектуру, що складається з бізнес-логіки разом із різними адаптерами. Деякі мікросервіси надають REST, RPC або API на основі повідомлень, а більшість служб використовують API, надані іншими службами. Інші мікросервіси можуть реалізувати веб-інтерфейс [14].

Шаблон архітектури мікросервісу суттєво впливає на відносини між програмою та базою даних. Замість того, щоб спільно використовувати одну схему бази даних з іншими службами, кожна служба має свою власну схему бази даних. З одного боку, такий підхід суперечить ідеї моделі даних для всього підприємства. Крім того, це часто призводить до дублювання деяких даних. Однак наявність схеми бази даних для кожної служби є важливою, якщо потріно отримати вигоду від мікросервісів, оскільки вона забезпечує слабке з’єднання. Кожен із сервісів має свою базу даних. Більше того, служба може використовувати тип бази даних, який найкраще відповідає її потребам, так звану архітектуру збереження поліглотів.

На (рис. 1.1) [17] зображено приклад програми для електронної комерції, яка приймає замовлення від клієнтів, перевіряє запаси та наявний кредит і відправляє їх. Додаток складається з кількох компонентів, включаючи StoreFrontUI, який реалізує інтерфейс користувача, а також деякі серверні служби для перевірки кредиту, підтримки запасів і замовлень на доставку. Додаток складається з набору сервісів.

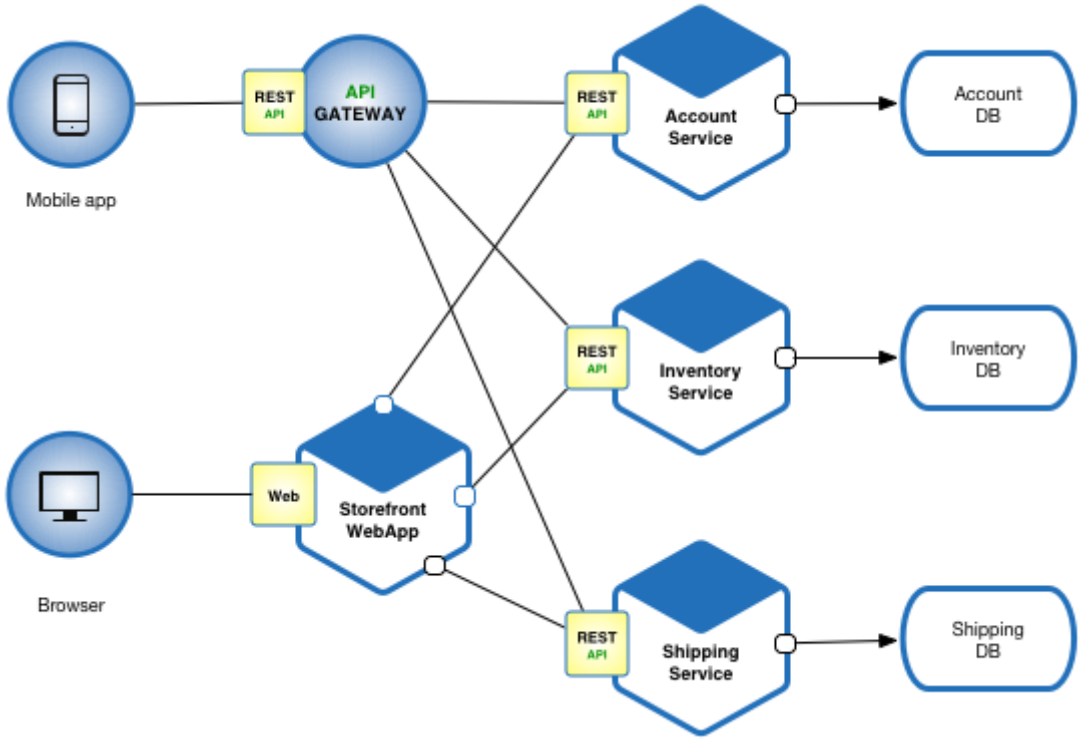


Рисунок 1.1 – Приклад архітектури електронної комерції з використанням мікросервісів

Найчастіше мікросервіси обговорюються як архітектурний підхід, який є альтернативою монолітній архітектурі. Щоб краще розрізняти архітектуру мікросервісів потрібно трохи описати, що саме мається на увазі під монолітами.

Якщо говорити по простому, то коли всі функції в системі повинні бути розгорнуті разом, то можна назвати це монолітом. Можливо, кілька архітектур відповідають цьому визначенню, але є ті як найчастіше використовуються: моноліт з одним процесом, модульний моноліт, і модульний моноліт з розподіленою системою баз даних [11].

Найпоширенішим прикладом, який спадає на думку під час обговорення монолітів, є система в якому весь код розгорнуто як єдиний процес, як на рисунку 1.2. Можна мати кілька екземплярів цього процесу з міркувань надійності або масштабування, але принципово весь код упакований в один процес. Насправді ці одно-процесні системи можуть бути простими розподіленими системами самостійно, оскільки вони майже завжди в

кінцевому підсумку зчитують дані або зберігають дані в базах даних, або представляють інформацію до веб чи мобільних додатків.

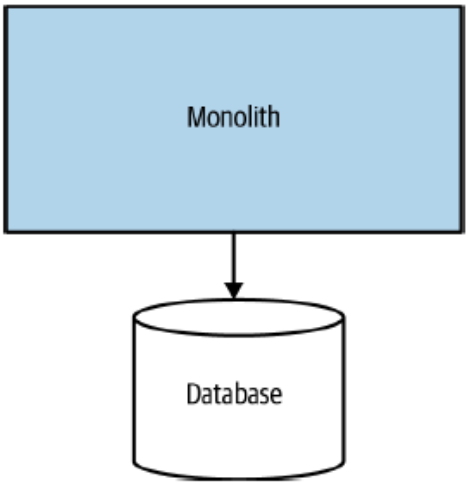


Рисунок 1.2 – Найпоширеніший вид монолітної архітектури

Як підмножина моноліту з одним процесом, модульний моноліт є варіацією в якій єдиний процес складається з окремих модулів. З кожним модулем можна працювати окремо, але все ще потрібно об’єднувати систему разом для розгортання, як показано на рисунку 1.3. Концепція розбиття програмного забезпечення на модулі не є новою; модульне програмне забезпечення має свої коріння в роботі, що виконується навколо структурного програмування в 1970-х і навіть раніше [14].

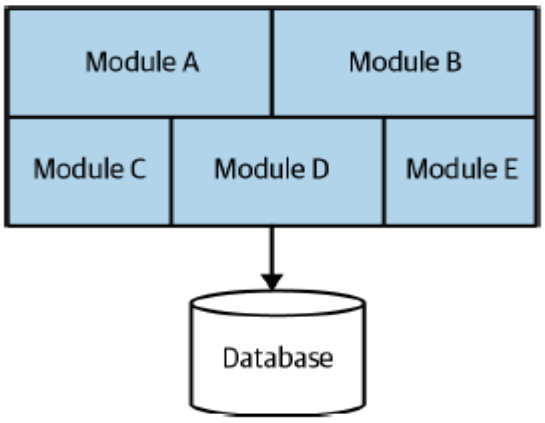


Рисунок 1.3 – У модульному моноліті код всередині процесу розділений на модулі

Для багатьох організацій модульний моноліт може стати відмінним вибором. Якщо межі модуля чітко визначені, це може забезпечити високий ступінь паралельної роботи, уникаючи проблем з більш розподіленою архітектурою мікросервісів, завдяки набагато простішою топологією розгортання. Shopify є чудовим прикладом організації, яка використовувала цю техніку як альтернативу мікросервісної декомпозиції, і, здається, це дуже добре працює для цієї компанії. Однією з проблем модульного моноліту є те, що в базі даних, як правило, не вистачає декомпозиції, яку ми знаходимо на рівні коду, що призводить до значних проблем, якщо хочете більш розбити моноліт у майбутньому. Деякі команди намагаються підштовхнути ідею модульного моноліту, де проходить декомпозиція баз даних (рис. 1.4).

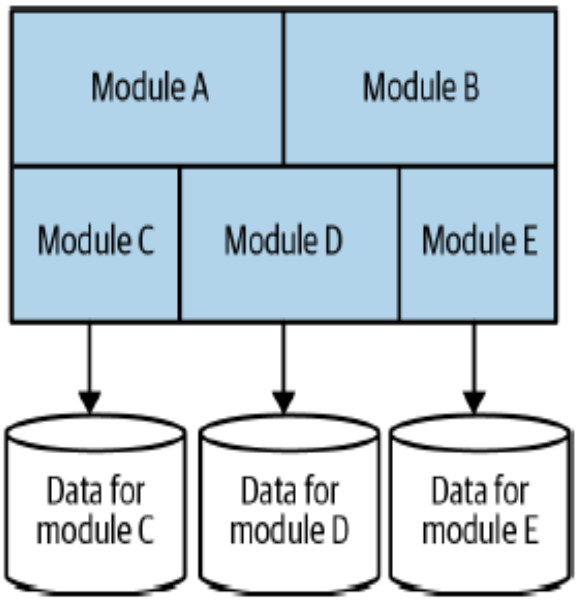


Рисунок 1.4 – Модульний моноліт з розподіленою базою даних

1.2 Ключові концепції мікросервісної архітектури

Вивчаючи мікросервіси, необхідно зрозуміти кілька основних ідей. Оскільки деякі аспекти часто не помічаються, важливо дослідити ці поняття, щоб переконатися, що саме змушує мікросервіси працювати і бути популярними.

1.2.1 Незалежне розгортання

Незалежне розгортання – це ідея того, що ми можемо внести зміни в мікросервіс, розгорнути його та передати ці зміни нашим користувачам без необхідності розгортати будь-які інші мікросервіси. Що ще важливіше, це не лише те, що ми можемо це зробити, важливо, що ми можемо керувати розгортанням у своїй системі.

Щоб забезпечити незалежне розгортання, потрібно переконатися, що розроблені мікросервіси слабо пов’язані: потрібно мати можливість змінити один сервіс, не змінюючи іншого. Це означає, що потрібні чітко визначені та стабільні контракти між цими сервісами. Деякі варіанти реалізації ускладнюють це — спільне використання баз даних, наприклад, особливо проблематичне.

1.2.2 Сформовано навколо бізнес-домену

Такі методи, як предметно-орієнтоване проєктування (domain-driven design), можуть дозволити краще структурувати код, щоб представити реальну картину бізнесу, в якому працює програмне забезпечення. З мікросервісною архітектурою, використовується та сама ідея, щоб визначити межі сервісів. Моделювання сервісів навколо бізнес-доменів, дозволяє полегшити розгортання нових функцій і рекомбінувати мікросервіси різними способами, щоб надати нову функціональність користувачам [12].

Розгортання якогось нового функціонала, який вимагає внесення змін до біль ніж одного мікросервіса, коштує дорого. Потрібно координувати роботу в кожному сервісі та, можливо, в окремих командах, а також ретельно керувати порядком, у якому нові версії цих сервісів розгорнуті. Це вимагає набагато більше роботи, ніж зробити ту саму зміну всередині одного сервісу. Звідси випливає, що потріно знайти способи зробити зміни між сервісами якомога рідшими.

Наприклад на рисунку 1.5. зображена стандартна триярусна архітектура [14]. Тут кожен рівень представляє різні межі сервісів, з кожною межею на основі відповідної технічної функціональності. Як показує практика зміни функціональності в одному рівні зазвичай охоплюють декілька рівнів у подібних типах архітектури.

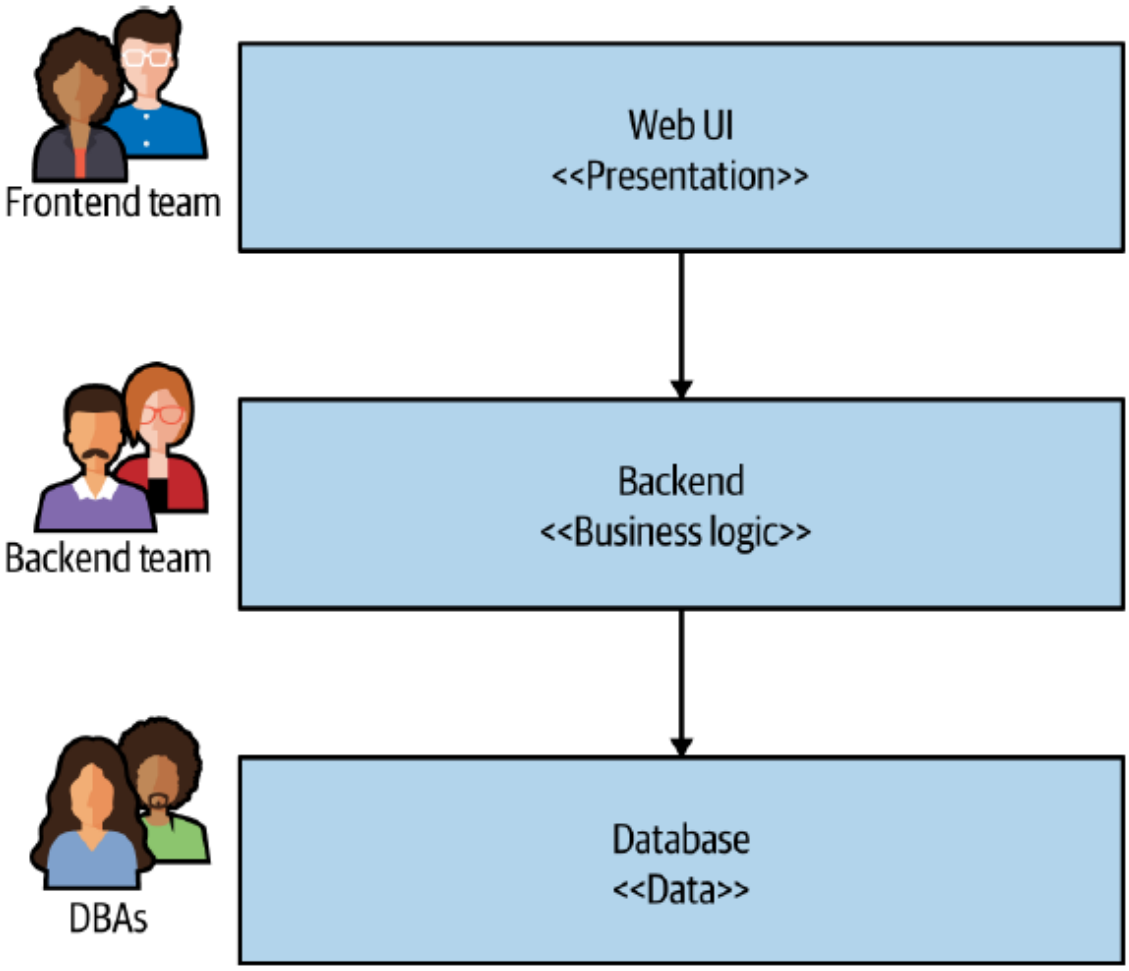


Рисунок 1.5 – Традиційна триярусна архітектура

1.2.3 Володіння власним станом

Одна з ідей мікросервісів, з якою людям найважче, це те, що слід уникати використання спільних баз даних. Якщо мікросервіс хоче отримати доступ до даних іншого мікросервіса, він повинен піти та запитати другий мікросервіс для ці дані. Це дає мікросервісам можливість вирішувати, що являється спільним, а що прихованим, що в свою чергу дозволяє чітко відокремити функціональні можливості, які можуть вільно змінюватися

(внутрішня реалізація) від функціональності, яку хочеться нечасто змінювати (зовнішній функціонал, яким користуються користувачі).

Таким чином, маючи чітке розмежування між внутрішніми деталями реалізації та зовнішнім контрактом для мікросервісу можна досягти зменшення потреб в змінах, несумісних зі зворотним зв’язком.

1.3 Переваги та недоліки мікросервісної архітектури

Історія та походження мікросервісів – це постійні зусилля щодо забезпечення кращого зв’язку між різними платформами, більшої простоти та більш зручних систем. Мікросервіси зазвичай розглядають як техніку розробки програмного забезпечення, яка організовує додаток як групу слабо пов’язаних служб [15].

Мікросервісна архітектура може надати величезну гнучкість у виборі технології, управління надійністю та масштабуванням, організації команд тощо. Ця гнучкість і є причиною чому багато людей використовують цю архітектуру. Але мікросервіси приносять з собою значний ступінь складності, хоча потрібно зауважити, що ця складність є виправданою. Для багатьох розробників мікросервіси стали системною архітектурою за замовчуванням, яку використовують практично в усіх ситуаціях. Багато організацій, особливо великі (Uber, Amazon, Netflix, Airbnb, Coca-Cola), показали, наскільки ефективними мікросервіси можуть бути [10]. Коли основні концепції мікросервісів правильно зрозумілі і реалізовані, вони можуть допомогти створити потужні, продуктивні архітектури. Однак як і різні архітектурні підходи – мікросервісний підхід має як свої плюси, так і мінуси.

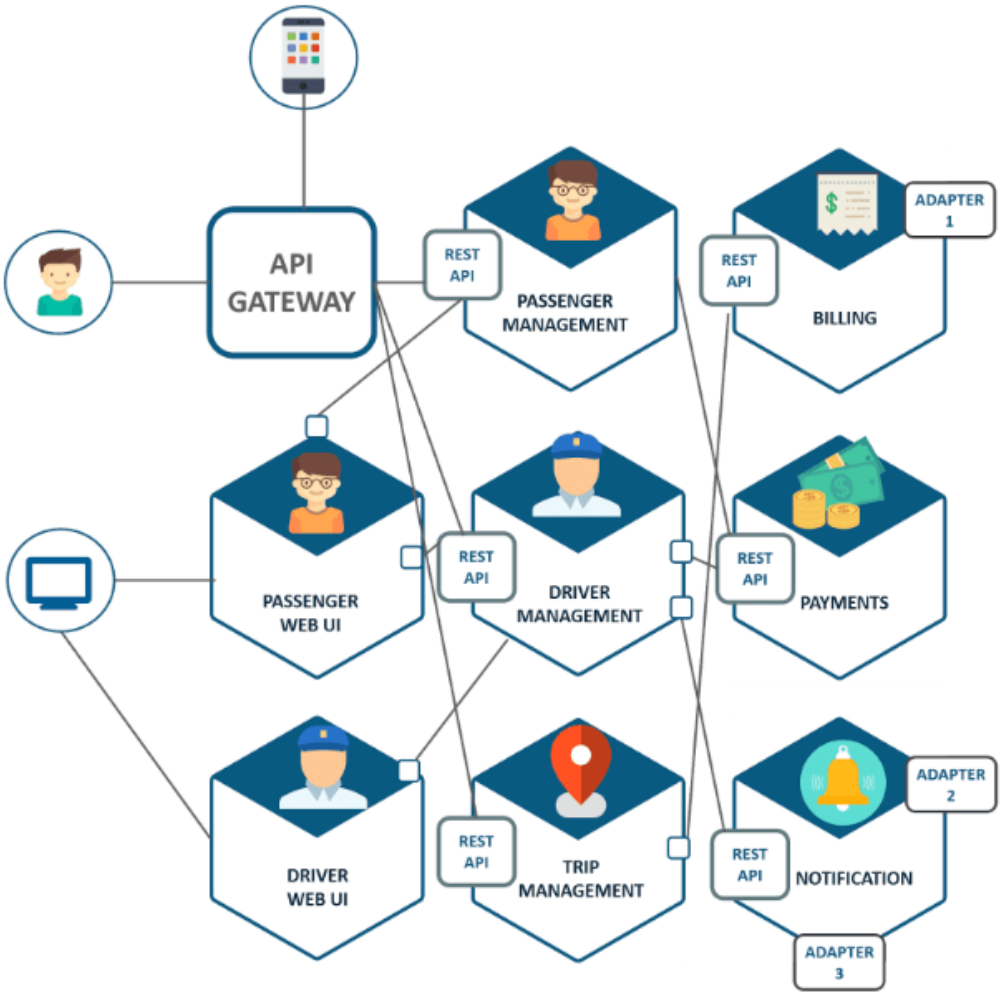


Рисунок 1.6 – Мікросервісна архітектура Uber [10]

1.3.1 Переваги мікросервісної архітектури

Основні переваги які присутні при використанні мікросервісів:

а) вирішує проблему складності, розкладаючи систему на набір керованих служб, які набагато швидше розробляються та набагато легші для розуміння та підтримки;

б) дає змогу окремо розробляти кожен функціонал командою, яка зосереджена на цьому сервісі;

в) зменшує бар’єр прийняття нових технологій, оскільки розробники можуть вільно вибирати будь-які технології, які мають сенс для їх

обслуговування, і не обмежені вибором, зробленим на початку проєкту (рис.1.3);

г) дозволяє розгортати кожен мікросервіс незалежно. В результаті це робить можливим безперервне розгортання для складних додатків;

д) дозволяє незалежно масштабувати кожен сервіс (рис. 1.4).

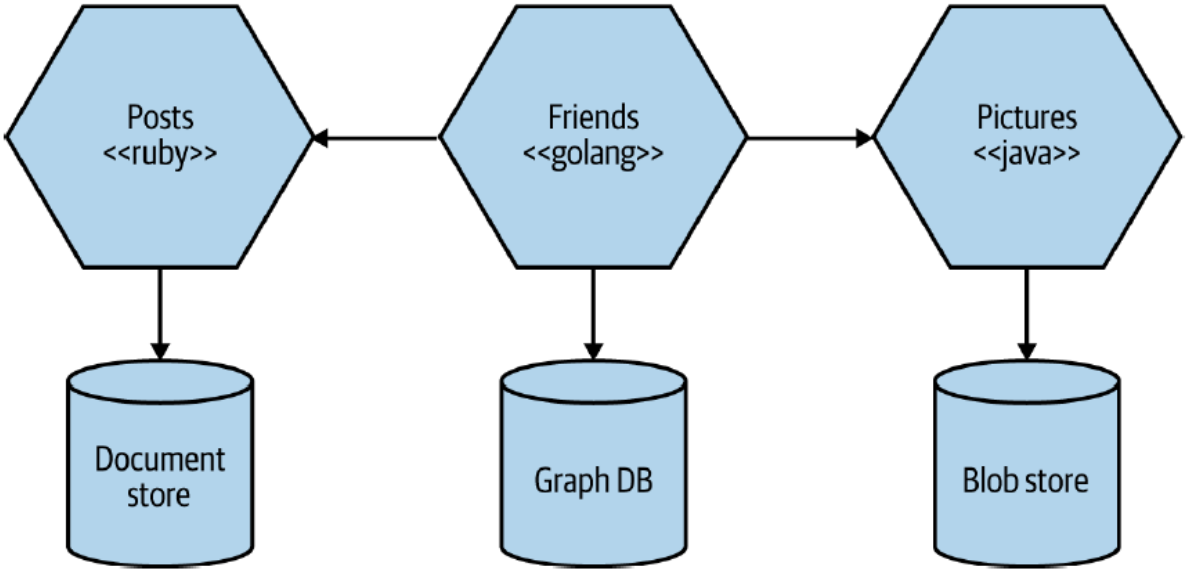


Рисунок 1.7 – Можливість використовувати різні технології [14]

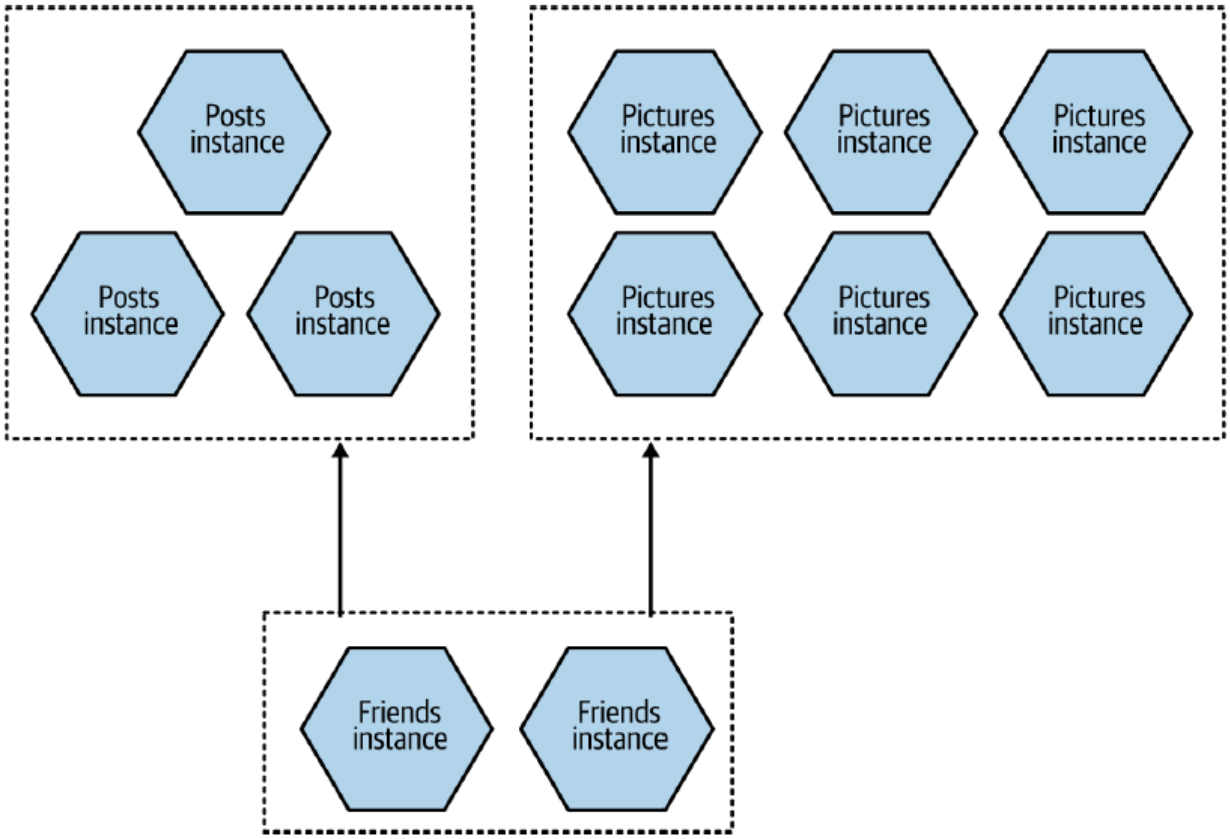


Рисунок 1.8 – Незалежне друг від другу масштабування сервісів [14]

1.3.2 Недоліки мікросервісної архітектури

а) ускладняється проєкт лише через те, що програма зроблена з використанням цієї архітектури є розподіленою системою. Вам потрібно вибрати і реалізувати механізм між процесного зв’язку, заснований на обміні повідомленнями або RPC, і написати код для обробки часткових збоїв і врахування інших помилок розподілених обчислень;

б) роздільна архітектура баз даних. Бізнес-транзакції, які оновлюють кілька бізнес-суб’єктів у програмі на основі мікросервісів, повинні оновлювати декілька баз даних, що належать різним службам. Використання розподілених транзакцій зазвичай не є варіантом, і в кінцевому підсумку вам доведеться використовувати підхід, заснований на узгодженості, що є більш складним для розробників;

в) тестування програми мікросервісів також набагато складніше, ніж у випадку монолітного проєкту. Для подібного тесту для сервісу вам

потрібно буде запустити цей сервіс та всі сервіси, від яких він залежить (або принаймні налаштувати заглушки для цих сервісів);

г) складніше впровадити зміни, які охоплюють декілька сервісів. У монолітній програмі можна просто змінити відповідні модулі, інтегрувати зміни та розгорнути їх за один раз. В архітектурі мікросервісів потрібно ретельно спланувати та координувати впровадження змін до кожного сервісу;

д) розгортання програми на основі мікросервісів також є більш складним. Мікросервісна програма зазвичай складається з великої кількості мікросервісів. Кожен сервіс матиме кілька екземплярів виконання. І кожен екземпляр потрібно налаштувати, розгорнути, масштабувати та відстежувати. Крім того, вам також потрібно буде реалізувати механізм виявлення сервісу. Ручні підходи до операцій не можуть масштабуватися до такого рівня складності, а успішне розгортання програми мікросервісів вимагає високого рівня автоматизації.

1.4 Висновки за розділом

Проведено аналіз предметної області та специфікації мікросервісів для контролю персональних даних користувачів. Розглянуті основні концепції мікросервісної архітектури та її відмінності від монолітної системи. Проаналізовано переваги та недоліки мікросервісної архітектури.

2 ПРОЄКТУВАННЯ МІКРОСЕРВІСІВ ДЛЯ КОНТРОЛЮ ПЕРСОНАЛЬНИХ ДАНИХ КОРИСТУВАЧІВ

Подібно до того, як робітник ніколи не почне будувати будинок без креслення, не слід починати кодування без детального, письмового проєкту для майбутньої програми. Якщо не знати передбачуваного результату програми, то витратиться багато часу на написання заплутаного коду, який не досягає певної мети. Проблема, з якою можна часто стикатися це те, що важко втриматися від спокуси взяти в руки клавіатуру, почати створювати класи, функції тощо. Через кілька годин неминуче можна отримати багато брудного коду та не буде чітких результатів, проблему, яку можна було б уникнути, приділивши час плануванню програми.

Кожен програмний проєкт, який планується — має починатися з визначення проблеми: високорівневої постановки проблеми, яку потрібно вирішити.

2.1 Розробка архітектури програмної системи

В якості серверної частини було вирішено використати Node.js. Node.js — це кросплатформне з відкритим вихідним кодом середовище виконання JavaScript. Це популярний інструмент практично для будь-яких проєктів.

Node.js працює на двигуні JavaScript V8, ядро Google Chrome, поза браузером. Це дозволяє Node.js бути дуже продуктивним. Програма Node.js працює в одному процесі, не створюючи новий потік для кожного запиту. Node.js містить у своїй стандартній бібліотеці набір асинхронних примітивів введення-виводу, які запобігають блокування коду JavaScript, і загалом бібліотеки в Node.js написані з використанням неблокуючих парадигм, що робить поведінку блокування скоріше винятком, ніж нормою [2].

Коли Node.js виконує операцію введення-виведення, наприклад, читання з мережі, доступ до бази даних або файлової системи, замість того,

щоб блокувати потік і витрачати цикли ЦП на очікування, Node.js відновить операції, коли відповідь повернеться.

Це дозволяє Node.js обробляти тисячі одночасних підключень з одним сервером, не вводячи тягар керування паралельністю потоків, що може бути значним джерелом помилок.

Node.js має унікальну перевагу, оскільки мільйони розробників інтерфейсу, які пишуть JavaScript для браузера, тепер можуть писати код на стороні сервера на додаток до коду на стороні клієнта без необхідності вивчати зовсім іншу мову.

В якості мови програмування вирішено використовувати TypeScript — це підмножина JavaScript, яка має типизацію і компілюється у звичайний JavaScript. Простішими словами, TypeScript технічно є JavaScript зі статичною типізацією, коли ви хочете це мати [20].

Для проєктування системи авторизації та збереження даних користувачів вирішено розробити два основні мікросервіси:

а) сервіс авторизації (auth microservice) буде виконувати всі необхідні дії для збереження основної інформації про користувачів та всіх авторизаційних процесів;

б) сервіс профілю (profile/user microservice). Сервіс для збереження додаткової інформації користувача.

Для спілкування між сервісами буде використовуватися брокера повідомлень, а саме NATS.

NATS — це технологія зв’язку, створена для світу, що стає все більш гіперпов’язаним. Це єдина технологія, яка дає змогу програмам безпечно спілкуватися через будь-яку комбінацію хмарних постачальників, локальних, периферійних, веб та мобільних пристроїв. NATS складається з сімейства продуктів з відкритим вихідним кодом, які тісно інтегровані, але можуть бути розгорнуті легко й незалежно [1]. NATS використовується в усьому світі тисячами компаній, охоплюючи варіанти використання, включаючи мікросервіси, периферійні обчислення, мобільні пристрої і може

використовуватися для розширення або заміни традиційного обміну повідомленнями.

Сервер NATS діє як центральна нервова система для створення розподілених додатків. Клієнтські API надаються більш ніж 40 мовами та фреймворками, включаючи Go, Java, JavaScript/TypeScript, Python, Ruby, Rust, C#, C і NGINX. Потокова передача даних у реальному часі, високостійке зберігання даних і гнучкий пошук даних підтримуються через JetStream, платформу потокової передачі наступного покоління, вбудовану в сервер NATS.

Отже між сервісами вирішено яке буде спілкування, тепер важливо також визначити спілкування з клієнтською частиною майбутнього проєкту. Для мобільних додатків, або інтернет сайтів потрібна точка доступу до наших серверів. Тому в якості такою точки доступу буде інший мікросервіс API Gateway.

API Gateway, шлюз-система, яка є загальною точкою входу в сучасних програмах, що працюють через API. Причому це можуть бути як монолітні програми, так і програми на основі мікросервісів (рис. 2.1) [22].

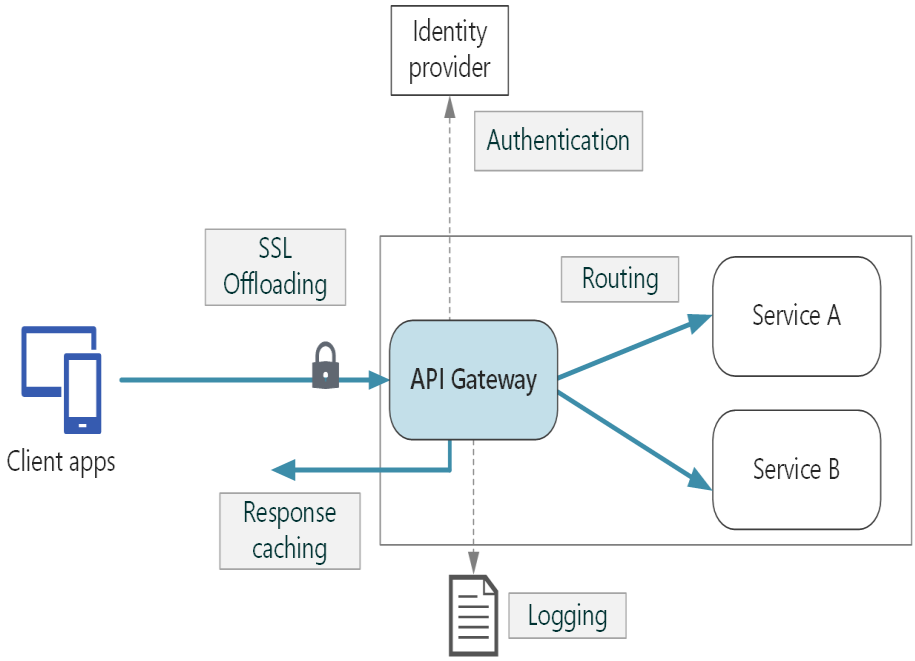


Рисунок 2.1 – API Gateway — один із основних шаблонів проєктування, навколо якого будується мікросервісна архітектура.

API Gateway реалізація переслідує низку цілей:

а) У зовнішній світ є тільки ода адреса, але так як існує багато мікросервісів, необхідно перенаправляти запити на екземпляри цих мікросервісів.

б) Побудова бекенд для фронтенда односторінкових додатків.

в) Спосіб поділу клієнтського програмного інтерфейсу від внутрішньої реалізації. Коли клієнт робить запит, шлюз API автоматично розбиває його на кілька запитів, надсилає їх у потрібні місця, видає відповідь та все відстежує.

г) Агрегація.

д) Кешування.

е) Безпека - одне з найважливіших завдань, які вирішує API Gateway, тому що ця система стоїть на вході в нашу інфраструктуру і саме на кордоні вирішується питання безпеки, після чого настає справа з довіреною зоною (DMZ).

Отже, майже повністю вирішено як буде виглядати майбутня система (рис. 2.2).

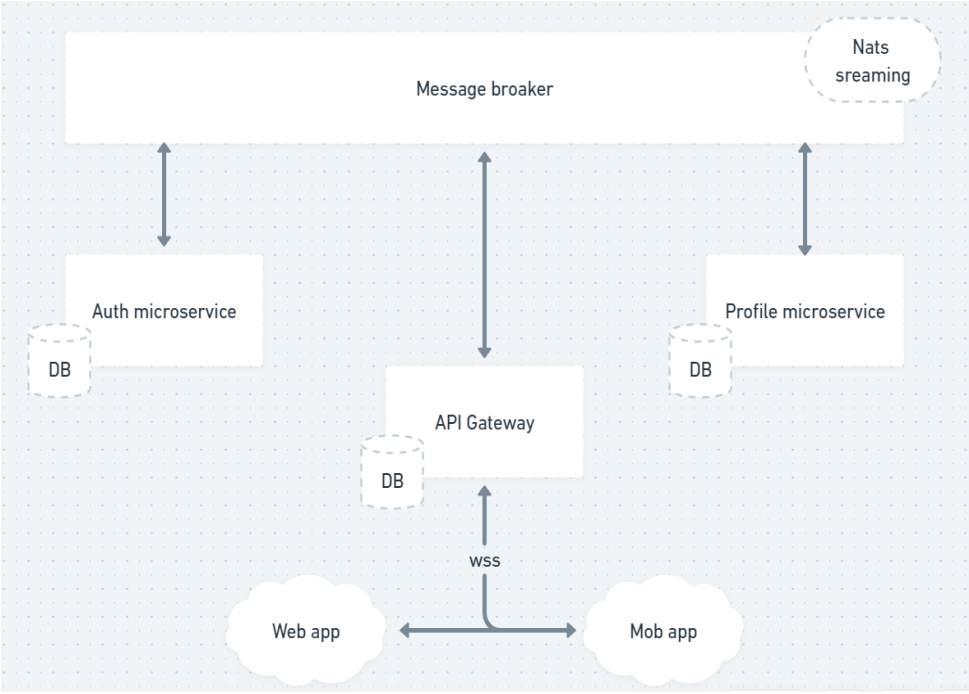


Рисунок 2.2 – Архітектура майбутньої мікросервісної системи.

Як видно з рисунку 2.2. спілкування з зовнішнім світом буде відбуватися за допомогою протоколу WSS. WebSocket — це протокол, що призначений для обміну інформацією між браузером та вебсервером в режимі реального часу. Він забезпечує двонаправлений повнодуплексний канал зв'язку через один TCP-сокет. WebSocket спроєктовано для втілення у веббраузерах та вебсерверах, але може також використовуватись будь-яким клієнт-серверним застосунком [9]. Прикладний програмний інтерфейс WebSocket був стандартизований W3C, крім того протокол WebSocket стандартизований IETF як RFC 6455.

У вебзастосунках доцільно використовувати протокол за необхідності відображення інформації в real-time. Альтернативна технологія — Server-sent events.

2.1.1 Розробка архітектури мікросервісів

Для загальної системи розпланована структуру, тепер потрібно визначитись зі структурою самих мікросервісів. Так як API Gateway є точкою входу о нашою системи, то і валідація даних буде відбуватися саме там. Також в цьому мікросервісі буде відбуватися ініціалізація з’єднання з WebSocket та https. Сервіс буде мітисти дві стратегії (Strategy pattern) одна для веб застосунку, друга – мобільного застосунку.

У розробці програмного забезпечення принципом програмування є інверсія керування (IoC). IoC інвертує потік контролю в порівнянні з традиційним потоком керування. Архітектура програмного забезпечення з таким дизайном інвертує управління в порівнянні з традиційним процедурним програмуванням. Саме таким чином потрібно намагатися побудувати мікросервіси.

В основному, ін’єкція залежностей (DI) є одним із способів досягнення IoC. Або можна сказати, що це реальна реалізація IoC. Замість того, щоб залежати від конкретної реалізації (класу, структури), ми залежимо від

інтерфейсів (протоколу), і всі залежності повинні надаватися ззовні. Це досягається шляхом відокремлення використання об’єкта від його створення. DI також допомагає дотримуватися принципу єдиної відповідальності [3].

Під час написання мікросервісів буде дотримуватися наступна структура (рис.2.3):

а) entity структура в якій описується майбутня схема для бази даних;

б) repository міститиме в собі запити для роботи з базою даних;

в) service міститиме в собі бізнес логіку, яка належить сервісу.

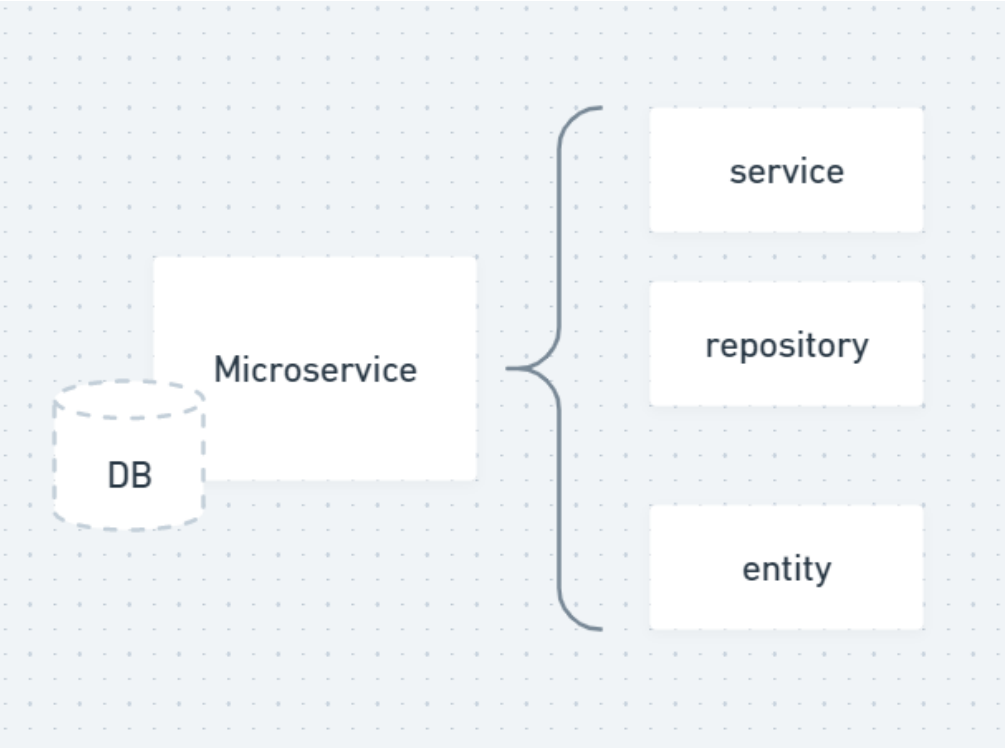


Рисунок 2.3 – Основна структура мікросервісу

Мікросервіс авторизації (auth microservice) буде складатися з наступних основних сервісів:

а) логіну (LoginService);

б) реєстрації (RegestrationService);

в) поштового сервісу (EmailService);

г) адміністратора (AdminService).

В свою чергу мікросервіс профілю (profile microservice) матиме 2 сервіса: ProfileService, BanService.

В залежності від кількості сервісів такою ж буде і кількість репозиторіїв та таблиць баз даних. Якщо потрібно буде приводити до третьої нормальної форми, то кількість таблиць може розширюватися.

2.2 Проєктування структури баз даних

Як зазначалось в першому розділи кожен мікросервіс має власну базу даних. Існує безліч різних типів баз даних, і вони написані багатьма мовами. Реляційні бази даних вміють зберігати дані в таблицях із жорсткими схемами, які пов’язані одна з одною через загальний стовпець, наприклад ідентифікатор, який стосується інформації лише для одного запису. Жорстка схема означає, що кожен запис повинен містити однакову інформацію, а кожен стовпець може містити лише один і той самий вказаний тип даних. Нереляційні бази даних, такі як MongoDB, ідеально підходять для зберігання даних, які не мають жорсткої схеми або визначеної форми, як-от нескінченний журнал чату.

В якості основи для мікросервісів я перейшов до вибору системи керування базами даних PostgreSQL. PostgreSQL — це система, яка використовує мову структурованих запитів (SQL). Вона популярна, оскільки є відкритою з вихідним кодом і може бути адаптована для багатьох цілей [18].

Для багатьох програмістів написання мовами баз даних може бути важким досвідом, особливо якщо вони не мали такої практики з цими мовами. З обмеженими знаннями запити, які пишуться, можуть бути досить простими. Проте переважна більшість програмістів мають міцну основу в об’єктно-орієнтованій мові програмування. Тому для облегшення написання запитів до бази даних буде використовуватися ORM, а саме TypeORM.

Об’єктно-реляційне відображення (ORM) — це техніка для перетворення даних між двома системами несумісних типів, у даному випадку, SQL та TypeScript. Це дозволить робити запити та маніпулювати даними з бази даних за допомогою об’єктно-орієнтованої мови програмування.

TypeORM — це ORM, який може працювати на платформах NodeJS, Browser, Cordova, PhoneGap, Ionic, React Native, NativeScript, Expo та Electron і може використовуватися з TypeScript та JavaScript (ES5, ES6, ES7, ES8). Його мета полягає в тому, щоб завжди підтримувати найновіші функції JavaScript і надавати додаткові функції, які допоможуть вам розробляти будь-які програми, які використовують бази даних – від невеликих програм з кількома таблицями до великомасштабних корпоративних програм з кількома базами даних [21].

TypeORM підтримує шаблони Active Record і Data Mapper, на відміну від усіх інших JavaScript ORM, які зараз існують, що означає, що ви можете писати високоякісні, слабо пов’язані, масштабовані та підтримувані програми найпродуктивнішим способом.

Для сервісу авторизації (Auth service) будемо мати базу клієнтів, яка в свою чергу матиме наступну структуру, зображену на рисунку 2.4.

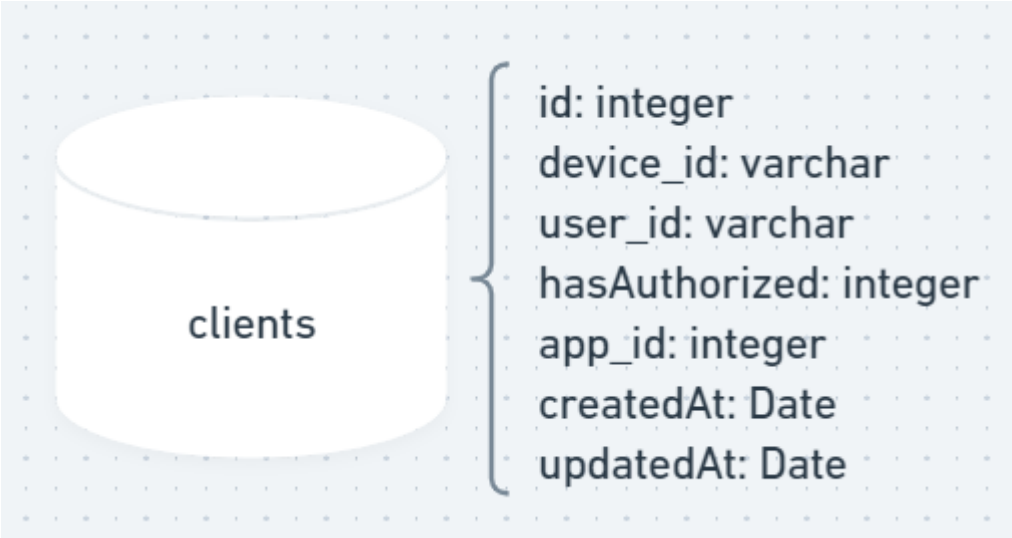


Рисунок 2.4 – Структура бази clients

Статус hasAuthorized буде містити інформацію про те чи є користувач авторизованим, чи не авторизованим. Таблиця clients є основною для користувачів, які зайшли через мобільний пристрій.

В таблицях admins, roles, emails, які зазначались в попередньому пункті будуть міститися дані, пов’язані з адміністраторами, їх ролями та шаблонами листів відповідно.

В свою чергу мікросервіс профілю (profile/user microservice) матиме 2 таблиці profiles (рис. 2.5), bans (рис. 2.6) з відносинами one-to-one.

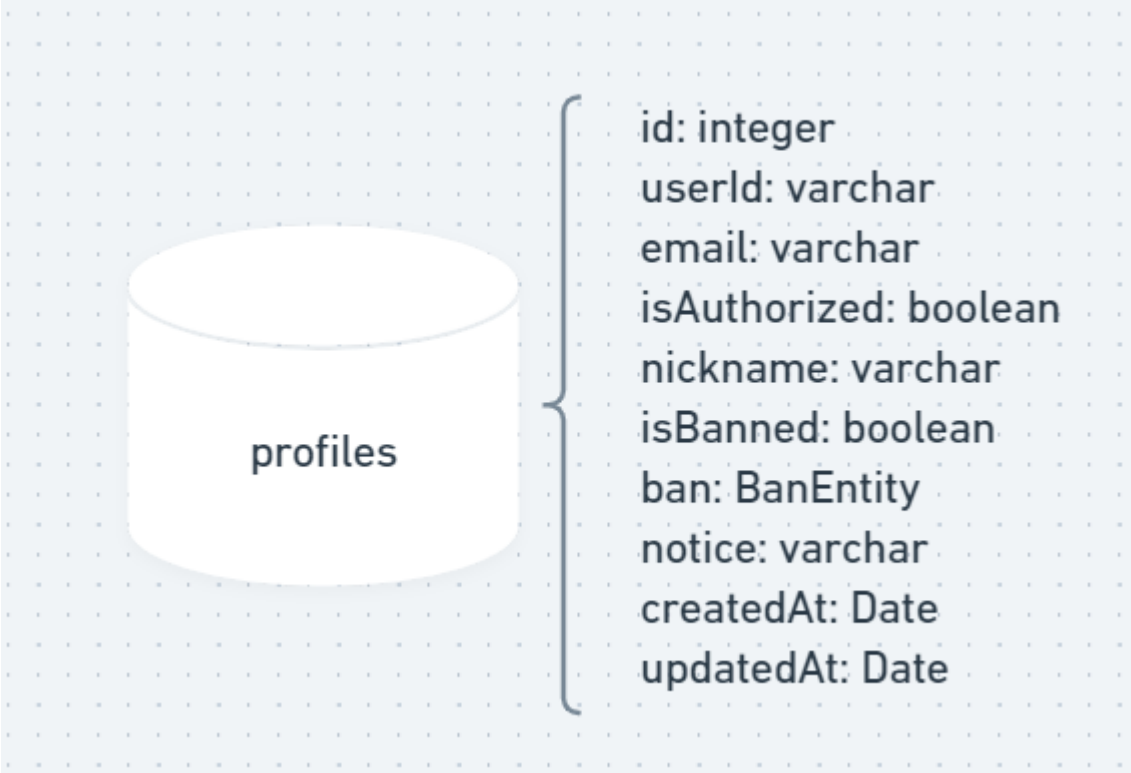


Рисунок 2.5 – Структура бази profiles

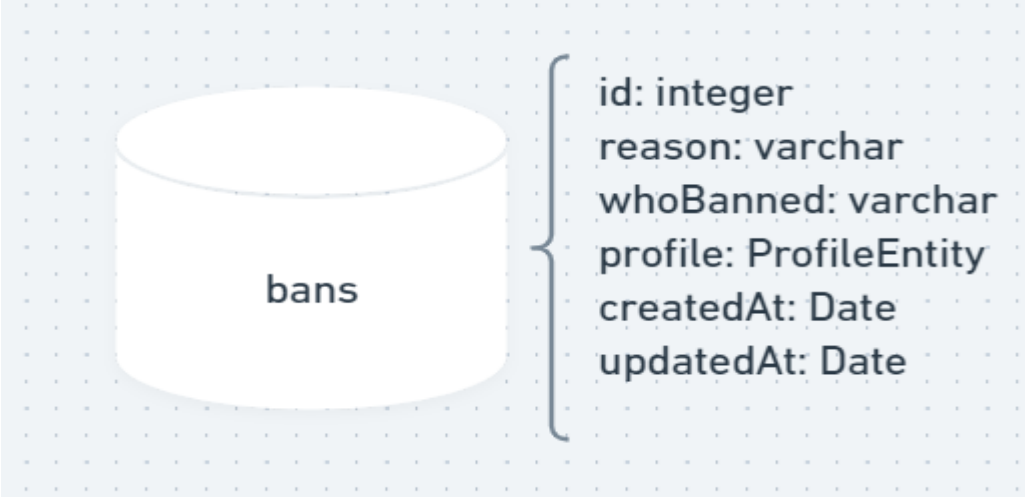


Рисунок 2.6 – Структура бази bans

Для зберігання токенів та інших даних які потрібно буде кешувати при роботі з сервісами, буде використовуватися Redis.

Redis — це сховище з відкритим вихідним кодом (ліцензія BSD), сховище структури даних у пам’яті, яке використовується як база даних, кеш-пам’ять, посередник повідомлень і механізм потокової передачі. Redis надає такі структури даних, як рядки, хеші, списки, набори, відсортовані набори із запитами діапазону, растрові зображення, гіперлогові журнали, геопросторові індекси та потоки. Redis має вбудовану реплікацію, сценарії Lua, вилучення LRU, транзакції та різні рівні збереження на диску, а також забезпечує високу доступність через Redis Sentinel та автоматичне розбиття на розділи за допомогою Redis Cluster [7].

2.3 Висновки за розділом

Виходячи з результатів проєктування, були вибрані інструментальні засоби для розробки програми та сформульовані вимоги до програмного забезпечення, необхідних для ефективного функціонування програмного засобу. Спроєктовано мікросервіси для контролю персональних даних користувачів. Розроблена архітектура програмної системи. Спроєктовані бази даних для збереження даних користувачів під час заходу до системи.

3 РЕАЛІЗАЦІЇ МІКРОСЕРВІСІВ ДЛЯ КОНТРОЛЮ ПЕРСОНАЛЬНИХ ДАНИХ КОРИСТУВАЧІВ

Добре написані проєкти мають чіткі послідовні конвенції щодо кодування з автоматизованим виконанням. Коли я розглядаю проєкт, і його код виглядає як будинок, побудований дитиною, використовуючи лише молоток і зображення будинку, це не вселяє впевненості, що код функціональний.

Чим більше зусиль вкладається в написання якісного коду, тим менше часу витрачається на налагодження та тестування. Можна підвищити якість коду та скоротити час, витрачений на налагодження, за допомогою послідовного робочого процесу розробки. Тому під час написання проєкту використовувався ESLint.

Окрім перевірки стилю, ESLint також є чудовими інструментом для пошуку певних класів помилок, наприклад, пов’язаних із змінною областю дії. Призначення неоголошених змінних і використання невизначених змінних є прикладами помилок, які можна виявити використовуючи ESLint [4].

Запуск всіх необхідних ключових додатків, взаємопов’язаних з мікросервісами буде відбуватися в Docker контейнері.

Docker — це проєкт з відкритим кодом для створення, доставки та запуску програм. Це програма командного рядка, фоновий процес і набір віддалених служб, які використовують матеріально-технічний підхід до вирішення поширених проблем програмного забезпечення та спрощують ваш досвід встановлення, запуску, публікації та видалення програмного забезпечення. Він досягає цього за допомогою технології операційної системи, яка називається контейнерами [16].

3.1 Розробка мікросервісу профілю

Спершу буде розроблений мікросервіс профілю (profile microservice). Як зазначалось в 2-му розділі він матиме 2 сервіси ProfileService та BanService. Під кожен сервіс потрібно написати свій репозиторій та створити 2 таблиці (bans, profiles). Однак, так як таблиці пов’язані та логіки зі взаємодією з таблицею bans не буде багато, то можна обійтися лише одним репозиторієм (ProfileRepository).

Буде розглянуто спочатку репозиторій ProfileRepository. Цей репозиторій міститиме наступні основні методи взаємодії:

а) create – для створення запису в таблиці;

б) getBanInfoStrict – для отримання інформації щодо бану користувача;

в) getByFilterStrict – універсальний метод, для отримання запису в базі за різними критеріями;

г) getAll – для отримання всіх записів в базі з пагінацією;

д) banStrict – для бану користувача;

е) unbanStrict – для роз бану користувача;

ж) isEmailExist – метод для перевірки наявності користувача в базі по email (необхідний для авторизації).

Написані в репозиторії методи будуть використовуватися для бізнес логіки, яка знаходиться в сервісах.

Таким чином, в сервісі профілю (ProfileService) є наступні методи:

а) changeNickname – для зміни ніку,

б) checkNickname – для валідації ніку,

в) getByUserId – отримання профілю для адміністратора,

г) getAll – отримання з пагінацією усіх записів,

д) getMy – отримання профілю для користувача,

е) banProfile – бан профілю для адміністратора,

ж) getBanInfo – отримання інформацію щодо бану для адміністратора,

з) getMyBanInfo – отримання інформацію щодо бану для користувача,
и) getByParams – отримання інформацію щодо профілю за різними критеріями для адміністратора,
к) getMyNickname – отримання ніку для користувача,
л) getNicknamesByUserIds – отримання списку ніків для адміністратора,
м) deleteUser – видалення користувача (насправді буде проводитися бан користувача на безкінечний строк),
н) isEmailExists – перевірка на існування пошти,
о) unban – розбан користувача.

В залежності від майбутніх потреб можна буде спокійно розширювати цей сервіс додатковими методами, або створити ще один сервіс.

Всі необхідні дані для підключення PostreSQL, NATS, Redis тощо, зберігаються у змінних середовища (Environment variables). Для Node.js такі змінні знаходяться в файлі .env.

Під час розгортання сервісів у кластерах та локальному запуску без необхідних підключень та змінних в .env сервіс закінчить свою роботу з помилкою. Але може бути таке, що в .env знаходяться допоміжні змінні, які не впливають на запуск сервісу, але використовуються для деяких функцій. Прикладом таких змінних може бути: ціна за зміну нікнейму, нагорода за запрошеного друга, нік за замовчуванням тощо. Тому варто також попіклуватися і про перевірку на наявність таких змінних. Для цього можна реалізувати невеличку функцію (рис. 3.1), яку можна викликати в конструкторі класу, де використовуються необхідні змінні.

```
export const checkConfigVariableList = (props: Record<string, unknown>): void => {
  const excludedValues = Object.keys(props).filter((key) => (!props[key] ? key : null));

  if (excludedValues.length) {
    throw new Error(`For '${excludedValues.join(', ')}' must be proper config variables`);
  }
};
```

Рисунок 3.1 – Функція для перевірки даних

Використовувати функцію можна таким чином, як на рис 3.2.

```
checkConfigVariableList({
  permanentBanPeriod: this.permanentBanPeriod,
  defaultCurrency: this.defaultCurrency,
  changeNicknamePrice: this.changeNicknamePrice,
});
```

Рисунок 3.2 – Приклад використання функції checkConfigVariableList

Після написання необхідних класів та функцій, створення файлів, внесення змінних структура майже кожного мікросервісу матиме наступний вигляд, як на рисунку 3.3.

В головній папці знаходяться необхідні файли для різних конфігурацій yarn, typescript, esLint тощо.

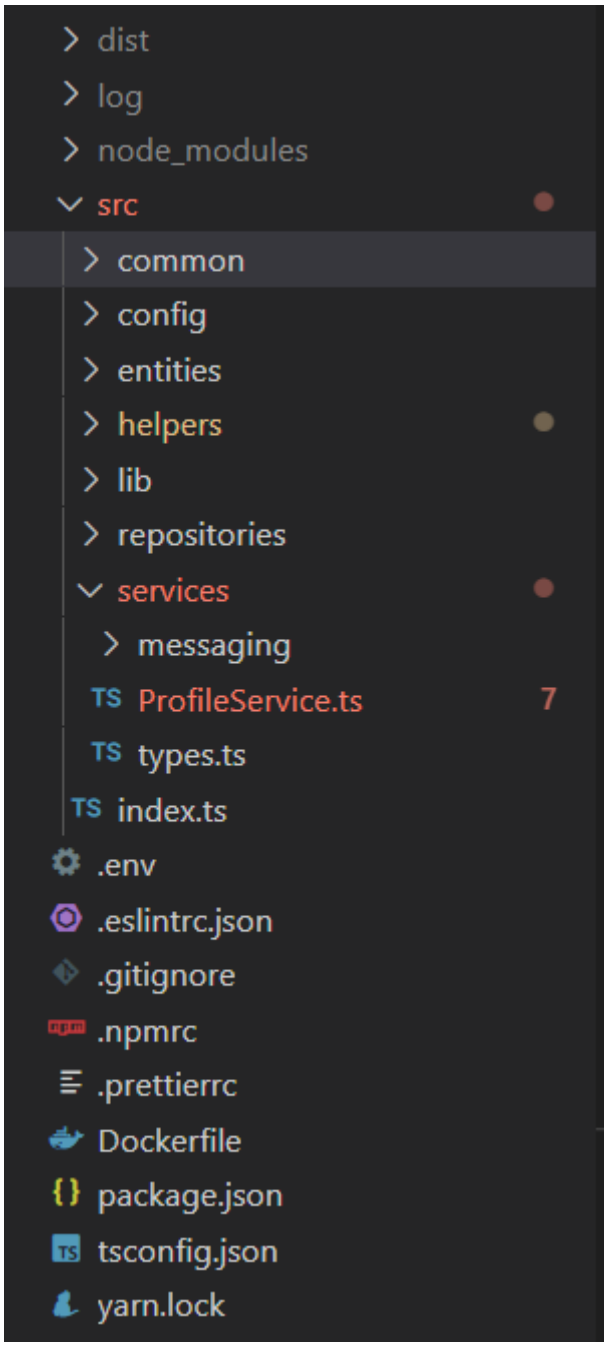


Рисунок 3.3 – Структура мікросервісу

3.2 Розробка мікросервісу авторизації

Аналогічно мікросервісу профілю, мікросервіс авторизації матиме таку ж саму структуру, як на рисунку 3.3. В 2-му розділі була продемонстрована для таблиці клієнтів. Але сервіс авторизації дещо складніший, ніж просто одна таблиця.

Для цього мікросервісу було створено, ще декілька сервісів, репозиторіїв та таблиць. Розглянемо основні:

а) confirm_emails таблиця (рис. 3.4) для збереження даних під час одного з реєстраційних процесів.

б) web_sessions зберігає дані користувача, який реєструється або авторизується через веб додаток (рис.3.5).

в) users таблиця зберігає основну інформацію про користувача (рис.3.6).

```
id              integer default nextval('confirm_emails_id_seq'::regclass)
email           varchar
device_id       varchar
hash            varchar
code            varchar
email_hash      varchar
pass_hash       varchar
referral_code   varchar
attempts        integer default 0
target          varchar
created_at      timestamp default now()
is_code_entered boolean default false
```

Рисунок 3.4 – Структура таблиці web_sessions

```
id           integer default nextval('web_sessions_id_seq'::regclass) -- pa
user_id      varchar
access_token varchar
exp_time     integer
created_at   timestamp default now()
updated_at   timestamp default now()
```

Рисунок 3.5 – Структура таблиці confirm_emails

```
id            integer default nextval('users_id_seq'::regclass)
email         varchar
user_id       varchar
password_hash varchar
roles         json
is_banned     boolean default false
enabled       boolean default true
google_id     varchar
facebook_id   varchar
created_at    timestamp default now()
updated_at    timestamp default now()
```

Рисунок 3.6 – Структура таблиці users

Процес реєстрації складається з декількох запитів, але з однаковою логікою, як для веб додатку, так і для мобільного додатку. Так само і процес логіну.

3.3 Розробка мікросервісу API gateway

API gateway це сервіс, який є точкою входу в програму із зовнішнього світу і діє як єдина точка входу для певної групи мікросервісів. На високому рівні він відповідає за маршрутизацію запитів, композицію API та інші функції, такі як аутентифікація.

Деякі служби можуть використовувати gRPC або, можливо, протокол обміну повідомленнями AMQP. Такі протоколи добре працюють всередині, але можуть бути важко використовуваними мобільним або веб клієнтом. Або, можливо, механізм якогось протоколу може бути важко адаптувати на якійсь клієнтській платформі.

API gateway відповідає за маршрутизацію запитів, композицію API та трансляцію протоколу. Усі запити API від зовнішніх клієнтів спочатку надходять до шлюзу API. Шлюз API направляє деякі запити до відповідної служби. Шлюз обробляє інші запити за допомогою шаблону API Composition

і викликає кілька служб і зводить результати. Він також може здійснювати переклад між зручними для клієнта протоколами, такими як HTTP і WebSockets, і недружніми для клієнта протоколами, які використовуються службами.

Так як під час роботи розроблялися два мікросервіси auth, profile, то API Gateway структура взаємозв’язку матиме вигляд на рисунку 3.7.

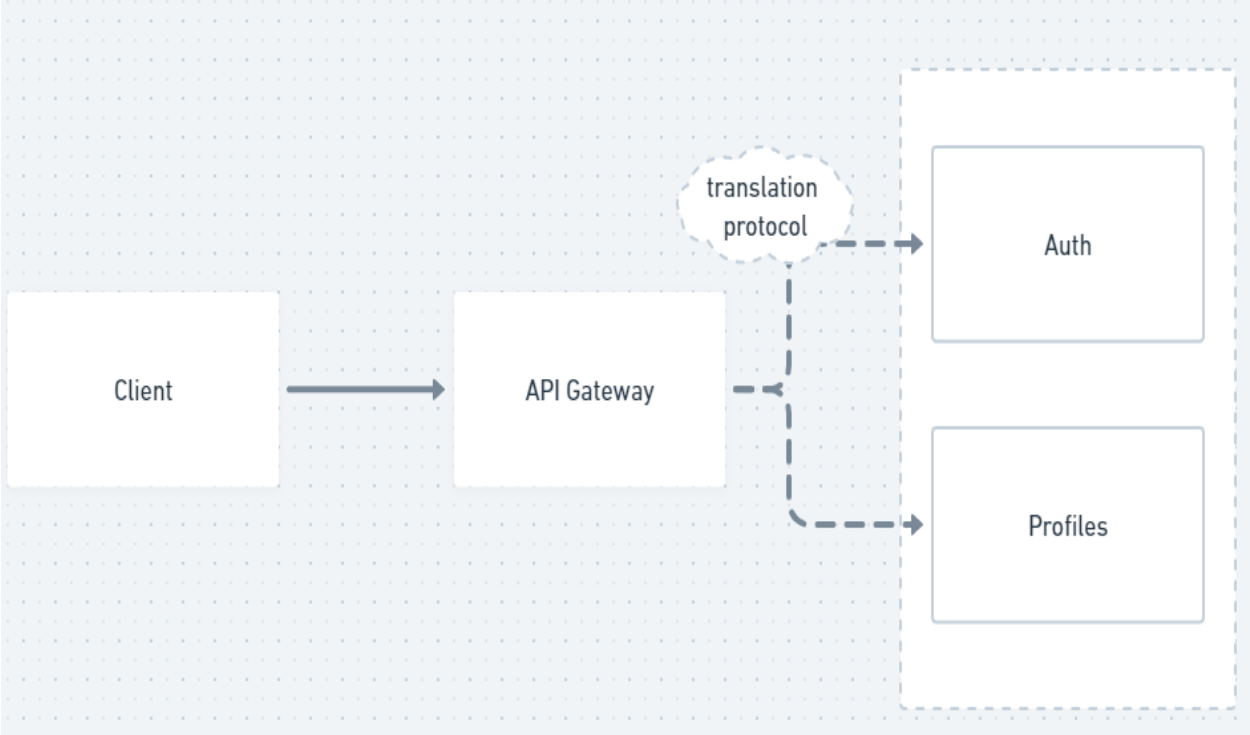


Рисунок 3.7 – Взаємозв’язок API Gateway з іншими мікросервісами та користувачем

В мікросервісі API Gateway будуть знаходитись наступні структури:

а) auth допоміжний клас для визначення по якій стратегії рухатися, будь то мобільна версія додатку, чи веб;
б) router транспортний прошарок, який буде відповідати за спілкування між мікросервісами;
в) sender допоміжна структура для router, яка буде надсилати дані на відповідні топіки в NATS;
г) listener клас, для підписування на топіки NATS;

д) state клас, який відповідає за маніпулювання, хешування та збереження даних користувача, для подальшого використання;

е) strategy клас, який реалізовує ідею однойменного шаблону стратегія (Strategy pattern).

Шаблон стратегії – це поведінковий шаблон проєктування, який дозволяє вибрати алгоритм під час виконання. Замість безпосередньої реалізації окремого алгоритму код отримує під час виконання інструкції щодо того, який із сімейства алгоритмів використовувати [5].

Також в API Gateway знаходяться обгортки для https, wss підключення клієнта. В цілому в будь якому проєкті всі бібліотеки, допоміжні класи слід обгортати. Існує загальна концепція «обгортання» об’єкта через проміжний клас, який може адаптуватися один до одного.

Також до кожного мікросервісу, у тому числі і до API Gateway добавлено допоміжний клас Semaphore.

Семафор обмежує доступ до ресурсу певній кількості споживачів. Будь-які нові запити до ресурсу будуть зупинені на потім, і врешті-решт будуть оброблені. Другими словами семафор є цілочисельною змінною, спільною між кількома процесами. Основною метою використання семафора є синхронізація процесів і контроль доступу до спільного ресурсу в одночасному середовищі [19].

Структура мікросервісу API Gateway відрізняється від інших мікросерісів і має наступний вид, як на рисунку 3.8.

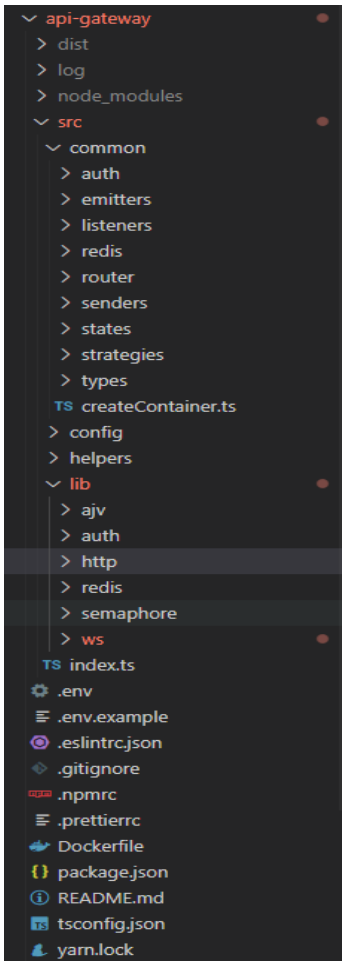


Рисунок 3.8 – Структура API Gateway

Так як API Gateway є точкою входу до розроблених мікросервісів, то він буде одним з найнавантаженіших сервісів. Але завдяки кластеризації Node.js та можливістю NATS, можна «плодити» будь-які сервіси. Таким чином запити будуть переходити між однаковими мікросервісами за алгоритмом Round-robin.

3.4 Висновки за розділом

Реалізовано серверну частину на базі мікросервісної архітектури для контролю персональних даних користувачів. Розроблено мікросервіс авторизації та профілю для реєстрації та авторизації користувачів у системі. Розроблено мікросервіс API Gateway для контролю доступу до всіх сервісів, транспортування необхідних повідомлень.

4 ПРИНЦИПИ РОБОТИ РОЗРОБЛЕНОЇ МІКРОСЕРВІСНОЇ СИСТЕМИ ТА БЕЗПЕКА ДАНИХ

Жоден код не є ідеальним, таким чином, жодне програмне забезпечення не є ідеальним. Деякі помилки – це проста неефективність коду, тоді як інші можуть мати катастрофічний вплив на безпеку, стабільність або якість – але позбутися шкідливих звичок програмування нелегко.

З проєктуванням за контрактом (Design by contract, DBC), терміном, який ввів консультант з програмування Бертран Мейєр, організації повинні визначити угоди про взаємодію програмних модулів: так само, як люди домовляються про контракти, так само повинна бути домовленість і про системи, які розроблюються [8].

Техніка проєктування стверджує, що програма повинна робити ні більше, ні менше, ніж вона заявляє, як це визначено передумовами, постумовами та інваріантами класу, які деталі організації в мові програмування. При такому підході кожен раз, коли програма не дотримується контракту, це призводить до помилки.

4.1 Перевірка та безпека даних

Під час написання коду використовувалась мова програмування TypeScript, яка дозволила контролювати вхідні та вихідні дані. Саме завдяки типізації можна було уникнути багатьох фантомним помилок, які потім важно вислідити.

Під час написання функцій, класів, сервісів в цілому тощо TypeScript допомагає програмісту, але також можуть виникати помилки в даних, які надсилає користувач. Незважаючи на те, що на стороні клієнтської частини є перевірки на вхідні дані, серверна частина повинна теж забезпечити валідність всіх даних.

Як зазначалось раніше, розроблена мікросервісна архітектура взаємодіє між собою за допомогою брокеру повідомлень NATS. Так як мікросервісна архітектура може містити величезну кількість сервісів, то писати перевірку на вхідні дані в кожній функції (яка видна ззовні) – затратно по часу та ресурсам.

Для вирішення такої проблеми вся основна перевірка буде знаходитись в API Gateway. У разі якщо дані не валідні, то попросту метод який повинен виконуватися – не виконається.

Постає питання, як API Gateway знати, які саме змінні та їх типи приймає та чи інша функція. Як відомо мікросервіси взаємодіють між собою за допомогою брокеру повідомлень, а саме через топіки (topics). Топіки можна згенерувати для кожного метода автоматично, отже можна також і згенерувати для кожної функції свій контракт, в якому будуть описуватися дані, їх типи та обмеження. Більш того також можна згенерувати і контракт для вихідних даних.

Для генерування типів даних та їх обмежень використовувалась бібліотека TypeBox — це бібліотека, яка створює в пам’яті об’єкти JSON Schema, які статично можна вивести як типи TypeScript. Схеми, створені цією бібліотекою, розроблені для відповідності правилам статичної перевірки типів компілятора TypeScript. TypeBox дозволяє створити уніфікований тип, який може бути як статично підтверджений компілятором TypeScript, так і під час виконання за допомогою стандартної перевірки схеми JSON.

TypeBox можна використовувати як простий інструмент для створення складних схем або інтегрувати в служби RPC або REST, щоб допомогти перевірити дані JSON.

Приклад розробленої схеми (Schema) для методу login з сервісу авторизації зображено на рисунку 4.1.

```
login: {
  accessLevel: 'public',
  schema: Type.Object({
    email: Type.String(),
    password: Type.String({ minLength: 6 }),
    deviceId: Type.Optional(Type.String()),
    appId: Type.Optional(IntType()),
  }),
  returnSchema: Type.Object({
    access_token: Type.String(),
    expires_in: IntType(),
    refresh_expires_in: IntType(),
    refresh_token: Type.String(),
    userId: Type.String(),
  }),
},
```

Рисунок 4.1 – Приклад схеми для методу login

Як видно зі схеми функція login приймає в якості змінної об’єкт з наступними ключами:

а) email – пошта користувача типу string;

б) password – пароль користувача типу string з обмеженнями на мінімальну довжину в 6 символів;

в) deviceId – не обов’язкове поле, яке являється унікальним ідентифікатором пристрою користувача;

г) appId – не обов’язкове поле-ідентифікатор додатку з якого надходить запит.

Також після виконання функція її результат видно зі схеми, яка повертається (returnSchema).

Кожна схема після компіляції генерую JSON файл з об’єктом наступного виду (рис. 4.2.).

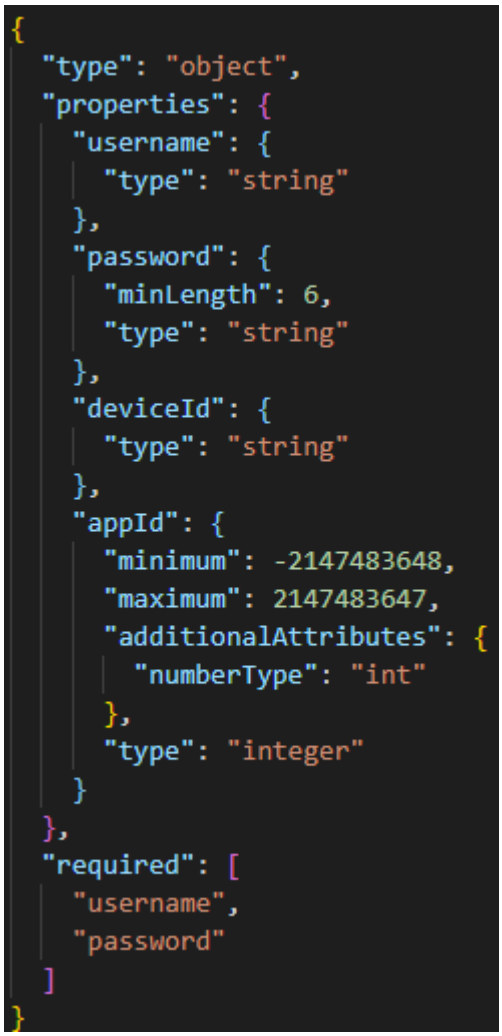


Рисунок 4.2 – Згенерований JSON об’єкт для схеми методу login

Як вказують автори бібліотеки TypeBox, вона не надає жодної перевірки схеми JSON. Тому варто використовувати такі бібліотеки для перевірки схем як AJV.

Саме AJV бібліотека і буде перевіряти в API Gateway всі схеми, які написані для методів. Щоб придержуватися принципу інверсії залежностей, для AJV також як і для багатьох бібліотек була написана обгортка (wrapper). Також ця бібліотека дозволяє контролювати і текст помилок, якщо дані уведенні неправильно можна про це повідомити та вказати, які саме допущенні помилки (рис. 4.3, рис. 4.4).

```
"response": {
    "type": "Left",
    "value": {
        "type": "validation.invalid",
        "msg": "Validation error",
        "data": ["must have required property 'email'", "must have required property 'password'"],
        "status": 1423409
    }
}
```


Рисунок 4.3 – Повідомлення про відсутність полів під час тригеру методу login


```
"response": {
    "type": "Left",
    "value": {
        "type": "validation.invalid",
        "msg": "Validation error",
        "data": ["Invalid type of password - must be string"],
        "status": 1423409
    }
}
```


Рисунок 4.4 – Повідомлення про некоректний тип для поля паролю

Також під час роботи з базами даних, підрахунку тих чи інших результатів, діставання даних зі структур тощо, можуть виникати помилки. Для їх обробки в багатьох мовах програмування, TypeScript не виняток, використовують блоки try..catch.

Конструкція try...catch намагається виконати інструкції в блоку try, і, у випадку помилок, виконує блок catch. Для того, щоб можна було перевіряти на наявність помилок на вищому рівні, під час розробки використовувалась бібліотека sweet-monads. Це проста у використанні реалізація монад зі статичним визначенням типів і відокремленими пакетами.

У функціональному програмуванні монада — це шаблон проєктування програмного забезпечення зі структурою, яка об’єднує фрагменти програми (функції) та обертає їх значення, що повертаються, у тип з додатковим обчисленням. На додаток до визначення обгортаючого монадного типу, монади визначають два оператори: один для обгортання значення в тип монади, а інший для складання разом функцій, які виводять значення монадного типу (вони відомі як монадні функції) [13].

Приклад повертання помилки під час роботи з базою даних та використання right/left монади зображено на рисунку 4.5.

```
async add(params: DeepPartial<AuthDeviceEntity>): AsyncEitherWithErr<AuthDeviceEntity> {
  try {
    const client = this.create(params);
    await this.save(client);
    return right(client);
  } catch (e) {
    return left({
      type: errorsRegistry.db.insert,
      msg: (e as Error).message,
      trace: (e as Error).stack,
    });
  }
}
```

Рисунок 4.5 – Використання right/left монад під час запиту до бази даних

Після того як повертається результат виконання його можна перевірити на ту, чи іншу монаду. Якщо трапилась помилка, то її можна обробити, в іншому випадку просто розпаковуємо дані з монади (рис. 4.6).

```
const userInfoEither = await this.em.getCustomRepository(ConfirmEmailRepository)
  .getEntityByPasswordHash(emailHash);
if (userInfoEither.isLeft()) return userInfoEither as unknown as AsyncEitherWithErr<AuthAuthSetPasswordResult>;
const { value: { email, referralCode } } = userInfoEither;
```

Рисунок 4.6 – Приклад використання повернутої монади

4.2 Система ролі для користувачів

Майже кожен додаток в тому числі і мобільний мають свою систему ролі, а саме відрізняються звичайні користувачі, модератори, адміністратори тощо.

Методи в мікросервісах, які мають доступ ззовні теж повинні мати свої рівні доступності. Так як API Gateway є точкою входу до мікросервісів та їх методів, то очевидно, що він і повинен контролювати ці рівні.

Для кожної стратегії мобільної, чи веб версії контроль ролі для користувача виглядає однаково. Перевірка рівня доступу до методу

описується в схемах в полі accessLevel (рис. 4.1), які обговорювались в попередньому пункті.

Маємо наступні рівні доступу до методів:

а) public – доступ має будь який користувач додатку,

б) private – доступ має лише адміністратор,

в) protected – доступ тільки для авторизованого користуача,

г) internal – між сервісні методи, доступ не має ніхто, окрім самих мікросервісів.

Приклад базової перевірки рівня методу зображено на рис. 4.7. Дані про рівень надходить як і перевірка типів зі JSON об’єкту.

```
if (isInternal) return right({ info: {}, authorized: false, infoLoaded: false });

if (isPublic) return right({ info: {}, authorized: true, infoLoaded: false });

if (isProtected && !request.auth.token) {
  return left({
    type: errorsRegistry.auth.accessDenied,
    msg: 'Not enough permissions',
  });
}

if (isPrivate && !request.auth.token) {
  return left({
    type: errorsRegistry.auth.accessDenied,
    msg: 'Not enough permissions',
  });
}
```

Рисунок 4.7 – Базова перевірка рівня доступу методів

За необхідністю перевірку можна доповнювати своїми правилами та більш гнучко контролювати того чи іншого користувача. Наприклад на рис. 4.8. при перевірці userRoleEither можна також перевірити не тільки наявність ролі, а й яка саме роль у адміністратора: administrator, superAdministrator, moderator, helper тощо. І відповідно до ролі давати доступ то тих чи інших методів системи, чи навпаки забороняти доступ.

```
    if (isPrivate) {
      if (userRoleEither.value) {
        this.nats.publish<AdminActionsProps>(AdminActionsBroadcast, {
          id: request.id,
          method: request.method,
          params: request.params,
          agent: request.agent,
          userId: userInfoByToken.userId,
        });

        return right({
          info: userInfoByToken,
          authorized: true,
          infoLoaded: true,
        });
      }
      return left({
        type: errorsRegistry.auth.accessDenied,
        msg: 'You don\'t have permissions to use this',
      });
    }
  }
```

Рисунок 4.8 – Перевірка наявності ролі для приватного рівня доступу

Для методів private, protected використовується токен, який повертається користувачеві під час логіну, реєстрації. В токен вбудована вся необхідна інформація про користувача. Якщо токен не валідний, чи зовсім відсутній, то користувача не допускає до методу (рис. 4.9).

Окрім системи ролі можна обмежити кількість запитів, які надсилаються до системи від певного користувача, або навіть від всіх користувачів до конкретного методу.

```
{
    "result": {
        "request": {
            "id": "l3u6h5tt-dmkykdty70-23hk6mt6my7",
            "method": "profile_getMyProfile",
            "lang": "en",
            "params": {},
            "agent": "user",
            "auth": {
                "token": null,
                "deviceId": null,
                "type": null
            }
        },
        "response": {
            "type": "Left",
            "value": {
                "type": "auth.accessDenied",
                "msg": "Not enough permissions",
                "status": 15666
            }
        }
    }
}
```


Рисунок 4.9 – Запит та відповідь при відсутності токена користувача

Отже, використання схем для методів та єдиної точки входу до них надає можливість захистити систему від атак. А система ролі дозволяє розділити ці методи між користувачами різного рівня доступу, чи навпаки заборонити доступ всім.

4.3 Висновки за розділом

Розглянуто принципи роботи розробленої програмної системи. Детально описана перевірка та безпека даних. Продемонстровано розроблену систему ролі для користувачів. Визначена послідовність дій при виконанні основних операцій.

5 ТЕСТУВАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ДЛЯ КОНТРОЛЮ ПЕРСОНАЛЬНИХ ДАНИХ КОРИСТУВАЧІВ

Тестування програмного забезпечення — це процес дослідження програмного забезпечення з метою виявлення помилок та визначення відповідності між фактичною та очікуваною поведінкою програмного забезпечення, що здійснюється на основі набору тестів, обраних певним чином. У ширшому сенсі тестування програмного забезпечення — це методика контролю якості програмного забезпечення, яка включає розробку тесту, виконання тесту та аналіз отриманих результатів.

Основна мета – показати, що продукт готовий до випуску на ринок, що всі заявлені розробником функції працюють стабільно. Тестування необхідне і самому розробнику, щоб переконатися, що продукт готовий, а клієнтам побачити, за що вони заплатили [6].

Навряд чи знайдеться розробник, який ідеально пише код. Люди схильні до помилок, людська помилка може призвести до порушення нормальної роботи програмного забезпечення на всіх етапах розробки програмного забезпечення. Людина не може все зберегти в пам’яті, тому виправлення – нормальна справа. Більше того, навіть найкваліфікованіший розробник може не помітити помилок, оскільки він/вона розглядає свій код досить довго.

Важливі всі рівні тестування, незалежно від використовуваних моделей і методологій.

5.1 Модульне тестування

Модульне тестування (Unit testing). Цей рівень тестування дозволяє перевірити функціонування окремо взятого елемента системи. Що вважати елементом – модулем системи визначається контекстом.

Для перевірки правильності методів можна визивати кожен метод локально у конструкторі класу. Так як бізнес логіка в даному проєкті міститься

в сервісам, то і перевіряти варто методи саме там. Наприклад, був спрацьований метод в сервісі користувачів мікросервісу авторизації (рис. 5.1.) та перевірена коректність його роботи з логів (рис. 5.2.) та наявністю запису в базі даних (рис. 5.3).

```
constructor(
  private readonly em: EntityManager,
  loggerFactory: Logger,
) {
  this.logger = loggerFactory.getLogger(`(${this.constructor.name})`);

  void (async () => this.logger.debug(await this.add({
    email: 'test@mail.com',
    userId: 'testUserId',
    passwordHash: 'testPasswordHash',
    roles: [],
    isBanned: false,
    googleId: 'testGoogleId',
    facebookId: '',      You, 1 second ago • Uncommitted changes
  })))();
}
```

Рисунок 5.1 – Тригер методу add в конструкторі класу UserService


```
[2022-06-06T20:05:28.548] [DEBUG] [7112] (service=auth)(UserService) - {"type
":"Right","value":{"id":1,"email":"test@mail.com","userId":"testUserId","pass
wordHash":"testPasswordHash","roles":[],"isBanned":false,"googleId":"testGoog
leId","facebookId":"","appIds":[],"createdAt":"2022-06-06T14:05:28.451Z","upd
atedAt":"2022-06-06T14:05:28.451Z"}}
```


Рисунок 5.2 – Результат виконання методу у консолі

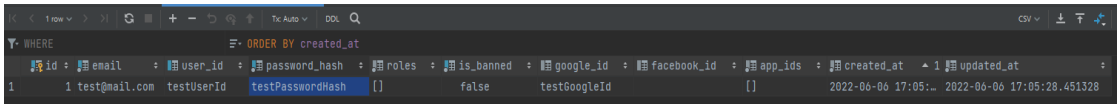
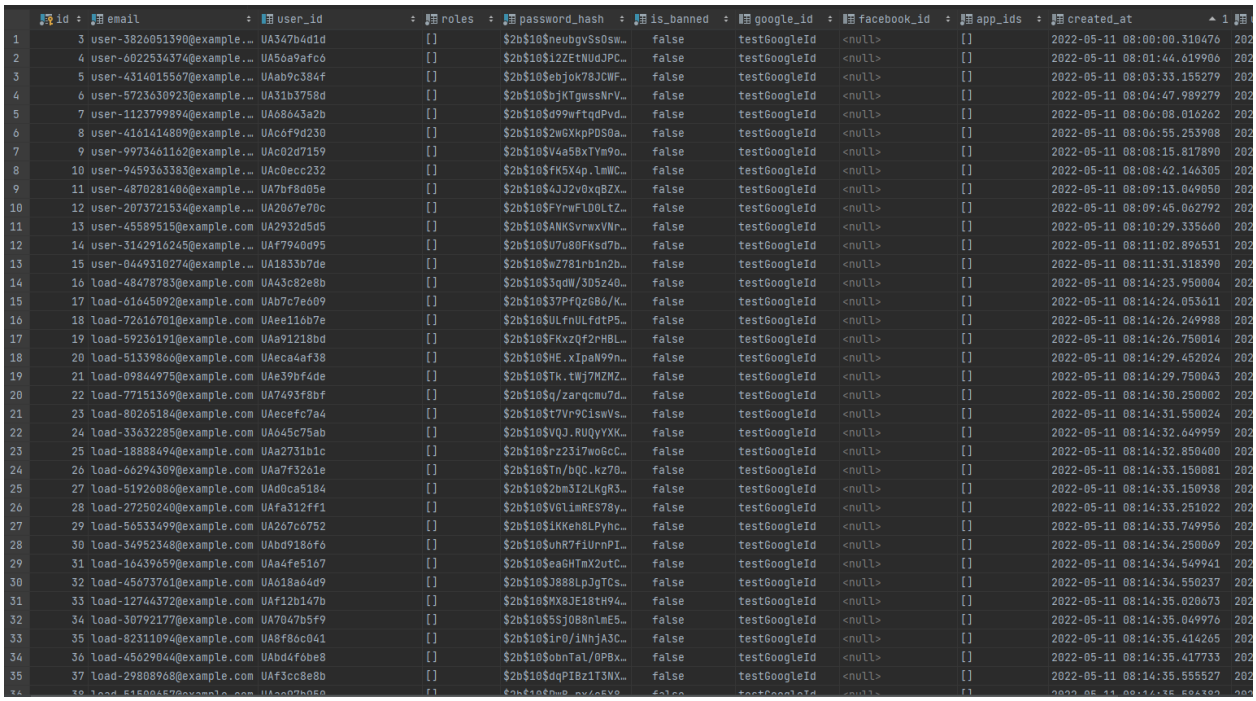


Рисунок 5.3 – Запис у базі даних

Таким чином можна перевірити усі методи, незважаючи на те, чи використовуються вони ззовні, чи є вони приватними.

Для роботи з базою даних використовувалась програма DataGrip, цей інструмент чудово справляється зі своєю роботою та добре інтегрований для PostgreSQL.

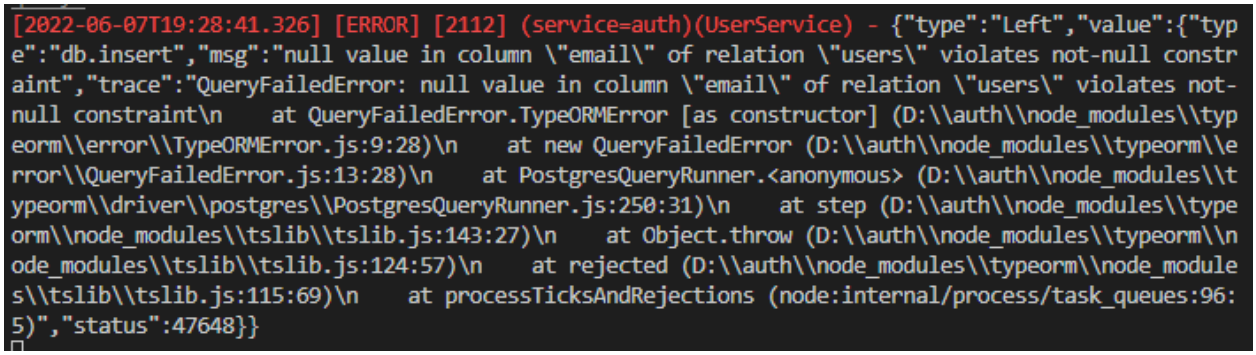
Можна написати додаткові функції, які будуть імітувати викликання методів і генерувати унікальні userId, email тощо (рис. 5.4).



	id	email	user_id	roles	password_hash	is_banned	google_id	facebook_id	app_ids	created_at
1	3	user-3826051390@example.…	UA347b4d1d	[]	$2b$10$neubgvSsOsw…	false	testGoogleId	<null>	[]	2022-05-11 08:00:00.310476
2	4	user-6022534374@example.…	UA56a9afc6	[]	$2b$10$i2ZEtNUdJPC…	false	testGoogleId	<null>	[]	2022-05-11 08:01:44.619906
3	5	user-4314015567@example.…	UAab9c384f	[]	$2b$10$ebjok78JCWF…	false	testGoogleId	<null>	[]	2022-05-11 08:03:33.155279
4	6	user-5723630923@example.…	UA31b3758d	[]	$2b$10$bjKTgwssNrV…	false	testGoogleId	<null>	[]	2022-05-11 08:04:47.989279
5	7	user-1123799894@example.…	UA68643a2b	[]	$2b$10$d99wftqdPvd…	false	testGoogleId	<null>	[]	2022-05-11 08:06:08.016262
6	8	user-4161414809@example.…	UAc6f9d230	[]	$2b$10$2wGXkpPDS0a…	false	testGoogleId	<null>	[]	2022-05-11 08:06:55.253908
7	9	user-9973461162@example.…	UAc02d7159	[]	$2b$10$V4a58xTYm9o…	false	testGoogleId	<null>	[]	2022-05-11 08:08:15.817890
8	10	user-9459363383@example.…	UAc0ecc232	[]	$2b$10$fK5X4p.lmWC…	false	testGoogleId	<null>	[]	2022-05-11 08:08:42.146305
9	11	user-4870281406@example.…	UA7bf8d05e	[]	$2b$10$4JJ2v0xqBZX…	false	testGoogleId	<null>	[]	2022-05-11 08:09:13.049050
10	12	user-2073721534@example.…	UA2067e70c	[]	$2b$10$FYrwFlDOLtZ…	false	testGoogleId	<null>	[]	2022-05-11 08:09:45.062792
11	13	user-45589515@example.com	UA2932d5d5	[]	$2b$10$ANKSvrwxVNr…	false	testGoogleId	<null>	[]	2022-05-11 08:10:29.335660
12	14	user-3142916245@example.…	UAf7940d95	[]	$2b$10$U7u80FKsd7b…	false	testGoogleId	<null>	[]	2022-05-11 08:11:02.896531
13	15	user-0449310274@example.…	UA1833b7de	[]	$2b$10$wZ781rb1n2b…	false	testGoogleId	<null>	[]	2022-05-11 08:11:31.318390
14	16	load-48478783@example.com	UA43c82e8b	[]	$2b$10$3qdW/3D5z40…	false	testGoogleId	<null>	[]	2022-05-11 08:14:23.950004
15	17	load-61645092@example.com	UAb7c7e609	[]	$2b$10$37PfQzGB6/K…	false	testGoogleId	<null>	[]	2022-05-11 08:14:24.053611
16	18	load-72616701@example.com	UAee116b7e	[]	$2b$10$ULfnULfdtP5…	false	testGoogleId	<null>	[]	2022-05-11 08:14:26.249988
17	19	load-59236191@example.com	UAa91218bd	[]	$2b$10$FKxzQf2rHBL…	false	testGoogleId	<null>	[]	2022-05-11 08:14:26.750014
18	20	load-51339866@example.com	UAeca4af38	[]	$2b$10$HE.xIpaN99n…	false	testGoogleId	<null>	[]	2022-05-11 08:14:29.452024
19	21	load-09844975@example.com	UAe39bf4de	[]	$2b$10$Tk.tWj7MZMZ…	false	testGoogleId	<null>	[]	2022-05-11 08:14:29.750043
20	22	load-77151369@example.com	UA7493f8bf	[]	$2b$10$q/zarqcmu7d…	false	testGoogleId	<null>	[]	2022-05-11 08:14:30.250002
21	23	load-80265184@example.com	UAecefc7a4	[]	$2b$10$t7Vr9CiswVs…	false	testGoogleId	<null>	[]	2022-05-11 08:14:31.550024
22	24	load-33632285@example.com	UA645c75ab	[]	$2b$10$VQJ.RUQyYXK…	false	testGoogleId	<null>	[]	2022-05-11 08:14:32.649959
23	25	load-18888494@example.com	UAa2731b1c	[]	$2b$10$rz23i7woGcC…	false	testGoogleId	<null>	[]	2022-05-11 08:14:32.850400
24	26	load-66294309@example.com	UAa7f3261e	[]	$2b$10$Tn/bQC.kz70…	false	testGoogleId	<null>	[]	2022-05-11 08:14:33.150081
25	27	load-51926086@example.com	UAd0ca5184	[]	$2b$10$2bm3I2LKgR3…	false	testGoogleId	<null>	[]	2022-05-11 08:14:33.150938
26	28	load-27250240@example.com	UAfa312ff1	[]	$2b$10$VGlimRES78y…	false	testGoogleId	<null>	[]	2022-05-11 08:14:33.251022
27	29	load-56533499@example.com	UA267c6752	[]	$2b$10$iKKeh8LPyhc…	false	testGoogleId	<null>	[]	2022-05-11 08:14:33.749956
28	30	load-34952348@example.com	UAbd9186f6	[]	$2b$10$uhR7fiUrnPI…	false	testGoogleId	<null>	[]	2022-05-11 08:14:34.250069
29	31	load-16439659@example.com	UAa4fe5167	[]	$2b$10$eaGHTmX2utC…	false	testGoogleId	<null>	[]	2022-05-11 08:14:34.549941
30	32	load-45673761@example.com	UA618a64d9	[]	$2b$10$J888LpJgTCs…	false	testGoogleId	<null>	[]	2022-05-11 08:14:34.550237
31	33	load-12744372@example.com	UAf12b147b	[]	$2b$10$MX8JE18tH94…	false	testGoogleId	<null>	[]	2022-05-11 08:14:35.020673
32	34	load-30792177@example.com	UA7047b5f9	[]	$2b$10$5SjOB8nlmE5…	false	testGoogleId	<null>	[]	2022-05-11 08:14:35.049976
33	35	load-82311094@example.com	UA8f86c041	[]	$2b$10$ir0/iNhjA3C…	false	testGoogleId	<null>	[]	2022-05-11 08:14:35.414265
34	36	load-45629044@example.com	UAbd4f6be8	[]	$2b$10$obnTal/OPBx…	false	testGoogleId	<null>	[]	2022-05-11 08:14:35.417733
35	37	load-29808968@example.com	UAf3cc8e8b	[]	$2b$10$dqPIBz1T3NX…	false	testGoogleId	<null>	[]	2022-05-11 08:14:35.555527

Рисунок 5.4 – Автоматично згенеровані дані, записані до таблиці users

Тестування методів таким чином допомагає виявити помилки раніше з мінімальними зусиллями. Якщо виникли якісь труднощі, то все що потрібно – це просто змінити код і запустити методи знову. Завдяки такому підходу та правильному запису результатів до консолі можна навіть виявити помилки від бази даних (рис. 5.5), винятки в функціях (рис. 5.6), проблемами з даними тощо.



```
[2022-06-07T19:28:41.326] [ERROR] [2112] (service=auth)(UserService) - {"type":"Left","value":{"type":"db.insert","msg":"null value in column \"email\" of relation \"users\" violates not-null constraint","trace":"QueryFailedError: null value in column \"email\" of relation \"users\" violates not-null constraint\n    at QueryFailedError.TypeORMError [as constructor] (D:\\auth\\node_modules\\typeorm\\error\\TypeORMError.js:9:28)\n    at new QueryFailedError (D:\\auth\\node_modules\\typeorm\\error\\QueryFailedError.js:13:28)\n    at PostgresQueryRunner.<anonymous> (D:\\auth\\node_modules\\typeorm\\driver\\postgres\\PostgresQueryRunner.js:250:31)\n    at step (D:\\auth\\node_modules\\typeorm\\node_modules\\tslib\\tslib.js:143:27)\n    at Object.throw (D:\\auth\\node_modules\\typeorm\\node_modules\\tslib\\tslib.js:124:57)\n    at rejected (D:\\auth\\node_modules\\typeorm\\node_modules\\tslib\\tslib.js:115:69)\n    at processTicksAndRejections (node:internal/process/task_queues:96:5)","status":47648}}
```

Рисунок 5.5 – Помилка від бази даних

```
[2022-06-07T19:27:22.950] [ERROR] [9596] (service=auth)(UserService) - TypeError: Cannot read properties of undefined (reading 'userInfo'
    at D:\auth\dist\services\UserService.js:27:33
    at new UserService (D:\auth\dist\services\UserService.js:32:11)
    at InternalDependencyContainer.construct (D:\auth\node_modules\tsyringe\dist\cjs\dependency-container.js:267:16)
    at InternalDependencyContainer.resolveRegistration (D:\auth\node_modules\tsyringe\dist\cjs\dependency-container.js:156:51)
    at InternalDependencyContainer.resolve (D:\auth\node_modules\tsyringe\dist\cjs\dependency-container.js:100:33)
    at D:\auth\node_modules\tsyringe\dist\cjs\dependency-container.js:287:29
    at Array.map (<anonymous>)
    at InternalDependencyContainer.construct (D:\auth\node_modules\tsyringe\dist\cjs\dependency-container.js:266:34)
    at InternalDependencyContainer.resolveRegistration (D:\auth\node_modules\tsyringe\dist\cjs\dependency-container.js:156:51)
    at InternalDependencyContainer.resolve (D:\auth\node_modules\tsyringe\dist\cjs\dependency-container.js:100:33)
```

Рисунок 5.6 – Помилка під час зчитування userInfo

5.2 Інтеграційне тестування

Інтеграційне тестування – це спосіб тестування програмного забезпечення шляхом групування програмних компонентів разом. У складній програмній системі існує безліч взаємозв'язків, застосовних до певної функції, що ускладнює написання інтеграційних тестів [6]. Це можна проілюструвати на рисунку 5.7. У цьому робочому процесі ми маємо тестові вхідні дані, які надаються системному компоненту A, який потім взаємодіє з компонентами B і C. Результат виводу потім перевіряється з очікуваним результатом.

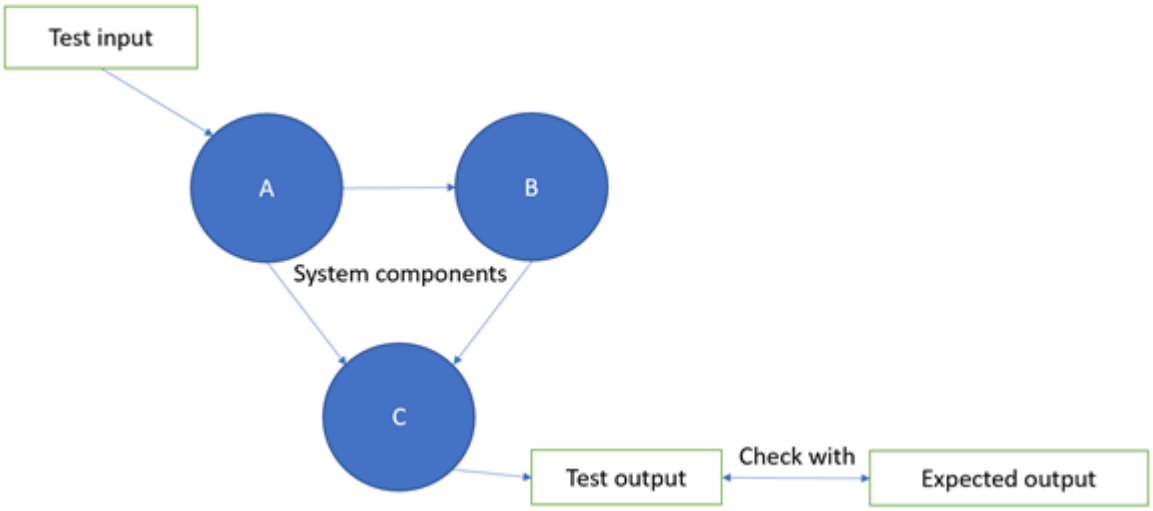


Рисунок 5.7 – Приклад робочого процесу інтеграційного тесту

Інтеграційне тестування так само, як і модульне потребую необхідних знань в написані автоматизованих тестів. Так як цією проблемою займаються спеціально навчені люди QA, то в роботі також використовувалось ручне інтеграційне тестування.

Для такого типу тестування потрібно вже повністю запустити не тільки якийсь мікросервіс, а й інші з ним пов’язані мікросервіси. Тож запустивши усе необхідно можна почати тестувати. Але постає питання, яким саме чином імітувати підключення до нашого API Gateway? Для цього можна використати розширення для Chrome браузеру Advanced WebSocket Client (рис. 5.8).

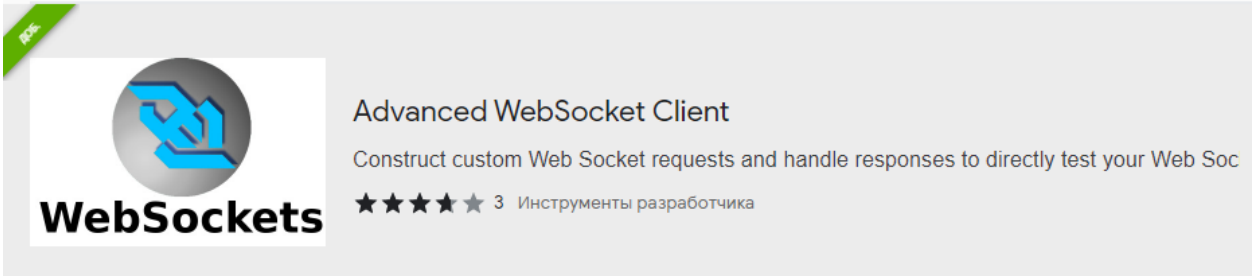


Рисунок 5.8 – Advanced WebSocket Client розширення для Chrome

Дане розширення допомагає зручно тестувати ws/wss з’єднання та має наступний інтерфейс (рис. 5.9). Для тестування потрібно вказати до якого саме сервера потрібно встановити підключення. Так як API Gateway працює локально, то вказано ws://localhost:4000. Коли зв’язок відкрився, то в консолі в мікросервісі API Gateway можна побачити повідомлення про успішне підключення та створення clientId (рис. 5.10). Після цього можна тестувати запити до методів.

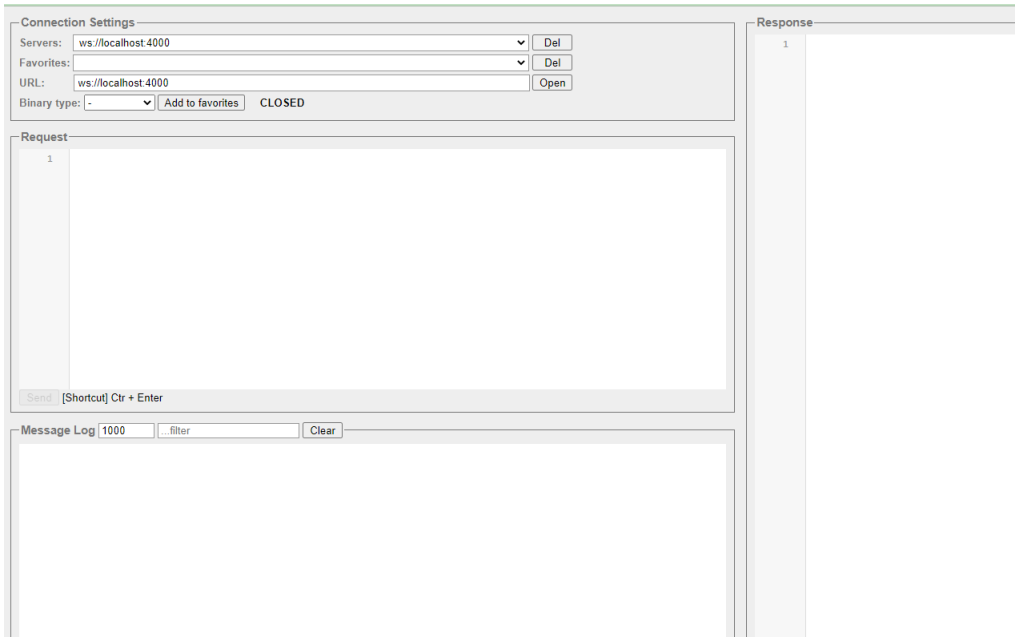


Рисунок 5.9 – Інтерфейс Advanced WebSocket Client

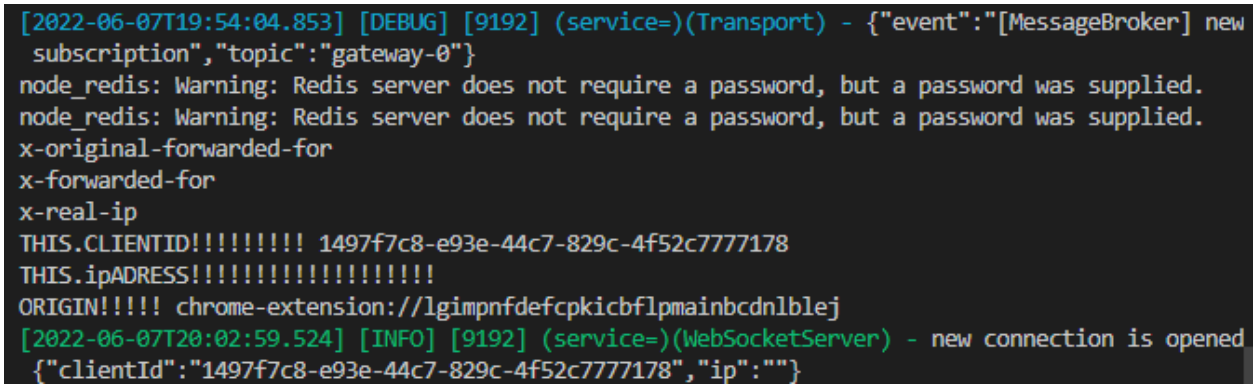


Рисунок 5.10 – Повідомлення в API Gateway про нове підключення

Тепер можна перевірити реєстрацію, яка складається с 3х методів. На рисунку 5.11 зображено перший запит createRequestToConfirmEmail.

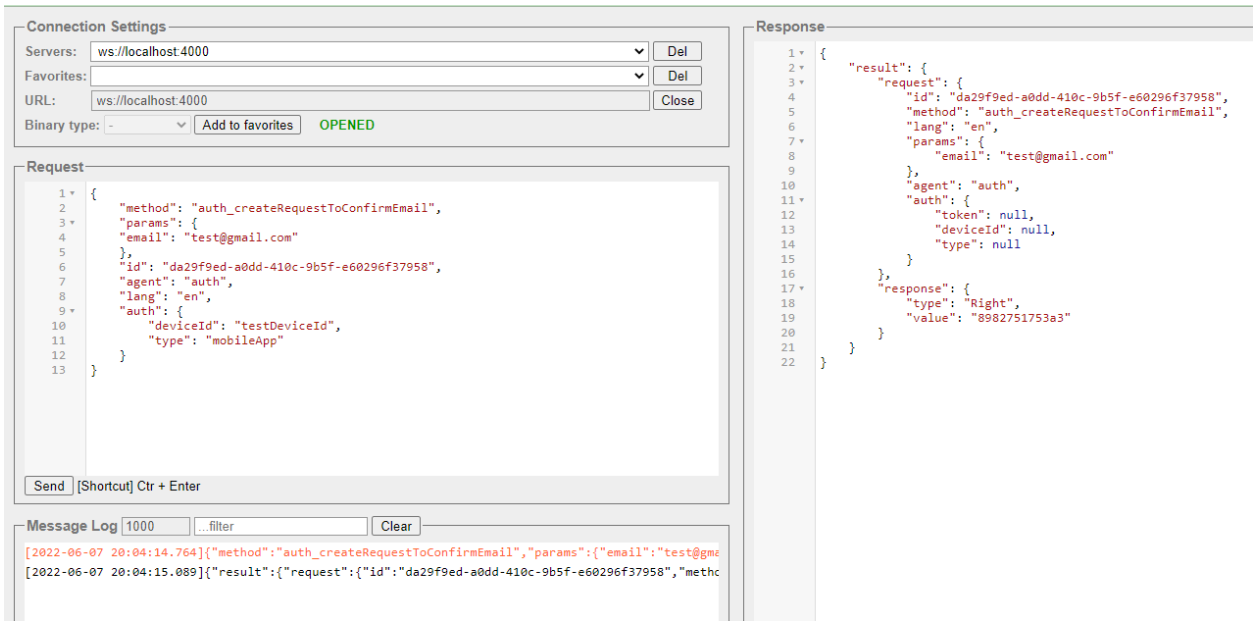


Рисунок 5.11 – Тестування запиту createRequestToConfirmEmail

З рисунку 5.11 видно, що запит спрацював успішно і повернув значення, яке потім потрібно використати в наступному запиті confirmEmailByCode (рисю 5.12). Значення code береться з таблиці confirm_emails, в майбутньому це значення буде відсилатися на пошту, яку вказав користувач.

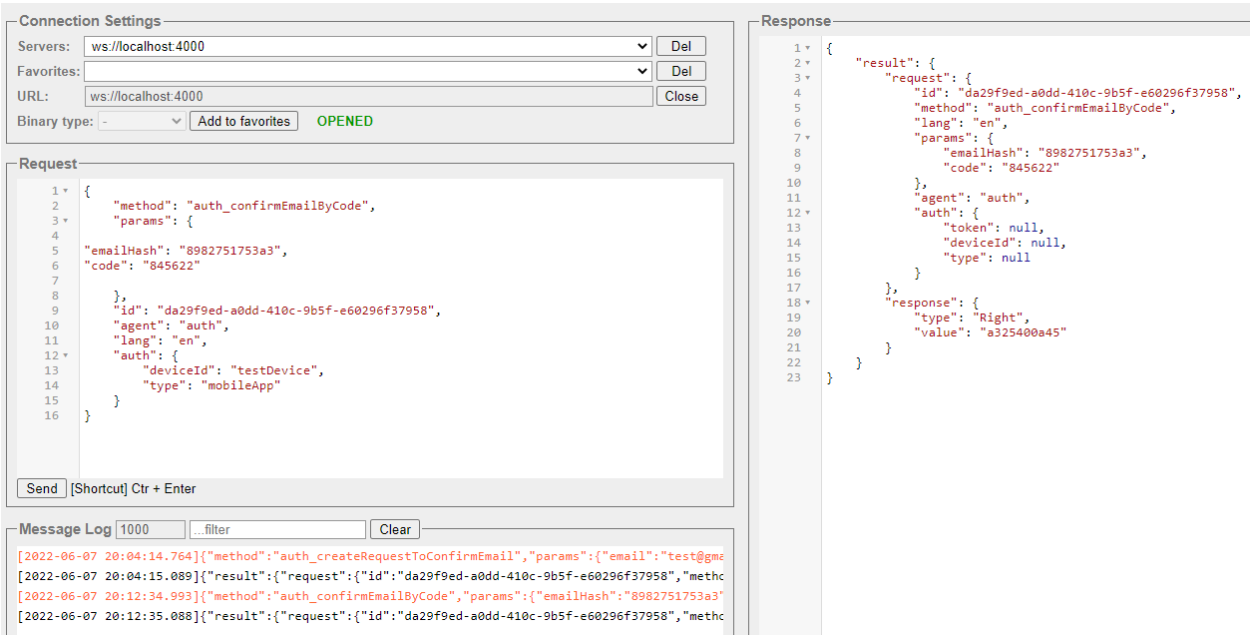


Рисунок 5.12 – Тестування запиту confirmEmailByCode

Метод відпрацював успішно, тепер черга останнього методу setPassword (рис. 5.13). Цей метод повертає усі необхідні дані для клієнту, які він потім використовує для авторизації.

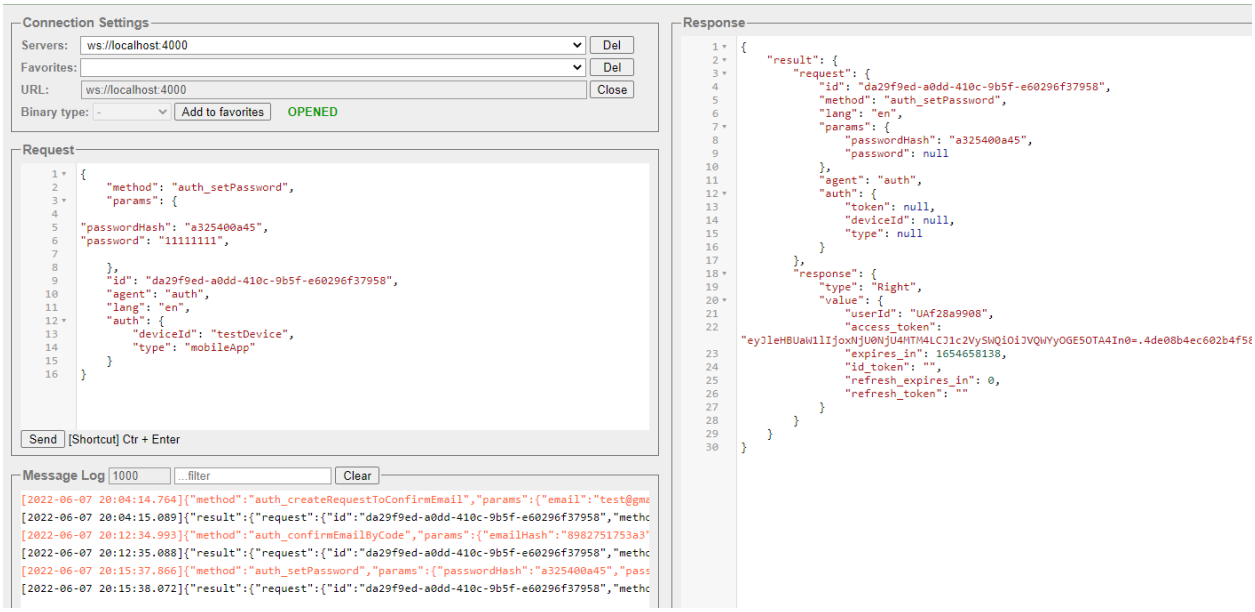


Рисунок 5.13 – Тестування запиту setPassword

Переконавшись в наявності записів у таблицях users, web_sessions в базі даних auth та таблицю profiles в базі даних user, можна стверджувати, що

реєстрація працює. Після реєстрації був перевірений ще й метод логіну (рис. 5.14).

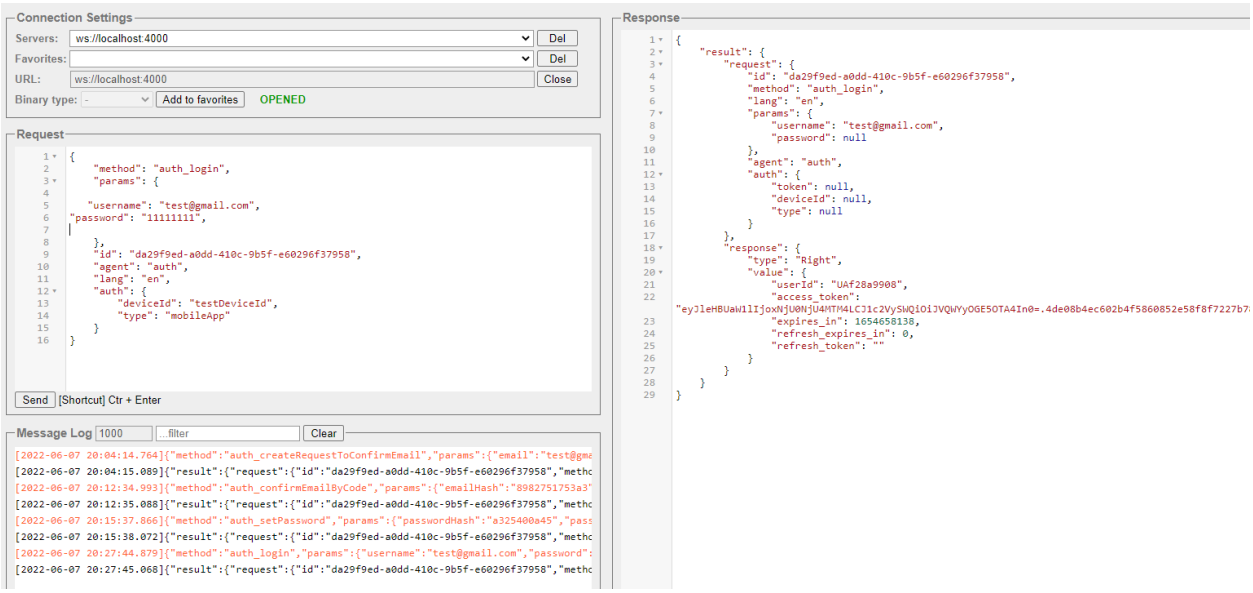


Рисунок 5.14 – Тестування запиту login

Після вводу неправильного email з’явилась помилка про неіснуючого користувача (рис.5.15), або неправильний пароль – помилка про некоректний пароль (рис. 5.16).


```
{
    "result": {
        "request": {
            "id": "da29f9ed-a0dd-410c-9b5f-e60296f37958",
            "method": "auth_login",
            "lang": "en",
            "params": {
                "username": "tesat@gmail.com",
                "password": null
            },
            "agent": "auth",
            "auth": {
                "token": null,
                "deviceId": null,
                "type": null
            }
        },
        "response": {
            "type": "Left",
            "value": {
                "type": "db.notFound",
                "msg": "User not found",
                "status": 18291
            }
        }
    }
}
```


Рисунок 5.15 – Помилка про неіснуючого користувача

```
{
    "result": {
        "request": {
            "id": "da29f9ed-a0dd-410c-9b5f-e60296f37958",
            "method": "auth_login",
            "lang": "en",
            "params": {
                "username": "test@gmail.com",
                "password": null
            },
            "agent": "auth",
            "auth": {
                "token": null,
                "deviceId": null,
                "type": null
            }
        },
        "response": {
            "type": "Left",
            "value": {
                "type": "auth.invalidCredentials",
                "msg": "Invalid password",
                "status": 18560
            }
        }
    }
}
```


Рисунок 5.16 – Помилка про некоректний пароль

Таким чином, можна перевірити всі методи і тим самим виявити, які саме помилки трапляються.

Під час тестування виникало достатньо помилок і потрібно було їх виправляти. Для швидшого виправлення та для зменшення кількості помилок можна виділити наступні пункти:

а) детальне та правильне логування помилок, виключень;

б) уважність під час написання запитів до бази даних;

в) перед інтегрованим тестуванням слід спочатку зайнятися модульним тестуванням, якщо на то є час.

Загалом, інтеграційне тестування — це хороший спосіб захиститися від помилок, які можуть з’явитися у системі. Це також заощаджує час на розробку та налагодження, оскільки будь-яку проблему можна виправити після невдачі в тестуванні функції або набору функцій.

5.3 Висновки за розділом

В цьому розділі проведено тестування програмного засобу різними способами, а також показано його придатність до використання. Продемонстровано основні принципи роботи з сервісами.

ВИСНОВКИ

В результаті проведеного вивчення питання щодо програмного забезпечення для контролю персональних даних користувачі було поставлена задача створити ссерверну частину для з використанням технології Node.js, PostgreSQL та мови TypeScript на основі мікросервісної архітектури.

Мікросервісна архітектура надає більше можливостей і більше рішень, які потрібно приймати. Виготовлення прийняття рішень у цій архітектурі є набагато більш поширеною діяльністю, ніж у більш простій, монолітній системі.

Під час роботи над проєктом мікросервісна архітектура продемонструвала свої плюси і якщо потрібно розширювати проєкт, то тепер це стає більш зручніше. Підтримка таких проєктів розгалужується на декілька команд, які можуть працювати окремо одна від одної.

Отже, можна виділити наступні пункти, які допоможуть краще проникнути до такої архітектури:

а) потрібно знаходити способи зробити кожне рішення невеликим за обсягом, таким чином, якщо ви помилитеся, ви впливаєте лише на невелику частину вашої системи;

б) навчіться приймати концепцію еволюційності архітектури, в якій ваша система розвивається, працює і змінюється з часом коли дізнаєшся щось нове;

в) потрібно думати не про великі переписи, а про серію змін, внесені у вашу систему з часом, щоб підтримувати її працездатність.

Таким чином це дало можливість створити програмне забезпечення під час роботи над проєктом. Тестування показало коректну роботу розробленої серверної частини.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. About NATS [Електроний ресурс]. – Режим доступу: https://nats.io/about/
2. About Node.js [Електроний ресурс]. – Режим доступу: https://nodejs.org/en/about/
3. Dependency Injection for dummies? [Електроний ресурс]. – https://8r14z.medium.com/dependency-injection-for-dummies-168dad181a3d
4. ESLint Philosophy [Електроний ресурс]. – https://eslint.org/docs/about/#philosophy
5. Gamma E. Design Patterns: Elements of Reusable Object-Oriented Software / E. Gamma, J. Vlissides – Addison-Wesley Professional, 1994. – 540 p.
6. Glenford J. The Art of Software Testing / J. Glenford – Wiley; 3rd edition, 2011. – 256 p.
7. Introduction to Redis [Електроний ресурс]. – https://redis.io/docs/about/
8. Learn 5 defensive programming techniques from experts [Електроний ресурс]. – https://www.techtarget.com/searchsoftwarequality/feature/Learn-5-defensive-programming-techniques-from-experts
9. Lombardi A. WebSocket: Lightweight Client-Server Communications / A. Lombardi – O'Reilly Media, 2015. – 144 p.
10. Microservice Architecture — Learn, Build, and Deploy Applications [Електроний ресурс]. – https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a
11. Microservices vs Monolith: which architecture is the best choice for your business? [Електроний ресурс]. – Режим доступу: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/
12. Mitra R. Microservice Architecture: Aligning Principles, Practices, and Culture / R. Mitra, M. McLarty – O'Reilly Media, 2016. – 146 p.

13. Monad (functional programming) [Електроний ресурс]. – https://en.wikipedia.org/wiki/Monad_(functional_programming)

14. Newman S. Building Microservices: Designing Fine-Grained Systems / S. Newman – O'Reilly Media, 2021. – 500 p.

15. Newman S. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith / S. Newman – O'Reilly Media, 2019. – 272 p.

16. Nickoloff J. Docker in Action, Second Edition / J. Nickoloff, S. Kuenzli – Manning; 2nd edition, 2019. – 336 p.

17. Pattern: Microservice Architecture [Електроний ресурс]. – https://microservices.io/patterns/microservices.html

18. Riggs S. PostgreSQL 14 Administration Cookbook: Over 175 proven recipes for database administrators to manage enterprise databases effectively / S. Riggs – Packt Publishing, 2022. – 608 p.

19. Semaphores — Break the ice with concurrency! [Електроний ресурс]. – https://medium.com/@kiitvishal89/semaphores-break-the-ice-with-concurrency-fc1cee0f4e2f

20. Vanderkam D. Effective TypeScript: 62 Specific Ways to Improve Your TypeScript / D. Vanderkam – O'Reilly Media, 2019. – 264 p.

21. What is TypeORM? [Електроний ресурс]. – https://typeorm.io/

22. Что такое API Gateway: введение [Електроний ресурс]. – https://highload.today/api-gateway-endpoints/

Додаток А

Зауваження нормоконтролера

Таблиця А.1 – Зауваження нормоконтролера

Сторінка	Позначення	Зміст зауваження
ПЗ		Нещільне заповнення сторінок, рисунки з двох сторін - порожніми рядками
ПЗ		Назви підрозділі, пунктів тощо - з червоного рядка
С. 19,24, 32,…		Невірний перелік
С. 61-62		Помилки в оформленні джерел

Додаток Б

Графічні матеріали

ДВНЗ «ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ»
Факультет комп'ютерно-інформаційних технологій та автоматизації
Кафедра прикладної математики і інформатики

Випускна кваліфікаційна робота
бакалавр
на тему:
«Розробка мобільних мікросервісів для контролю персональних даних користувачів»
зі спецчастиною:
«Розробка серверної частини, програмних модулів, баз даних засобами TypeScript, Node.js, PostgreSQL»

Виконав: студент 4-го курсу, групи ІПЗз-18
Гриценко О.С.
Керівник зав. каф. ПМІ, д.т.н., проф.
Дмитрієва О.А.

Покровськ, Луцьк – 2022 р.

Постановка завдання

Об'єкт дослідження: процес збереження персональних даних користувачів при вирішенні завдання побудови серверної частини з використанням мікросервісної архітектури.

Предметом дослідження: серверна частина на основі мікросервісів, орієнтована на безпечне збереження даних користувачів за умови процесу реєстрації та авторизації у системі мобільного додатку.

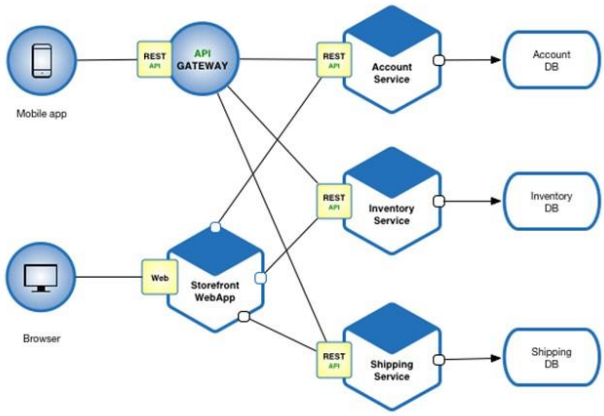
Мета: розробка серверної частини для безпечного та зручного збереження даних користувачів з подальшим простим розширенням розробленої системи.

Задачі

В процесі розробки були вирішені наступні задачі:

1) зробити огляд мікросервісної архітектури;
2) визначити необхідні функції для реалізації серверної частини;
3) вибрати необхідний технологічний стек для реалізації серверу та реалізувати серверну частину;
4) провести тестування розробленої системи.

Приклад архітектури електронної комерції з використанням мікросервісів



Ключові концепції мікросервісної архітектури

1. Незалежне розгортання
2. Сформовано навколо бізнес-домену
3. Володіння власним станом

Переваги мікросервісної архітектури

1) вирішує проблему складності, розкладаючи систему на набір керованих служб, які набагато швидше розробляються та набагато легші для розуміння та підтримки;
2) дає змогу окремо розробляти кожен функціонал командою, яка зосереджена на цьому сервісі;
3) зменшує бар'єр прийняття нових технологій, оскільки розробники можуть вільно вибирати будь-які технології, які мають сенс для їх обслуговування, і не обмежені вибором, зробленим на початку проєкту;
4) дозволяє розгортати кожен мікросервіс незалежно. В результаті це робить можливим безперервне розгортання для складних додатків;
5) дозволяє незалежно масштабувати кожен сервіс.

Недоліки мікросервісної архітектури

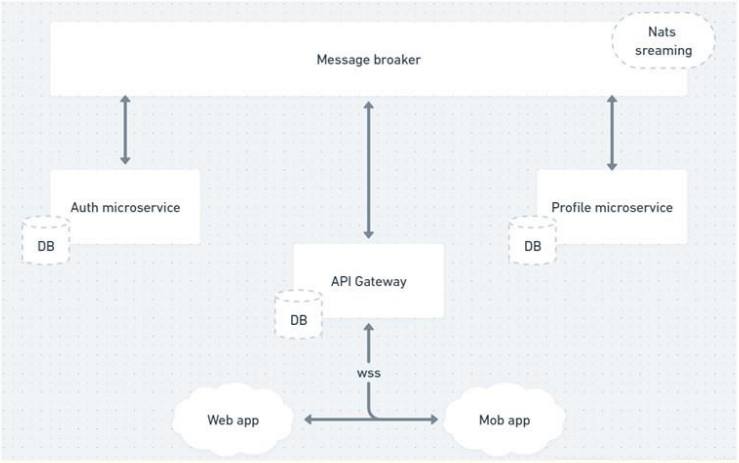
1) ускладняється проєкт;
2) роздільна архітектура баз даних;
3) тестування програми мікросервісів також набагато складніше, ніж у випадку монолітного проєкту;
4) складніше впровадити зміни, які охоплюють декілька сервісів;
5) розгортання програми на основі мікросервісів також є більш складним.

Розробка архітектури програмної системи

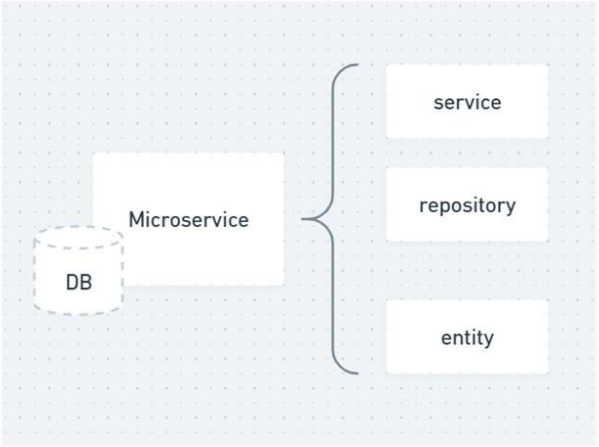
Стек:

1. Node.js;
2. TypeScript;
3. NATS;
4. PostgreSQL, TypeORM;
5. Redis.

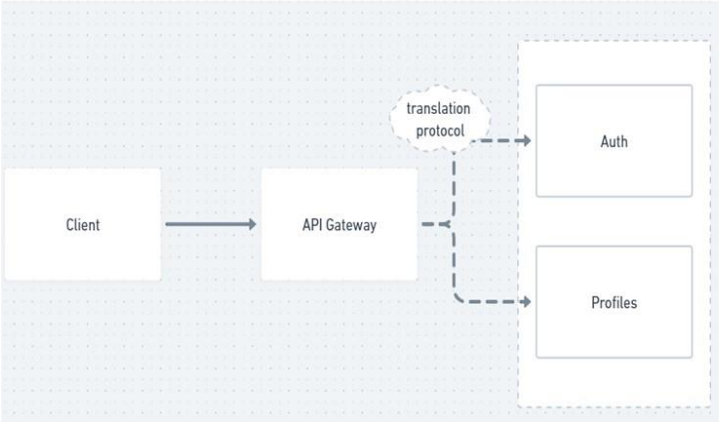
Архітектура розробленої мікросервісної системи



Основна структура мікросервісу



Взаємозв’язок API Gateway з іншими мікросервісами та користувачем



Перевірка та безпека даних

```
login: {
  accessLevel: 'public',
  schema: Type.Object({
    email: Type.String(),
    password: Type.String({ minLength: 6 }),
    deviceId: Type.Optional(Type.String()),
    appId: Type.Optional(IntType()),
  }),
  returnSchema: Type.Object({
    access_token: Type.String(),
    expires_in: IntType(),
    refresh_expires_in: IntType(),
    refresh_token: Type.String(),
    userId: Type.String(),
  }),
},
```

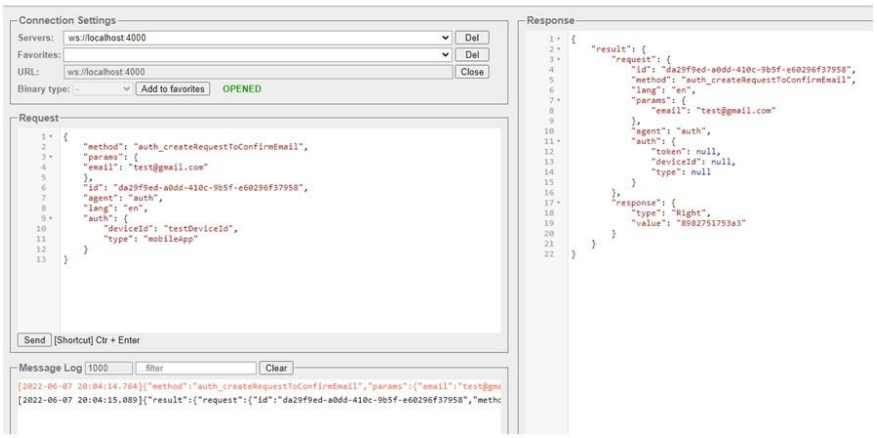
Приклад схеми для методу login

Система ролі для користувачів

```
if (isInternal) return right({ info: {}, authorized: false, infoLoaded: false });

if (isPublic) return right({ info: {}, authorized: true, infoLoaded: false });

if (isProtected && !request.auth.token) {
  return left({
    type: errorsRegistry.auth.accessDenied,
    msg: 'Not enough permissions',
  });
}

if (isPrivate && !request.auth.token) {
  return left({
    type: errorsRegistry.auth.accessDenied,
    msg: 'Not enough permissions',
  });
}
```

Базова перевірка рівня доступу методів

Тестування розробленої системи



Тестування запиту createRequestToConfirmEmail

Висновки

1) потрібно знаходити способи зробити кожне рішення невеликим за обсягом, таким чином, якщо ви помилитеся, ви впливаєте лише на невелику частину вашої системи;
2) навчітися приймати концепцію еволюційності архітектури, в якій система розвивається, працює і змінюється з часом коли дізнаєшся щось нове;
3) потрібно думати не про великі переписи, а про серію змін, внесені у систему з часом, щоб підтримувати її працездатність.