

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Комп'ютерних наук і технологій

(повне найменування факультету)

Комп'ютерні системи та мережі

(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

магістра

(ступінь вищої освіти)

на тему КОМП'ЮТЕРНА СИСТЕМА АУТЕНТИФІКАЦІЇ КОРИСТУВАЧІВ
НА ОСНОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

Виконав: студент 2 курсу, групи КНТ-511м
спеціальності _____

123 «Комп'ютерна інженерія»

Кострикін М.М.

(прізвище та ініціали)

Керівник Скрупський С.Ю.

(прізвище та ініціали)

Рецензент Єфименко М.В.

(прізвище та ініціали)

м. Запоріжжя
2022 рік

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування вищого навчального закладу)

Інститут, факультет інформатики та радіоелектроніки, комп'ютерних наук і технологій
 Кафедра «Комп'ютерні системи та мережі»
 Ступінь вищої освіти (освітній ступінь) магістерський
 Спеціальність 123 Комп'ютерна інженерія
 (код і найменування)
 Освітня програма (спеціалізація) Комп'ютерні системи та мережі
 (назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри Кудерметов Р.К.« » 2022 року

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

Кострикіна Максима Максимовича

(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) Комп'ютерна система аутентифікації користувачів на основі мікросервісної архітектури

керівник проєкту (роботи) Скрупський Степан Юрійович, к. т. н., доцент
 (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від «7» 11 2022 року № 357

2. Строк подання студентом проєкту (роботи) 13 грудня 2022 року

3. Вихідні дані до проєкту (роботи) Наявні методи та алгоритми реалізації функціональних вузлів під мікросервісну архітектуру. Алгоритми авторизації та аутентифікації користувачів. Технології побудови сервісів JavaScript, NodeJS, Express, PostgreSQL, Sequelize.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1) Аналіз технічного завдання та інструментів для реалізації проєкту;

2) Розробка архітектури проєкту та програмних модулів;

3) Реалізація системи зі спроектованими алгоритмами;

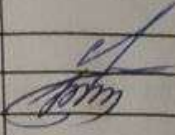
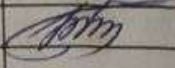
4) Дослідницький аналіз розробленої системи.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Структурна схема розробленої системи;

Порівняльний аналіз ефективності розроблених алгоритмів;

Розроблені алгоритми

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	приймав виконане завдання
1-4	Скрупський С.Ю., доцент		
Нормоконтроль	Щербак Н.В., ст. викл.		

7. Дата видачі завдання 01.09.2022 р.

КАЛЕНДАРНИЙ ПЛАН

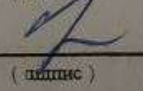
№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітки
1	Аналіз наявних методів реалізації, порівняння існуючої елементної бази, виділення недоліків та переваг	01.09.2022 р.	
2	Розробка архітектури проєкту	25.09.2022 р.	
3	Розробка алгоритмів роботи системи	10.10.2022 р.	
4	Розробка програмних модулів	16.10.2022 р.	
5	Порівняльний аналіз розробленої системи з існуючими рішеннями та виведення висновків щодо її ефективності	20.10.2022 р.	
6	Розробка макетів сторінок системи	27.10.2022 р.	
7	Оформлення отриманих результатів у ПЗ	03.11.2022 р.	
8	Оформлення графічного матеріалу	05.11.2022 р.	
9	Оформлення допоміжного матеріалу	07.11.2022 р.	

Студент



М.М. Кострикін
(ініціали та прізвище)

Керівник проєкту (роботи)



С.Ю.Скрупський
(ініціали та прізвище)

РЕФЕРАТ

ПЗ: 99 с., 37 рис., 5 таблиць, 17 джерел.

JAVASCRIPT, NODEJS, EXPRESS, POSTGRES, REST, JWT,
МІКРОСЕРВІСИ

Об'єкт дослідження – механізми аутентифікації, авторизації та ідентифікації користувачів.

Предмет дослідження – комп'ютерна система аутентифікації користувачів на базі мікросервісної архітектури.

Мета дослідження – підвищення продуктивності методів аутентифікації та авторизації на базі систем з мікросервісною архітектурою.

Для досягнення поставленої мети було проаналізовано методи реалізації аутентифікації та контролю доступом; спроектовано функціональну схему пристрою; спроектовано алгоритми роботи API-шлюзу, керування сесіями та контролю доступом; розроблено мікросервісну систему, яка реалізує спроектовані алгоритми; здійснено дослідження роботи алгоритмів системи.

В результаті виконання роботи було розроблено мікросервісну систему, яка дозволяє створювати користувачів, а також реалізує методи їх аутентифікації, авторизації та верифікації.

ЗМІСТ

ВСТУП.....	7
1 АНАЛІЗ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ РОЗРОБКИ.....	9
1.1 Актуальність.....	9
1.2 Огляд особливостей мікросервісної архітектури.....	10
1.3 Огляд моделей авторизації.....	14
1.4 Міжсервісна аутентифікація.....	22
1.5 Огляд мов програмування.....	23
1.6 Аналіз технологій СУБД.....	29
1.7 Висновки аналізу інструментів і технологій.....	32
2 ПРОЄКТУВАННЯ АЛГОРИТМІВ СИСТЕМИ.....	33
2.1 Огляд існуючих підходів реалізації аутентифікації.....	33
2.3 OAuth 2.0 і OpenID Connect.....	42
2.4 Огляд існуючих готових рішень.....	44
2.5 Проектування розроблюваних алгоритмів.....	48
2.6 Висновки проектування алгоритмів.....	62
3 РЕАЛІЗАЦІЯ АЛГОРИТМІВ ТА ТЕСТУВАННЯ СИСТЕМИ.....	63
3.1 Реалізація алгоритму API-шлюзу.....	63
3.2 Реалізація сервісу керування сесіями.....	67
3.3 Реалізація сервісу авторизації.....	78
3.4 Тестування системи.....	80
3.5 Висновки з реалізації та тестування алгоритмів.....	84
4.1 Методи оцінювання продуктивності мікропроцесорної системи.....	85
4.2 Визначення показників оцінювання продуктивності системи.....	87

4.2 Оцінка продуктивності системи.....	88
4.3 Висновки проведеного дослідження.....	93
ВИСНОВКИ.....	96
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ.....	98

Перелік графічних матеріалів:

Креслення 1 – Алгоритм оновлення токена сесії

Креслення 2 – Алгоритм створення сесії

Креслення 3 – Алгоритм створення користувача

Креслення 4 – Алгоритм видалення сесії

Креслення 5 – Алгоритм перевірки прав доступу

Креслення 6 – Алгоритм роботи API-шлюзу

Креслення 7 – Структурна схема пристрою

Креслення 8 – Діаграма дослідження витрат часу на виконання повної ітерації роботи системи

Креслення 9 – Діаграма дослідження витрат часу на виконання запиту системою

ВСТУП

Питання безпечного використання комп'ютерних систем було і залишається достатньо гострим. Процедура перевірки того, чи являється користувач тим, за кого себе видає є одною з провідних та найпоширеніших у сучасному ІТ секторі. Тому надійний та відносно простий у використанні захист користувацьких даних все ще залишається актуальним і являється метою дослідження для багатьох спеціалістів.

Архітектура мікросервісів все частіше починає застосовуватися для проектування та впровадження веб-систем, великомасштабних додатках і службах. На етапі розробки та впровадження програмного продукту перед спеціалістами ставиться задача вирішення багатьох проблем безпеки. І одною з ключових з таких проблем є саме аутентифікація та авторизація користувача. Вибраний підхід до реалізації цього питання значною мірою впливає як на кінцевий користувацький досвід, так і на безпосередньо процес розробки та підтримки продукту. А тому вкрай важливо розуміти та правильно використовувати існуючі архітектурні шаблони та вміти їх модифікувати в залежності від поставлених задач.

Використання мікросервісної архітектури під час проектування програмного забезпечення збільшує можливості для проведення атаки та ймовірність успішної реалізації загроз безпеки інформації. Розвиток ідей мікросервісної побудови програмного забезпечення та супутніх з цим технологій призвели до появи різноманітних підходів у реалізації архітектури безпеки сучасних рішень, і як наслідок відсутність єдиного підходу до її побудови. Натомість наразі вже існують технологічні лідери в галузі мікросервісної розробки, є добре відомі недоліки вразливості у реалізації та особливості конфігурації різних архітектурних підходів, і є безліч найрізноманітніших підходів для побудови надійної архітектури.

Задачею даної роботи є дослідження та вдосконалення наявних методів аутентифікації користувача при роботі з веб-додатками на базі мікросервісної архітектури, розбір типових архітектурних підходів до реалізації аутентифікації та авторизації у мікросервісних системах, виявлення їх переваг та недоліків з метою подальшого вдосконалення. За мету ставиться отримання покращених архітектурних підходів до аутентифікації та отримання порівняльної характеристики ефективності існуючих архітектурних шаблонів з їх вдосконаленими варіантами.

Для досягнення поставленої задачі необхідно виконати наступні завдання:

- провести аналіз існуючих рішень;
- спроектувати алгоритми та на їх основі розробити сервіси;
- провести аналіз отриманих результатів.

1 АНАЛІЗ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ РОЗРОБКИ

Важливою задачею, яка стоїть перед початком виконання будь-якого проєкту є огляд та аналіз програмних засобів, інструментів та методів, які будуть використовуватися при розробці. В поточному розділі аналізується актуальність розглядаємого питання задача, розглядаються сучасні підходи, відбувається вибір використовуваного програмного комплексу, який буде застосовано при розробці.

1.1 Актуальність

Розвиток концепції архітектури мікросервісів був однією з головних тенденцій у розробці програмних додатків за останні кілька десятиліть. Термін "Мікросервіс", або "архітектура мікросервісів" вперше був введений у 2010 році для опису певного стилю архітектури, який можна використовувати для розробки веб-додатків. Завдяки появі різноманітних гнучких практик концепція архітектури мікросервісу набула популярності. Вона забезпечує плавне горизонтальне масштабування програм і дає команді розробників повну відповідальність за розроблений сервіс. Розповсюдження цього архітектурного підходу призводить до того, що питання надійності та безпечності використання стає край важливим.

Відповідно до опитування O'Reilly [1], у якому прийняли участь більш ніж 1500 респондентів, проведеного 31 січня 2020 року, більшість респондентів (92%) вважають свій досвід роботи з архітектурою мікросервісів достатньо успішним, але в свою чергу, в той самий час згідно з цим опитуванням, лише 21% респондентів стурбовані безпекою в системах з мікросервісною архітектурою. У 2019 році Open Web Application Security Project (OWASP) переглянула методологію тестування безпеки API [2]. Методологія фокусується на питаннях автентифікації та авторизації при використанні методів API.

Приведені дані говорить про те, наскільки питання безпечного використання мікросервісних веб-додатків є недооціненим, незважаючи на важливість цієї сфери, адже недостатній рівень захищеності внутрішніх даних компаній чи персональних даних користувача може стати причиною значних репутаційних та фінансових втрат, а також серйозно нашкодити користувачам платформи.

1.2 Огляд особливостей мікросервісної архітектури

Архітектурний стиль мікросервісів — це підхід, при якому єдиний монолітний додаток поділяється на сукупність окремих, невеликих, незалежних та самодостатніх сервісів, які слабо пов'язані між собою та спілкуються за допомогою таких веб-протоколів як HTTP, AMQP, gRPC [3]. Кожен сервіс має окреме призначення та будується заради конкретної бізнес-потреби та розгортається без якої-небудь залежності від всієї іншої системи з використанням автоматизованого середовища. Коли починає йти мова про забезпечення гнучкої розробки складних корпоративних додатків — такий спосіб розробки має значні переваги.

Заради кращого розуміння особливостей підходу мікросервісної архітектури потрібно порівняти цю архітектуру з монолітним стилем розробки програмного забезпечення.

Монолітний стиль — це класичний підхід до розробки програмного забезпечення, який полягає у тому, що увесь додаток будується єдиним модулем і працює незалежно і автономно від інших додатків. Моноліт називають великим і неповоротким, що якісно описує цю архітектуру для проектування програмного забезпечення. Будь-які зміни, навіть самі невеликі, які вносяться до розроблюваного додатку, потребують перебудови та розгортання всього проєкту в цілому. З часом це призводить до того, що стає значно складніше зберігати хорошу модульну структуру проєкту, вносити зміни до логіки окремого модуля,

не впливаючи тим самим на код інших модулів. Також, монолітні додатки може бути достатньо важко масштабувати, коли різні модулі системи будуть мати конфліктні вимоги до ресурсів. При цьому масштабувати доводиться увесь додаток, навіть при умові, що цього потребує лише тільки декілька окремих модулів додатку.

Моноліт зручно використовувати на початкових етапах проектів, так як такий підхід полегшує розгортання невеликих додатків та не витрачає занадто багато ресурсів для проектування складної логіки взаємодії окремих компонентів системи. Це дозволяє швидко випускати все, що є в складі монолітного додатку у кінцевому варіанті. Мікросервіси, на противагу цьому, розгортаються окремо та незалежно, не чіпаючи інші модулі.

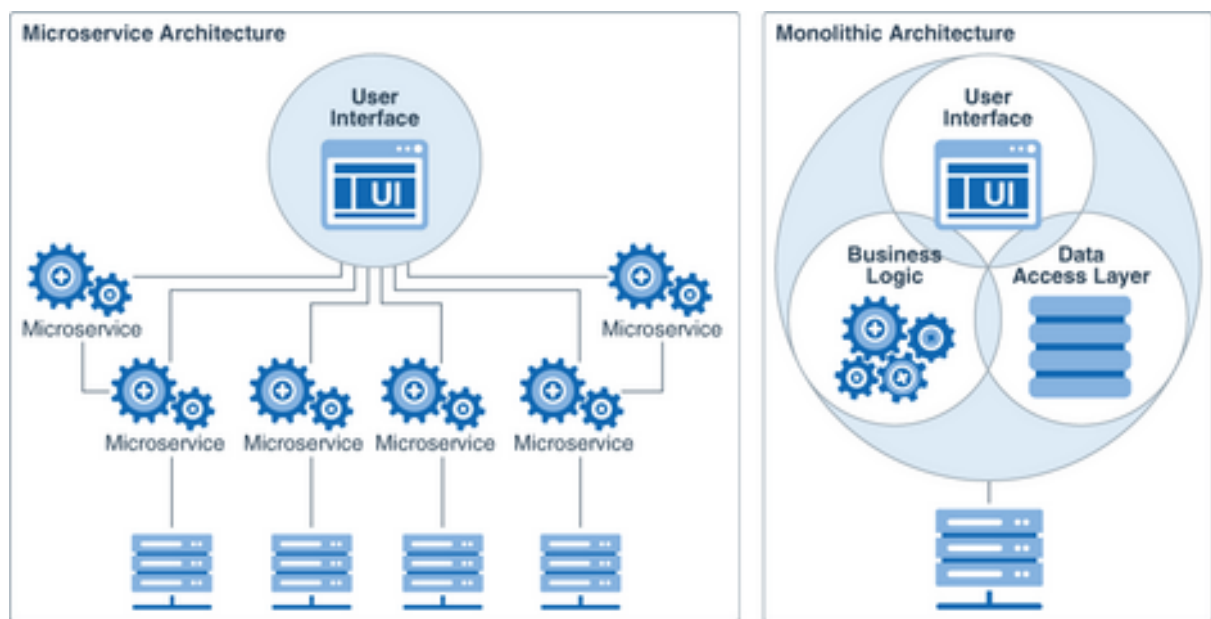


Рисунок 1.1 — Порівняння мікросервісів з монолітом

Використовуючи як монолітну, так і мікросервісну архітектуру можна отримати свою вигоду в залежності від різних факторів. Наприклад, при використанні монолітного стилю зручніше використовувати одну базу коду, тому основною перевагою цієї архітектури полягає у швидкості розробки. Але, в свою чергу з часом моноліти, як правило, в процесі довготривалої підтримки, перетворюються у нагромадження великої кількості коду, в якій стає занадто

складно орієнтуватися, що значно впливає на швидкість розробки та підтримки проєкту. Також проблемою є неможливість оновлення ядра системи, що стає перешкодою для впровадження нових технологій, складність масштабування при довготривалій розробці і проблеми з безпекою через практично повну відсутність якої-небудь ізоляції компонентів. Таким чином проблема або помилка в окремому модулі може уповільнити або зруйнувати всю програму.

Мікросервіси не є універсальним рішенням усіх наявних у монолітних проблем, з якими стикаються розробники програмного забезпечення. Але завдяки особливостям підходу цей стиль надає значні переваги для великих проєктів, підвищуючи надійність системи в обмін на ускладнення процесу проектування. Цей стиль надає низку суттєвих переваг, серед яких, наприклад, можна виділити наступні:

- значне підвищення загальної швидкодії та продуктивності сервісу за рахунок асинхронної обробки вхідних запитів;
- можливість використання будь-яких мов програмування та баз даних. Таким чином, один сервіс може бути написаний на мові Java, а для іншого сервісу, який вимагає кращої продуктивності, можна використовувати мови Golang або C++;
- можливість масштабувати лише ті сервіси, які цього потребують, а в свою чергу інші частини системи запускати на менш потужному обладнанні;
- можливість вносити зміни в окремий сервіс та швидко публікувати його незалежно від всієї системи. З цього слідує, що нові функціональні можливості, або вирішення існуючих помилок здійснюється набагато швидше і оновлення скоріше потрапляють до кінцевих користувачів.
- при виході з ладу одного або декількох сервісів система в цілому зберігає свою працездатність. Таким чином спрощується виявлення проблеми та її ізоляція та вирішення [4].

Незважаючи на перелічені переваги, все ж таки мікросервісні системи мають і свої складності: значним ускладненням процесу тестування та взаємодії

різних команд, з більш високою складністю при реалізації взаємодії між сервісами. Також до недоліків такого підходу можна віднести і наступні особливості:

- мікросервіси значно ускладнюють процес розробки порівняно з монолітною архітектурою, оскільки з часом виникає дедалі більше служб, що були створені різними командами. Якщо розростання не контролюється належним чином, це призводить до зниження швидкості розробки та операційної ефективності;
- підвищення організаційних витрат для команд, які потребують додаткового рівня комунікації та співробітництва, щоб координувати роботу над оновленнями системи та розробки спільних інтерфейсів;
- експоненційне зростання витрат на інфраструктуру, коли кожен новий мікросервіс додатково має у своєму складі закладену вартість комплекту тестів, інструкцій з розгортання, інфраструктури хостингу, інструментів моніторингу тощо;
- проблемою є і налаштування сервісу, коли кожен з них має свій особистий набір інструкцій до розгортання. Крім того, додаткові труднощі можуть виникати у тому випадку, коли один цей процес виконується на кількох машинах;
- також, без єдиної платформи розгортання можливе виникнення ситуації, при якій список мов, стандартів ведення журналів та засобів моніторингу починає значно впливати на стандартизованість системи;
- у міру появи нових сервісів збільшується і кількість команд, працюючих над ними. Це призводить до того, що команді стає складно орієнтуватись, які сервіси вона може використовувати і до кого слід звертатись за підтримкою [5].

Отже, мікросервісна архітектура може підійти не у кожній ситуації, але її розвиток та зручність використання у поточних умовах створення програмного забезпечення призводить до значного зростання популярності. При цьому, якщо застарілий монолітний додаток працює без нарікань, її перетворення на

мікросервіси може не коштувати зусиль. Однак архітектура мікросервісів може виявитися корисною при значному зростанні організації та вимог до розроблюваних додатків. Мікросервісна архітектура, незважаючи на недоліки та проблеми впровадження, є кращим вибором для складних додатків що постійно розвиваються.

1.3 Огляд моделей авторизації

При проектуванні програмного продукту необхідно вирішувати значну кількість проблем безпеки, серед яких аутентифікація та авторизація користувача є однією з ключових, яка в залежності від вибраного підходу може кардинально впливати на підходи до реалізації та кінцевий вигляд додатку. Саме тому дуже важливо чітко орієнтуватися та розуміти які саме існують методи аутентифікації та як і при яких умовах їх правильно використовувати. Метою цього підрозділу як раз і є виявлення та надання характеристики існуючим на сьогоднішній день актуальним підходам.

Серед компонентів системи контролю доступу можна виділити основні, термінологію для опису яких було запроваджено у Національного інституту стандартів та технологій [6], а саме:

- Policy Administration Point (PAP) — точка керування політиками: надає користувацький інтерфейс для створення, управління, тестування та налаштування усіх правил контролю доступу;
- Policy Decision Point (PDP) — точка ухвалення рішення на запит: надає рішення про видачу доступу, оцінюючи визначену політику контролю доступу;
- Policy Enforcement Point (PEP) — точка застосування політики: застосовує визначене рішення у відповідь на запит до захищеного об'єкта;

- Policy Information Point (PIP) — інформаційна точка політики: виступає в якості джерела даних, необхідної для винесення оцінки політики щодо надання інформації, яку використовує PDP для прийняття рішень;
- Policy Retrieval Point (PRP) — виконує зберігання політики авторизації доступу XACML [7].

Зазвичай взаємодію між компонентами моделі організують за допомогою запитів та відповідей у форматі XACML або NGAC. Запит містить атрибути, а відповідь — результат обчислення політик. Політики можуть мати одне з чотирьох значень: "дозволити", "заборонити", "не застосовується", "не визначено". При цьому, відповідь містить у собі зобов'язання, які необхідно виконати у разі дозволу чи заборони доступу.

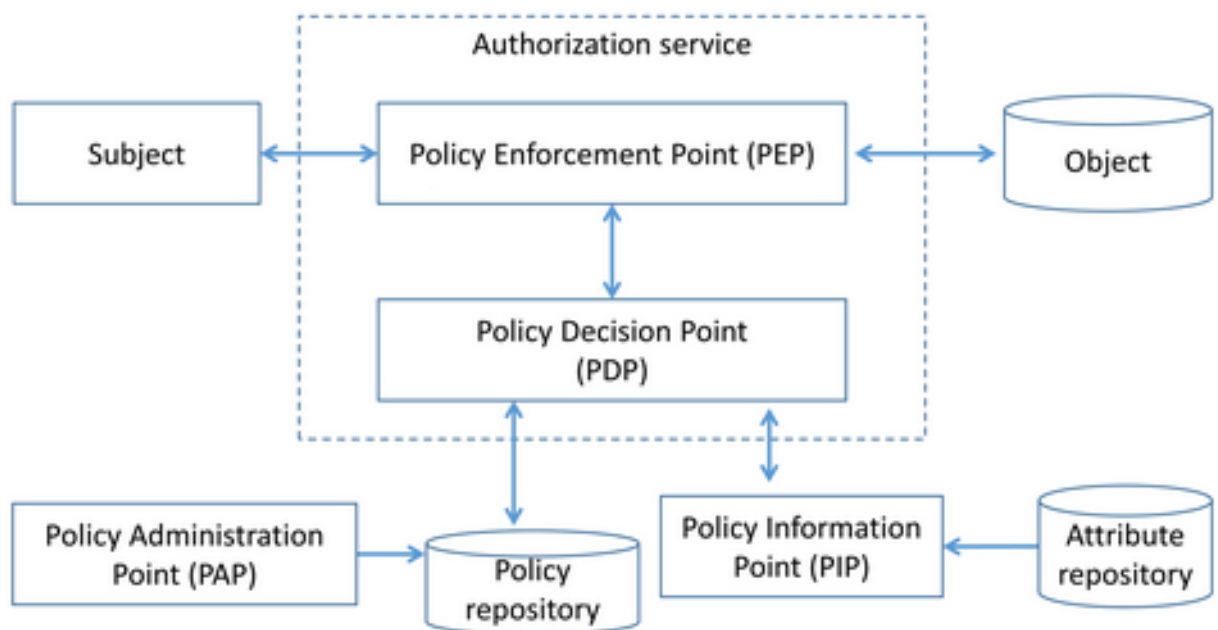


Рисунок 1.2 — Функціональні вузли управління контролем доступу [7]

Взагалі, існує два фундаментальних підходи до реалізації аутентифікації у мікросервісній моделі: на рівні API-шлюзу, що називають пограничною авторизацією, а також безпосередньо на рівні сервісу. Розглянемо обидва варіанти.

1.3.1 Погранична авторизація

У звичайних умовах авторизацію можна проводити виключно на пограничному рівні за допомогою шлюзу API. Такий шлюз використовується як центр авторизації для усіх подальших сервісів, за винятком тих випадків, коли постає необхідність забезпечення аутентифікації для кожного мікросервісу окремо. У цьому випадку рекомендується використовувати полегшені заходи контролю, такі як міжмережеве екранування та взаємна аутентифікація, для запобігання прямих анонімних підключень до внутрішніх служб в обхід шлюзу API.

Однак, в свою чергу, авторизація на пограничному рівні має і ряд недоліків:

- авторизація на цьому рівні може швидко перерости у важко керований процес зі складних мікросервісних системах, у якому буде міститися безліч ролей та правил контролю доступу;
- шлюз API стає єдиною точкою прийняття рішень, що одразу порушує принцип "ешелонованої оборони";
- зазвичай за керування шлюзом API відповідає команда експлуатації, через що команда розробки не може безпосередньо вносити зміни до правил авторизації, це призводить до уповільнення процесу розробки через необхідність міжкомандної взаємодії.

Зазвичай команда розробників реалізує авторизацію у двох місцях: в API шлюзі та безпосередньо на рівні сервісів. Для аутентифікації зовнішніх API можна використовувати токени доступу, або використовувати mTLS.

1.3.2 Авторизація на рівні сервісу

Авторизація на рівні окремого сервісу надає кожному мікросервісу значно більше контролю щодо реалізації наявних політик доступу. Як правило, визначають наступні підходи щодо реалізації авторизації на цьому рівні:

- децентралізована модель;
- централізована модель з єдиною точкою ухвалення рішення;
- централізована модель із вбудованою точкою ухвалення рішення.

1.3.3 Децентралізована модель

При використанні цієї моделі команда розробників реалізує PEP та PDP безпосередньо на рівні сервісу, що зображено на рисунку 1.3. Кожне правило контролю доступу, у тому числі усі атрибути, котрі необхідні для реалізації цих правил, визначаються та зберігаються окремо у кожному сервісі (крок 1). Після того як сервіс отримує запит на доступ разом із деякими метаданими авторизації (крок 2), сервіс обробляє його (крок 3) з метою генерації рішення політики контролю доступу, після чого вже виконує авторизацію (крок 4).

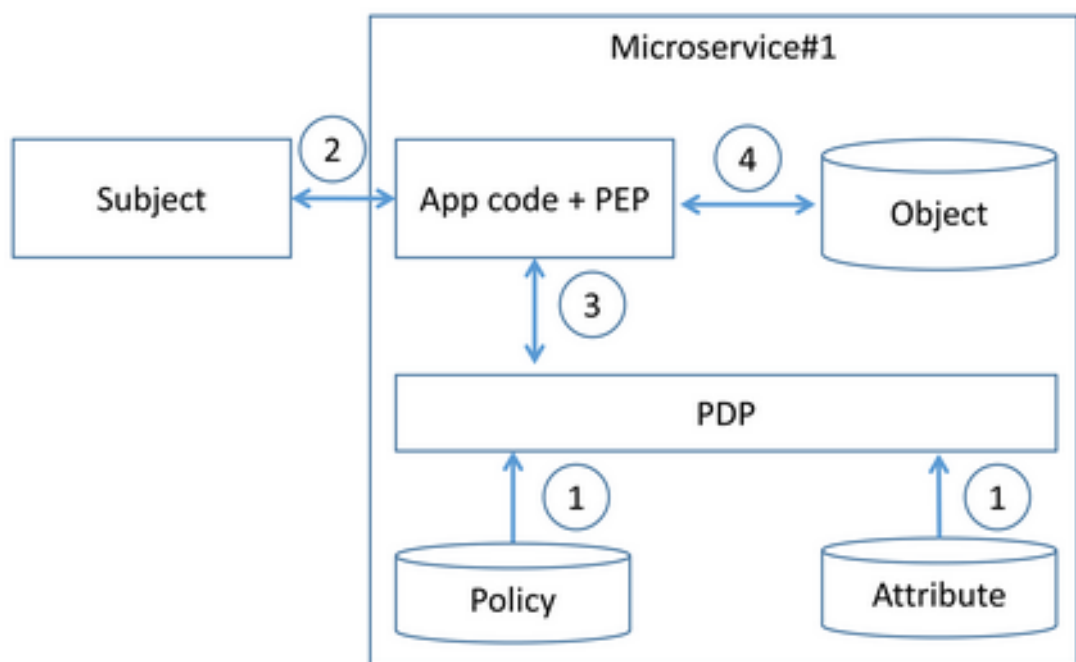


Рисунок 1.3 — Архітектура децентралізованої моделі

Метод реалізації авторизації на рівні мікросервісів є достатньо поширеним і при цьому більшість існуючих бібліотек та фреймворків підтримують цю можливість. Але це і означає, що при зміні логіки авторизації, код повинен оновлюватись щоразу і на кожному з мікросервісів. Як наслідок, даний підхід несе і ряд обмежень та особливостей, а саме:

- відповідальна команда повинна чітко розуміти особливості безпеки при використанні будь-яких фреймворків та правильно реалізовувати наявний функціонал у своїх мікросервісах;

- розробники повинні чітко розуміти політику контролю доступу та наявні у них права доступу для ролі/групи, і це, в свою чергу, може стати значною проблемою через складність усієї розроблюваної системи;
- через значну кількість сервісів у сучасних системах командам розробників може бути важко налаштовувати та підтримувати політики контролю доступу окремо для кожного мікросервісу;
- необхідність проведення тестів після будь-якої зміни вихідного коду.

Але неможливо ігнорувати той факт, що впровадження політики контролю доступу безпосередньо в код мікросервісу дає можливість розробникам значно глибше використовувати усі правила контролю доступу, так як мікросервіс має у собі усе необхідні дані батьківської моделі.

1.3.4 Централізована модель з єдиною точкою ухвалення рішення

Ця модель передбачає визначення, зберігання та оцінювання правил контролю доступу централізовано у єдиному центрі. Схема цієї моделі зображено на рисунку 1.4.

Спочатку у PDP визначаються правила контролю доступу (крок 1), після чого вони доставляються до централізованої PDP разом з необхідними атрибутами доступу, що будуть використовуватися для надання рішення (крок 2). Як тільки користувач викликає мікросервіс (крок 3), через мережевий виклик робиться запит до централізованого PDP, і PDP виносить рішення політики контролю доступу, надаючи оцінку вхідному запиту за визначеними правилами та атрибутами контролю доступу (крок 4). На основі цього рішення мікросервіс виконує авторизацію (крок 5).

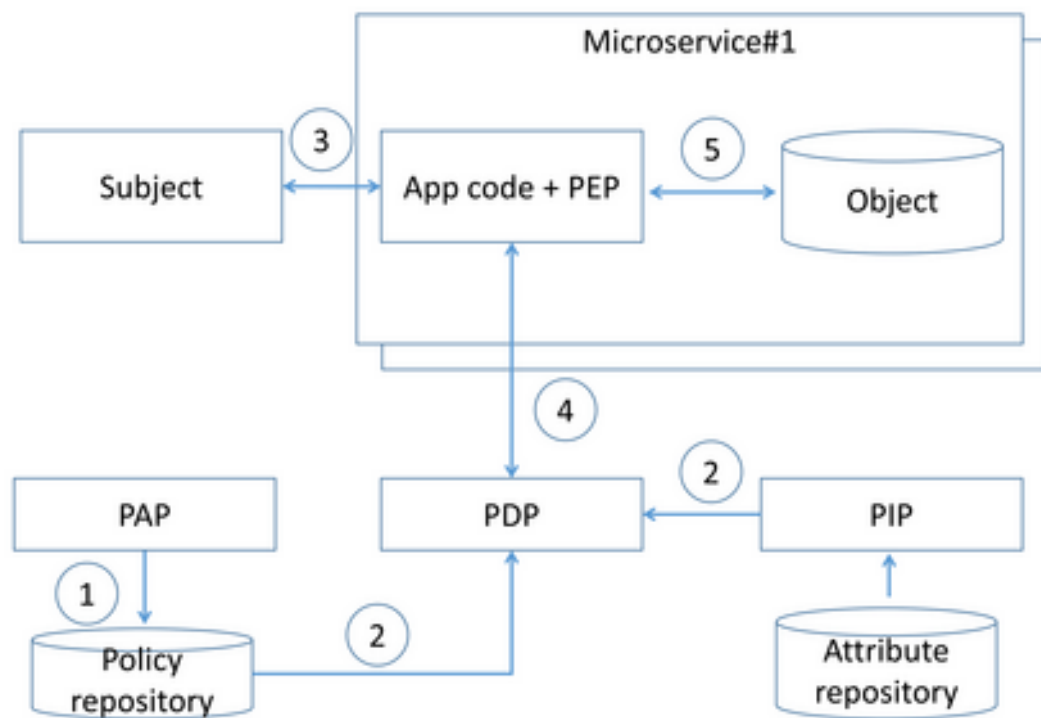


Рисунок 1.4 — Архітектура моделі з єдиною точкою ухвалення рішення

Такий підхід має ряд переваг, серед яких можна виділити наступні:

- надання можливості оновлювати правила контролю доступу команді розробників та безпеки без зміни вихідного коду, що надає можливість централізовано керувати політиками. Таким чином надається можливість вносити зміни у політики без безпосереднього впливу на мікросервіс;
- деякі правила можуть бути універсальними для усієї системи, через що вони можуть спільно використовуватися в у багатьох мікросервісах;
- команда розробників може сама реалізовувати правила контролю доступу, при чому виносячи це за межі основної частини вихідного коду. Завдяки такому підходу політики можуть бути доступні для періодичного аудиту;
- правила контролю доступу можуть використовуватися заради виявлення деяких аномалій безпеки.

1.3.5 Централізована модель із вбудованою точкою ухвалення рішення.

Ця модель передбачає централізоване визначення правил контролю доступу, при чому збереження та оцінювання виконується тільки на рівні мікросервісів. Схема цієї моделі зображено на рисунку 1.5.

У цій моделі правила контролю доступу визначаються за допомогою PAP (крок 1), після чого потрапляють у вбудовану PDP разом з необхідними атрибутами доступу, які застосовуються при використанні цих правил (крок 2). Коли користувач викликає сервіс (крок 3), викликається PDP, який генерує рішення політики контролю доступу, надаючи оцінку вхідному запиту за правилами та атрибутами контролю доступу (крок 4). І на основі цього рішення виконується авторизація (крок 5).

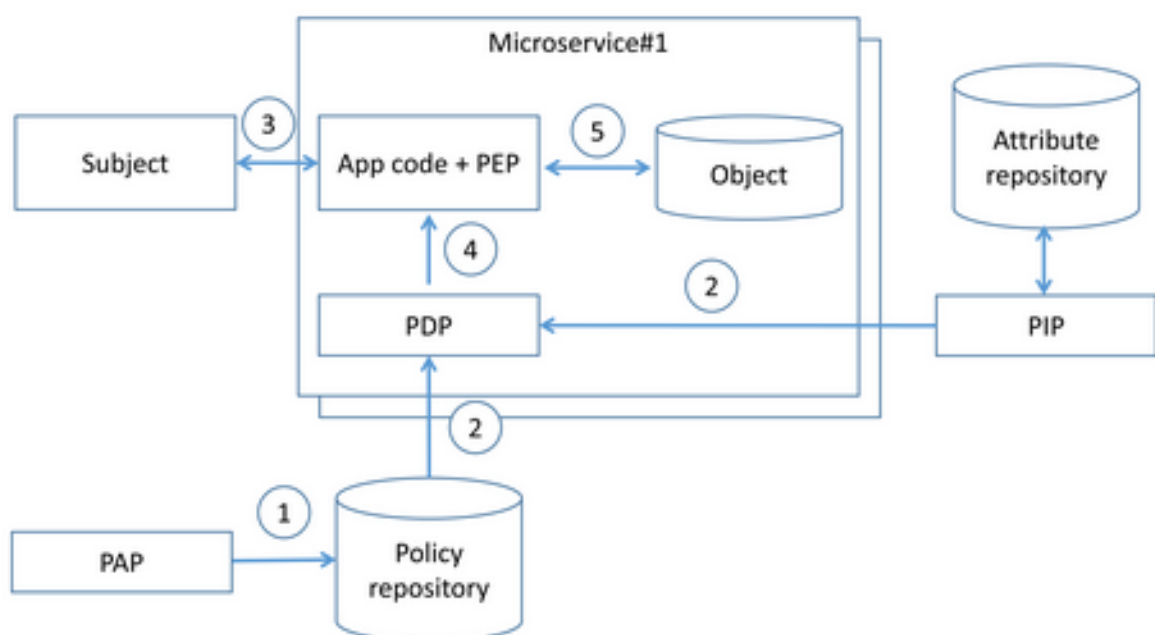


Рисунок 1.5 — Архітектура моделі із вбудованою точкою ухвалення рішення.

Через можливі неполадки та затримки при передачі даних у мережі або відмови у хоста рекомендується використовувати вбудовану PDP у вигляді мікросервісної бібліотеки на одному хості з цим мікросервісом. Вбудована PDP реалізується таким чином, щоб зменшити кількість залежностей, необхідних для авторизації та для досягнення низької затримки, що реалізується за рахунок зберігання політики авторизації та пов'язаних з цим процесом даних у пам'яті.

Ключова відмінність від "централізованого шаблону з єдиною точкою прийняття рішення" з підходом до кешування полягає у тому, що на стороні мікросервісу не зберігається рішення щодо авторизації, натомість там зберігається актуальна політика авторизації. Кешування рішень взагалі може стати причиною застосування застарілих правил авторизації та порушення правил контролю доступу.

Ця модель має ті ж переваги, що і "Централізована модель з єдиною PDP", і в додаток до цього ця модель не схильна до затримок обслуговування мережесих запитів, так як у ній використовується вбудовувана PDP на рівні мікросервісів. Але необхідно враховувати і наявність ряду проблем, пов'язаних з використанням цієї моделі:

- правила політик доступу створюються у ручному або напівручному порядку, розроблені командою безпеки, через що вони можуть містити помилки. Для уникнення вразливостей необхідно витратити більше ресурсів для проведення тестування та верифікації безпеки;
- необхідність комбінування з іншими моделями (наприклад, авторизація на прикордонному рівні) заради уникнення єдиної точки прийняття рішень та забезпечити дотримання принципу "ешелонованої оборони";
- складність з отриманням оновлень політик авторизації із централізованих PDP;
- необхідність використовувати формальної мови для опису політики контролю доступу та складність з використання сторонніх компонентів авторизації [8].

1.3.6 Висновки

Виконавши аналіз існуючих методів авторизації, було визначено їх переваги та недоліки, а також область їх застосування. Для кожного описаної моделі було розглянуто наявні недоліки та переваги, які безпосередньо впливають на процес роботи з цими моделями. У таблиці 1.1 було порівняно основні особливості усіх розглянутих моделей.

Таблиця 1.1 – Порівняльна характеристика моделей авторизації

Назва	Масштабованість	Затримка	Залежність від сторонніх компонентів	Реконфігурація на льоту
Децентралізована модель	Низька	Низька	-	-
Централізована модель з єдиною PDP	Висока	Висока	+	+
Централізована модель із вбудованою PDP	Висока	Низька	+	+

1.4 Міжсервісна аутентифікація

Серед методів міжсервісної аутентифікації існує два загальні підходи до реалізації, а саме:

- взаємна автентифікація на транспортному рівні (mTLS);
- на основі токенів, наприклад, JSON Web Tokens.

При використанні mTLS кожен мікросервіс сам визначає з ким він спілкується, чим досягається конфіденційність та цілісність даних переданих даних. На початку життєвого циклу мікросервіс отримує свою особисту пару відкритого та закритого ключів та відкритий ключ центру сертифікації, який і буде використовуватися при аутентифікації мікросервісу через mTLS. Серед недоліків використання mTLS виділяють наступні: первинне завантаження ключів, забезпечення безпечного доступу до них та оновлення ключів.

В свою чергу використання токенів відбувається вже на прикладному рівні. Токен виступає сховищем даних, у ньому може міститися різна інформація про мікросервіс, а також данні викликаючого мікросервісу. Такий викликаючий

мікросервіс отримує токен за допомогою звернення до служби видачі токенів, використовуючи власний ідентифікатор і пароль, після чого отриманий токен він використовує при кожному зверненні. Аутентифікація з використанням токенів найчастіше виконується з використанням протоколу TLS, завдяки чому забезпечується додаткова конфіденційність і цілісність даних.

1.5 Огляд мов програмування

Для розробки додатків на базі мікросервісної архітектури існує достатньо велика кількість різних мов програмування та фреймворків. У цьому підрозділі ставиться задача розглянути існуючі варіанти та визначитись з мовою, що буде використовуватися у проекті.

Вибір мови залежить від ряду факторів, що значною мірою впливають на побудову мікросервісного додатку та потребують ретельного аналізу. Серед основних критеріїв можна виділити наступні:

- популярність — базується на тому, скільки підприємств зробили цю мову чи фреймворк своїм стандартом. Також впливає на наявність якісної документації, що значно впливає на ефективність розробки;
- простота розробки — впливає на те, наскільки полегшується розробка та підвищується продуктивність роботи команди розробників. Також належить від наявності інструментів розробки, таких як IDE;
- незалежне розгортання — фреймворк повинен підтримувати усі аспекти незалежного розгортання — пряму, зворотну сумісність та багаторазове використання;
- придатність для мікросервісів — наскільки мова чи фреймворк підходить саме для створення мікросервісів;
- підтримка автоматизації — фреймворк повинен підтримувати автоматизацію завдань, пов'язаних зі збиранням та розгортанням мікросервісів.

Згідно опитуванню, проведеному компанією JetBrains, найпопулярнішими з існуючих мов програмування, що використовуються на розробки мікросервісних додатків є мови Java (41%), JavaScript (37%) і Python (25%) [10]. Діаграма зображена на рисунку 1.6.

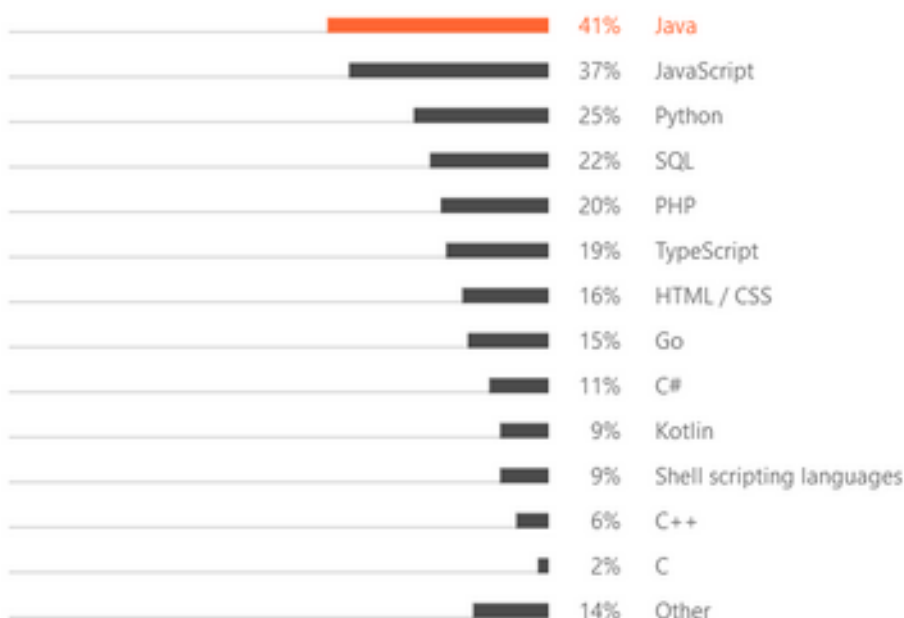


Рисунок 1.6 — Найпопулярніші мови для мікросервісів

Таким чином, необхідно розглянути саме ці мови програмування та найбільш придатні фреймворки на цих мовах.

1.5.1 Java

Мова програмування Java була створена у 1995 році і досі залишається одною з найпопулярніших мов у світі. Завдяки тривалості обслуговування та високій популярності однозначно можна говорити про те, що Java є надійним вибором для розробки нових додатків.

Завдяки використанню віртуальної машини Java (JVM) досягається платформозалежність середовища виконання, що також сприяє розповсюдженню цієї мови. Вагомою проблемою ранніх версій JVM була низька продуктивність. JVM є додатковим рівнем абстракції, яка створює додаткові витрати. Але сучасні версії JVM значно покращили продуктивність під час виконання.

Також особливістю використання є те, що JVM сумісний і з іншими мовами, такими як Kotlin, Scala або Ballerina. Використовуючи їх, команді розробників доступна і уся існуюча екосистема бібліотек та фреймворків Java

Завдяки анотаційному синтаксу значно спрощується прочитання коду, що є важливою особливістю мови Java, та є ключовим фактором, який робить її зручною для розробки мікросервісів. Java надає зручний користувацький інтерфейс, компоненти моделі, а також зручне підключення до використовуваних ресурсів, усе в межах однієї ізольованої та незалежно розгорнутої системи.

Крім того, багато стандартів Java EE добре підходять для використання у мікросервісах, наприклад:

- JAX-RS для використання API;
- JPA для обробки вхідних даних;
- CDI для впровадження залежностей і керування життєвим циклом системи.

На сьогоднішній день існує кілька фреймворків, які використовуються при розробці додатків з мікросервісною архітектурою. Серед них можна виділити наступні:

- Spring Boot — це проєкт, перед яким стоїть мета спрощення створення додатків на основі Spring. Він дозволяє спростити створення веб-додаток, вимагаючи від розробників мінімум зусиль щодо його налаштування та написання коду. Найбільш важливими особливостями цього фреймворку є: управління залежностями, автоматична конфігурація та вбудовані контейнери сервлетів;

- Spark — один із найкращих фреймворків для мікросервісів на Java, підтримує, у тому числі, створення веб-додатків і Kotlin. Spark є мінімалістичним фреймворком, що прагне простоти і забезпечує лише мінімальний набір функцій. Але при цьому він надає усі необхідні функції для створення мікросервісів, які помістяться в кілька рядків коду. Spark має досить чистий та полегшений синтаксис;

- Dropwizard — цей фреймворк використовується для розробки зручних, високопродуктивних та RESTful мікросервісів. Без додаткових налаштувань підтримує інструменти конфігурації, метрики програми, протоколювання та роботи;
- Restlet — фреймворк слідує архітектурному стилю RST, який допомагає розробникам розробляти мікросервіси. Підтримується компанією Apache Software License;
- Micronaut — full-stack фреймворк для побудови модульних, легко тестованих мікросервісних та безсерверних додатків. За його допомогою можливо створювати повнофункціональні мікросервіси, включаючи впровадження залежностей, автоконфігурацію, виявлення служб, маршрутизацію HTTP та клієнт HTTP. Перед Micronaut стоїть задача уникнення недоліків фреймворків Spring, Spring Boot, забезпечуючи швидший час запуску, зменшення обсягу пам'яті, мінімальне використання рефлексії та покращене юніт-тестування.

1.5.2 JavaScript

З моменту створення мова JavaScript пройшла значний та довгий шлях від скриптової мови для веб-сторінок до однієї з найважливіших мов у багатьох галузях розробки програмного забезпечення. На це значною мірою вплинула поява, зокрема, такої технології як Node.js, яка як раз і створюється для створення веб-додатків, у тому числі і мікросервісів.

Node.js — не є окремою мовою програмування, а виступає в якості платформи для використання JavaScript поза межами браузеру, що дозволяє створювати серверну частину веб-додатку. Node.js прекрасно реалізує парадигму JavaScript для всього. Вона передбачає, що одна мова програмування може використовуватися для повноцінної розробки веб-застосунків замість використання різних мов для роботи над фронтом та бекендом.

В якості ядра Node.js використовує Google V8. Це ядро було розроблено з метою покращення продуктивності та підвищення швидкості виконання коду JavaScript за рахунок JIT-компіляції. Саме ядро V8 написано на C++.

Однією з основних сильних сторін Node.js є використання однопотокового програмування, що реалізується завдяки трьом базовим концепціям Node.js: подіям, асинхронним API і не блокованому вводу/виводу.

Але, насправді, Node.js не є виключно платформою для виконання коду JavaScript. Також він використовується і для виконання TypeScript, який також є достатньо популярною мовою для розробки мікросервісів згідно опитуванням JetBrains (рисунок 1.6). Особливістю TypeScript є те, що це не є повноцінною та самостійною мовою програмування, це скоріше надбудова над JavaScript.

Багато проблем у JavaScript виникають через динамічну типізацію і в цілому дивну поведінку типів даних. І основною особливістю цієї мови є те, що вона додає явне статичне визначення типів до JavaScript. Також він відрізняється підтримкою використання повноцінних класів, що відповідають традиційним класам у об'єктно-орієнтованих мовах. Зрештою код TypeScript транспілюється в JavaScript та може бути виконаний у Node.js.

Існує декілька фреймворків для розробки мікросервісних додатків з використанням NodeJS:

- Express — це легкий та гнучкий фреймворк, який надає широкий набір функцій для створення веб-додатків. Однією з сильних сторін Express є його гнучкість та розширюваність. Express був розроблений з метою полегшення створення API та веб-додатків, що дозволяє значно економити час при розробці та робити веб-додатки ефективними. Також це сприяє ефективній розробці і мікросервісів;

- NestJS — передовий фреймворк, який використовує сучасні архітектурні підходи та диктує свої правила. NestJS базується на Express, тому будь-які розширення для Express, легко впровадити і сюди. У цьому фреймворку використовується прогресивний JavaScript з повною підтримкою TypeScript з поєднанням елементів об'єктно-орієнтованого, функціонального і реактивного функціонального програмування;

- Molecular — фреймворк, що створений спеціально для роботи з подієво-орієнтованою архітектурою мікросервісів. Містить вбудований реєстр

сервісів та виконує їх динамічне виявлення, здійснює балансування навантаження на запити та події, а також функцію стійкості до відмов та вбудоване кешування.

1.5.3 Python

Python — це достатньо популярна мова програмування високого рівня, яка підтримує інтеграцію з широким вибором технологій. Створення прототипів у Python є швидшою і легшою порівняно з іншими фреймворками та мовами.

Python широко використовується в веб-додатках, розробці програмного забезпечення, науці про дані та машинному навчанні. Популярність Python зумовлена тим, що ця мова є ефективною, простою у вивченні та має підтримку різних платформ. Мікросервіси на Python забезпечують широку сумісність із застарілими мовами, такими як ASP чи PHP, що дозволяє створювати фронтенд інтерфейси веб-сервісів для розроблених мікросервісів. Завдяки появі асинхронного програмування на Python з'явилися веб-фреймворки, продуктивність яких не поступається GO чи Node.js.

Існує значна кількість фреймворків для створення мікросервісів на Python. Деякі з них такі:

- Flask — мікрофреймворк для швидкої побудови додатків, у тому числі мікросервісів мовою Python. У комплекті з Flask немає ніякого набору інструментів та бібліотек, тому програміст може сам встановити їх залежно від завдань. Це робить Flask дуже гнучким та придатним для широкого переліку завдань. Flask використовує шаблонізатор Jinja2, а також Werkzeug — інструмент для роботи з WSGI — стандартом взаємодії між програмою Python, яка виконується на стороні сервера, і самим веб-сервером. Для створення ізольованого середовища у Python використовується модуль Virtualenv;

- Nameko — фреймворк був розроблений для побудови мікросервісів із вбудованою підтримкою RPC через AMQP, асинхронних подій. Також фреймворк має інтерфейси HTTP для тих, хто не може використовувати AMQP, і інтерфейс WebSocket для поєднання з Javascript;

- FastAPI — це сучасна, високопродуктивна веб-платформа для створення лаконічних та досить швидких HTTP API-серверів із вбудованими

валідацією, серіалізацією та асинхронністю. Його засновано на двох інших фреймворках: роботою з Веб займається Starlette, а за валідацію даних відповідає Pydantic.

1.5.4 Вибір мови програмування

Ключовою проблемою при розробці будь-якого програмного забезпечення є правильний вибір мови програмування та придатного фреймворку для швидкого та ефективного створення надійних програм.

Вибір фреймворку повинен ґрунтуватися на визнанні в галузі, наявності навичок у команди, активності підтримки, простоті розробки. Тому серед приведених фреймворків, придатних для побудови мікросервісних додатків, було обрано Express для JavaScript, через вище перелічені переваги. Цей фреймворк дає змогу швидко та ефективно розроблювати мікросервісні додатки, будучи впевненим у підтримці, безпечності, наявності широкої документації.

1.6 Аналіз технологій СУБД

Значущою частиною будь-якого веб-додатку є база даних. Інформація, що може зберігатися в базі даних, може бути будь-якою, наприклад дані додатків, каталог продукції, персональні дані клієнтів, хеш-сума паролів. Для доступу та маніпулювання цією інформацією застосовується система управління базами даних (СУБД).

СУБД — це сукупність програмних, мовних та інших засобів, які призначені для створення, управління, контролю, адміністрування та спільного використання БД різними користувачами. СУБД поділяють за тою моделлю, що в ній використовуються. Таким чином, СУБД, що засновані на роботі з реляційною моделлю даних, називають реляційними. Використання СУБД допомагає виконувати CRUD операції над даними, сортувати, створювати зв'язки між таблицями, і проводити інший зручний контроль за даними.

При порівнянні різних СУБД потрібно враховувати те, наскільки зручною вона буде для кінцевого користувача, наскільки вона придатна до масштабування, а також необхідно переконатися, що вона без проблем може бути інтегрована у мікросервісний додаток. При цьому мікросервісна архітектура сприяє децентралізації методів зберігання даних, що безпосередньо впливає методи зберігання та на використовувані СУБД.

У мікросервісних додатках кожному сервісу надають окрему базу даних, як створюючи окремі екземпляри загальної СУБД, так і використовуючи нестандартні бази. Такий підхід називається Polyglot Persistence. Він може застосовуватися і в монолітних програмах, але в мікросервісах такий підхід зустрічається частіше. На рисунку 1.6 зображено такий підхід у випадках з мікросервісними додатками.



Рисунок 1.6 — Використання різних СУБД для окремих сервісів [9]

Такий підхід у результаті дозволяє використовувати найбільш придатну модель СУБД для конкретного сервісу, що значно впливає на ефективність роботи системи в цілому.

1.6.1 Вибір СУБД

Для виконання поставленої задачі в якості СУБД буде використовуватись SQL база даних. Такий вибір обумовлений поширеністю таких систем, тим самим дослідження буде корисним для більшого числа людей. Так як вибір СУБД не є принциповим для основної мети роботи, вибір буде зроблений в сторону PostgreSQL.

PostgreSQL є не просто реляційною, а об'єктно-реляційною СУБД, що надає йому деякі переваги над іншими базами даних SQL з відкритим вихідним кодом, такими як MySQL, MariaDB і Firebird. Окрім іншого це дозволяє зберігати в ній, наприклад, json-об'єкти і проводити пошук за значеннями його ключів. Всі таблиці у базі представлені у вигляді об'єктів, які можуть успадковуватися, а вся взаємодія з таблицями виконується за допомогою об'єктно-орієнтованих функцій. Окрім типів даних, визначених стандартом, PostgreSQL підтримує ще і uuid, грошовий, перерахований, геометричний, бінарний тип, мережеві адреса, бітові строки, xml, json, та навіть масиви і композитні типи.

PostgreSQL має підтримку використання курсорів для переміщення по отриманим даним. Користувач отримує лише покажчик, при чому відповідь зберігається в пам'яті сервера баз даних. Це дозволяє зберігати покажчик між різними сеансами. Також ця СУБД підтримує побудову індексів відразу для декількох стовпців таблиці. Крім того, індекси можуть бути різних типів, крім hash і b-tree доступні GiST і SP-GiST для роботи з містами, GIN для пошуку по тексту, BRIN і Bloom. PostgreSQL підтримує регулярні вирази в запитах, рекурсивних запитів і успадкування таблиць.

1.6.2 Модель бази даних

Для роботи з СУБД PostgreSQL у фреймворку Express передбачено декілька можливих рішень:

- ORM Sequelize;
- колекція Node.js модулів node-postgres;
- бібліотека pg-promise, розроблена на основі node-postgres.

Node-postgres, як і pg-promise, підтримує колбеки, проміси, async/await, об'єднання з'єднань, підготовлені оператори, курсори, потокові результати, розширений аналіз типів. Проте, серед усіх представлених інструментів лише Sequelize надає змогу працювати з даними не лише за допомогою стандартних SQL запитів, але й за допомогою класів, що дозволяє суттєво спростити реалізацію моделі бази даних та підвищити ефективність роботи.

Об'єктно-реляційне відображення або ORM – це технологія програмування, яка дозволяє працювати з базою даних за допомогою об'єктно-орієнтованої парадигми. ORM використовується для представлення даних у вигляді об'єктів та спрощення збереження цих об'єктів в реляційну базу даних. При цьому ORM сама піклується про перетворення даних між двома цими станами. Принцип їх роботи ґрунтується на метаданих бази даних і об'єктів, таким чином об'єктам не потрібно нічого знати про структуру бази даних, а базі даних – нічого про те, як дані організовані у коді. ORM забезпечує можливість працювати з даними для бази і моделі незалежно у своїй вихідній формі.

Sequelize – це універсальна ORM, яка була розроблена для роботи з такими СУБД як PostgreSQL, MySQL, MariaDB, SQLite та MSSQL, що зіставляє таблиці бази даних та відносини між ними з класами. Sequelize має надійну підтримку транзакцій, відносини, активне та відкладене завантаження, реплікацію читання та багато іншого.

1.7 Висновки аналізу інструментів і технологій

В ході проведеного аналізу було визначено актуальність поточного дослідження, визначена необхідність у проведенні дослідження з вдосконалення наявних методів аутентифікації користувача. Було розглянуто питання забезпечення безпеки з рахунок розглядання питань реалізації моделей контролю доступом і аутентифікації користувачів.

Було розглянуто питання вибору мов програмування, проведено порівняльний аналіз, в результаті чого було визначено найпопулярніші та найпридатніші мови та фреймворки, за допомогою яких можна реалізовувати високоефективні мікропроцесорні системи. Також було розглянуто питання вибору бази даних та методи їх реалізації в умовах мікросервісів.

2 ПРОЄКТУВАННЯ АЛГОРИТМІВ СИСТЕМИ

Наступним етапом роботи є огляд існуючих алгоритмів аутентифікації користувачів у мікросервісних додатках, визначення їх особливостей та способів покращення, а також пошук оптимальних способів вирішення поставлених задач.

Таким чином, метою розділу є проведення проєктування власних покращених алгоритмів з використанням розглянутих у попередньому розділі програмних засобів і архітектурних підходів.

2.1 Огляд існуючих підходів реалізації аутентифікації

На даний момент існує ряд технічних підходів до реалізації аутентифікації у мікросервісних системах. Розглянемо основні найпоширеніші підходи.

У більшості випадків веб-додатки заради проведення аутентифікації вимагають від користувача його облікових даних — імені користувача та пароля. Після отримання ці дані перевіряються за допомогою наступних підходів:

- локально у самому додатку або за допомогою пристрою, на якому було встановлено програму;
- облікові дані надсилаються на зовнішній довірений сервер для подальшої перевірки.

При цьому сервер може мати ще один додатковий рівень перевірки, а саме — багатофакторну аутентифікацію (MFA), що реалізується за допомогою надсилання повідомлення на пошту чи телефон користувача з додатковим кодом безпеки.

2.1.1 Локальна аутентифікація

Найпростішим можливим методом аутентифікації є використання на етапі розробки хардкоду комбінації імені та паролю для стандартних користувачів. Такий підхід дозволяє вбудованим у додаток користувачам виконувати наступні захищені дії, наприклад, додати більше користувачів до системи. Хоча такий підхід і є найпростішим, в свою чергу він є достатньо вразливим до атак і в загалом є найменш безпечним. Будь-яка заздалегідь визначена комбінація імені користувача та паролю є небезпечним бекдором у систему. Крім того, по мірі росту додатку, список таких користувачів може швидко застаріти, оскільки буде відбуватися постійна зміна користувачів системи. Наявність застарілих облікових записів створює додаткові ризики для безпеки системи, оскільки їх можна використовувати для несанкціонованого доступу до неї.

2.1.2 Змінні модулі розпізнавання (РАМ)

РАМ — це модулі зі спільним API інтерфейсом, що забезпечують підтримку динамічної аутентифікації для програм і служб. РАМ можна використовувати з метою керування локальними користувачами на віртуальній машині, на якій розміщено додаток [11]. При цьому додаток може використовувати РАМ API для аутентифікації користувачів. Безпосередньо користувачі РАМ керуються центральним користувачем, а також зовнішніми політиками за допомогою Ansible або іншим інструментом керування конфігураціями. Під час того, як додаток звертається до API РАМ, відбувається завантаження модулю аутентифікації, як визначено у конфігураційному файлі `ram.conf`. Далі запит пересилається до основного модуля аутентифікації для виконання вказаної операції. Після чого РАМ повертає відповідь від модуля аутентифікації до програми.

2.1.3 HTTP аутентифікація

Базова HTTP аутентифікація була визначена у стандарті RFC 7617. Вона передбачає передачу облікових даних користувача як пари ідентифікатора користувача і паролю, які закодовані за допомогою base64. Ця схема аутентифікації вбудована в сам протокол HTTP. Оскільки облікові дані передаються у відкритому вигляді, цей метод є небезпечним і має поєднуватися з шифруванням на транспортному рівні.

На рисунку 2.1 показано зображено процес проходження аутентифікації за допомогою базової HTTP аутентифікації.

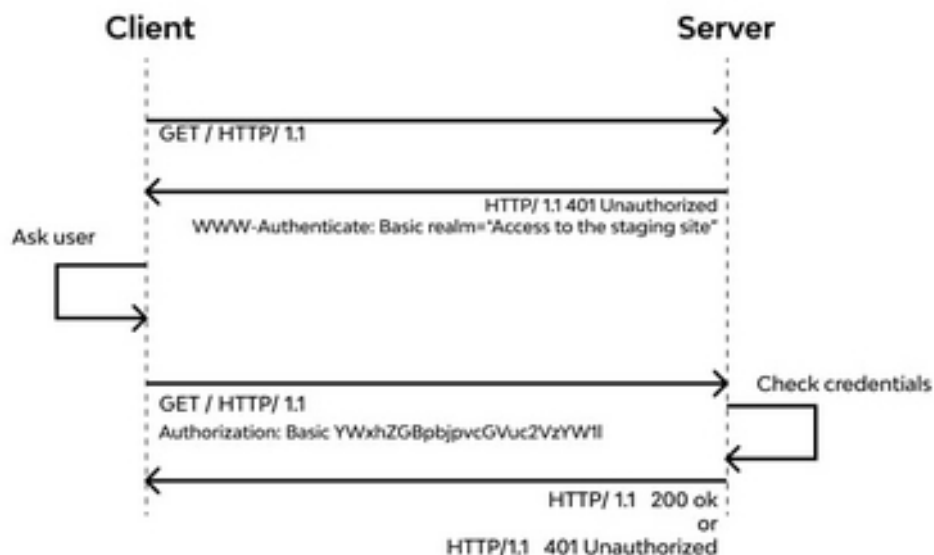


Рисунок 2.1 — Алгоритм базової HTTP аутентифікації

Якщо при першому зверненні до захищеного ресурсу у користувача не має доступу, тоді буде видана помилка 401 Unauthorized. Також відповідь містить тип аутентифікації та контекст, в рамках якого відбувається аутентифікація. Користувач передає HTTP-запит із заголовком авторизації, в якому міститься слово "Basic", після чого іде пара "ім'я користувача: пароль" у закодованому виді, на сервері ці дані перевіряється. Підхід має ряд небезпек. Найпоширеніша - атаки посередника, під час якої при використанні незахищеного з'єднання облікові дані

можуть перехопити зловмисники у момент передачі від клієнта до сервера чи назад.

Розвитком цього підходу стала технологія під назвою дайджест HTTP аутентифікації, яка змінює підхід до роботи з обліковими даними користувача. Для їх перевірки тепер використовується MD5-хеш з сіллю, що унеможлиблює підбір логіну чи паролю користувача. Хоча він і покращує старій алгоритм, вже ж таки залишається вразливим до багатьох атак.

У кінцевому рахунку технологія розвинулась у протокол Forms authentication, де процес аутентифікації відбувається на більш високому рівні моделі абстракції. Відмінністю є те, що HTTP-сервер вже не повідомляє про помилку доступу, замість чого перенаправляє користувача на іншу сторінку, зазвичай на сторінку авторизації. На ній користувач вносить свій логін та пароль, які відправляються у POST-запиті через захищений канал на сервер. Сервер, в свою чергу, повертає токен або ідентифікатор сесії, який і використовується для подальшого доступу до захищених ресурсів.

2.1.4 Розподілене керування сеансами

Існує декілька способів керування сеансами користувачів, як, наприклад, збереження сеансів на локальному вузлі, який безпосередньо відповідає на HTTP-запит, або визначення окремого рівня в розроблюваній архітектурі, так щоб вона змогла зберігати ці сеанси масштабованим і надійним способом. Тому для забезпечення цих переваг і досягнення масштабованості та відмовостійкості мікросервісів, підхід з розподіленим керуванням сеансами є найбільш придатним.

Цього можна досягнути користуючись різними способами, наприклад:

- Закріплена сесія надає гарантію того, що усі запити певного користувача будь надсилатися до того ж серверу, до якого було надіслано і перший, таким чином гарантуючи дані сеансу завжди правильні для цього користувача. Але такий підхід залежить від балансувальника навантаження, і він відповідає лише випадкам з горизонтально розширеними кластерами, і тому у випадках, коли балансувальник вимушений перевести користувача на інший сервер, усі його дані сеансу будуть втрачені.

- Реплікація сесії означає, що кожен екземпляр зберігає усі дані сеансу самостійно та виконує синхронізацію через мережу. Така синхронізація робить значне навантаження на пропускну здатність мережі. Тому поки дані сеансу підвертаються змінам, їх необхідно постійно синхронізувати з усіма вузлами.
- Централізоване зберігання сесій означає, що користувацькі дані можна отримати зі спільного для усіх мікросервісів сховища сесій, що гарантує те, що усі вони читають ті ж самі дані. Це достатньо ефективне та масштабоване рішення, але його недоліком є те, що спільне сховище потребує деякого механізму захисту.

2.1.5 Аутентифікація за допомогою токenu

Традиційним способом реалізації аутентифікації є використання сесії на стороні сервера за допомогою якого виконується збереження стану користувача. Оскільки у сервера є свій стан, це значним чином впливає на його горизонтальне розширення. Тому рекомендується використовувати токен для запису статусу користувацької авторизації.

Головною відмінністю токenu від сесії полягає в особливості організації сховища. Сесії зберігаються централізовано безпосередньо на сервері, в свою чергу токени — у самих користувачів, зазвичай у куках браузера. Токен містить інформацію про користувача, його систему, браузер, тому кожного разу, коли користувач виконує запит на сервер, сервер може ідентифікувати користувача та визначити, чи має він доступ до запитуваних ресурсів. Так як токен використовується для ідентифікації користувача, вміст його повинен бути зашифрованим для уникнення його фальсифікації.

JSON Web Token (JWT) – це JSON об'єкт, який був визначений у відкритому стандарті RFC 7519. Він вважається одним із найбезпечніших способів передачі інформації між двома учасниками. JWT складається з трьох основних частин: заголовка (header), корисне навантаження (payload) та підпису (signature). Заголовок та навантаження формуються окремо, а потім на їх основі обчислюється підпис [4].

Розглянемо структуру JWT. Заголовок, зазвичай, складається з двох полів: типу токена (в даному випадку JWT) та алгоритму хешування підпису:

Лістинг 2.1 – Заголовок JWT

```
{  
  "typ": "JWT",  
  "alg": "HS256"  
}
```

Згідно офіційного сайту jwt.io для алгоритму хешування рекомендується використовувати два наступні методи: HS256 та RS256. Але насправді можна використовувати будь-який алгоритм із приватним ключем.

Корисне навантаження – це будь-які дані, які використовуються для формування токена. Але стандарт передбачає кілька зарезервованих полів:

- iss - (issuer) видавець токена;
- sub - (subject) призначення токена;
- aud - (audience) одержувачі токена;
- exp - (expire time) термін дії токена;
- nbf - (not before) термін, до якого токен не дійсний;
- iat - (issued at) час створення токена;
- jti - (JWT id) ідентифікатор токена [12].

Payload не шифрується під час використання токена, тому не можна передавати в ньому дані, які не повинні потрапити у відкритий доступ.

Підпис формується на основі заголовка та корисного навантаження. Таким чином, якщо хтось спробує змінити дані у токені, він не зможе змінити підпис, не знаючи приватного ключа.

В мікросервісних системах зазвичай аутентифікацію організують за допомогою API-шлюзу. Це означає, що всі запити проходять лише через шлюз, приховуючись від мікросервісу. У шлюзі оригінальний токен користувача перетворюється в непрозорий токен, який може використовувати лише він сам.

Процес аутентифікації зображено на рисунку 2.2.

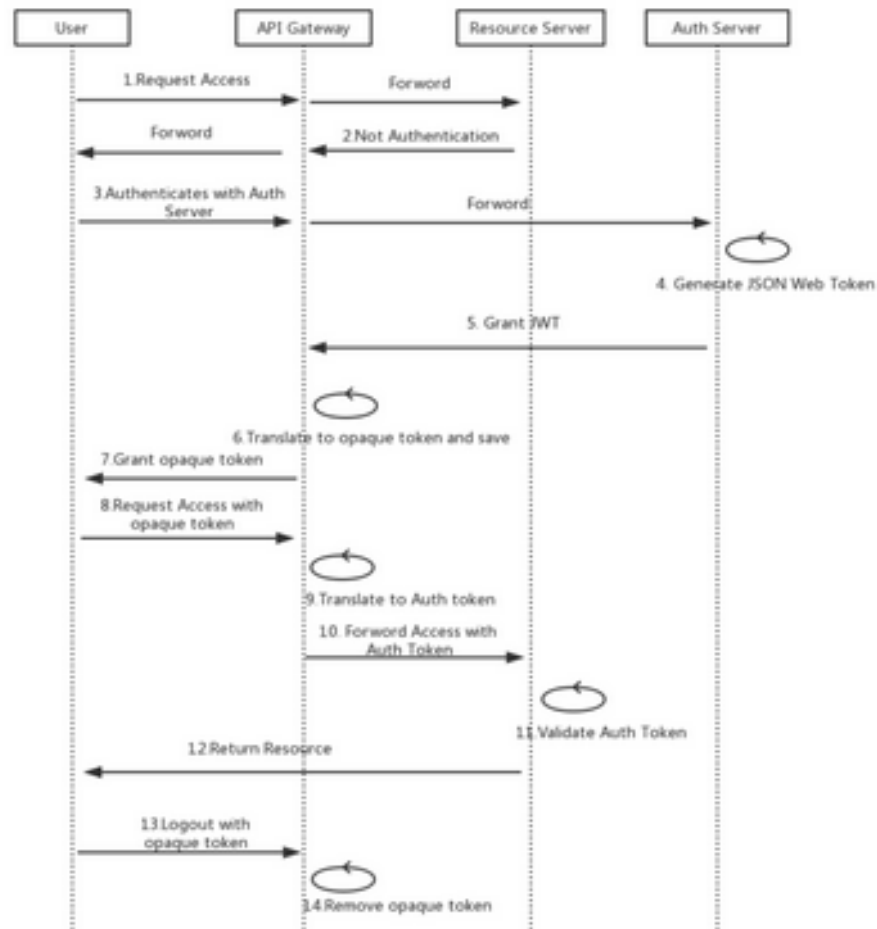


Рисунок 2.2 — Алгоритм аутентифікації за допомогою токена

2.1.6 Технологія єдиного входу (SSO)

Ідея SSO полягає у тому, що користувачу необхідно лише один раз увійти в систему, після чого він отримує доступ одразу до кількох систем. При використанні цього підходу сервіс, що виконує запит, передає права видачі необхідних для доступу токенів власне токен-провайдеру. Існує два найпопулярніші протоколи, що підтримують SSO: OpenID Connect (OIDC) і Security Assertion Markup Language (SAML) [12].

На рисунку 2.3 зображено процес проходження аутентифікації за допомогою технології єдиного входу.

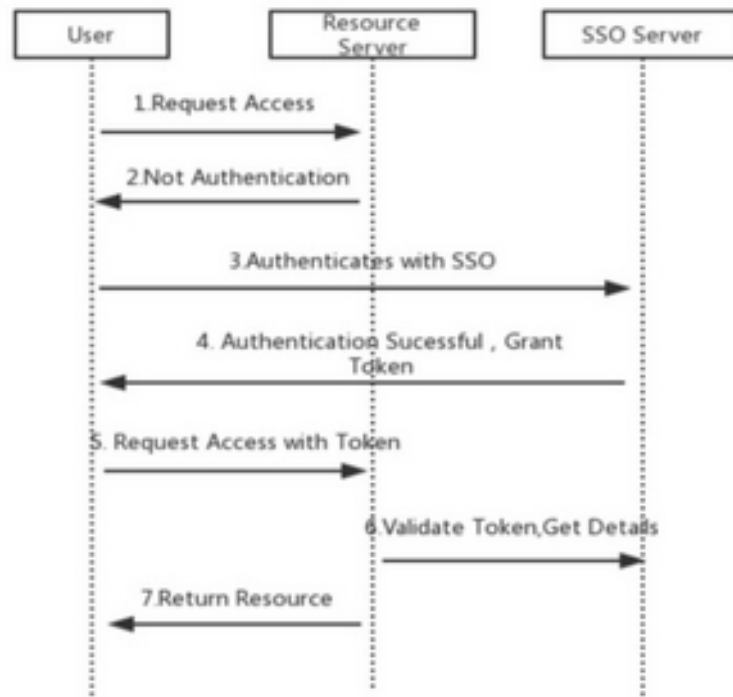


Рисунок 2.3 — Алгоритм аутентифікації на основі SSO

У контексті архітектури мікросервісів SSO може бути реалізованим у двох варіантах:

- аутентифікація кінцевих користувачів – для кінцевого користувача зручніше виконувати один вхід замість того, щоб кожного разу виконувати нову аутентифікацію. У такій ситуації можна використовувати рішення для керування ідентифікацією та доступом (IAM), за допомогою якої налаштувати базу даних користувачів та визначити їх дозволи для доступу в окремі сервіси. Мікросервіси перенаправляють користувачів до системи IAM під час аутентифікації, отримуючи SSO-токен, який і використовується під час входу користувачем;
- аутентифікація мікросервісів – використання SSO-токену для мікросервісів, що потребують підключення до інших мікросервісів або зовнішніх сервісів, які отримують доступ через API. У цьому випадку необхідно провести авторизацію служби або сервісу, а не користувачів. У цьому випадку також використовується IAM.

2.2 Огляд існуючих підходів реалізації авторизації

Основним завданням авторизації є розмежування доступу до цільового об'єкта на підставі даних про суб'єкт, що запитує доступ, та властивості цього об'єкта. Доступ визначається політиками та правилами, що розміщені у програмі на основі визначених вимог. Перевірка авторизації, як правило, виконуються після успішної аутентифікації користувача. Розглянемо існуючі схеми авторизації.

2.2.1 Контроль доступу на основі ролей (RBAC)

RBAC є найпопулярнішим підходом до обмеження доступу. Перевагою цього підходу є те, що замість індивідуального надання чи скасування доступу, користувачі об'єднуються у групи за ролями, яким і видаються права. При цьому RBAC має ряд обмежень, наприклад, не можливо налаштувати правило, використовуючи параметри, невідомі системі до того, як користувач почне працювати [13]. Дозволи можна призначати лише ролям користувачів, а не об'єктам і операціям. А також за допомогою моделі RBAC можливо тільки обмежити доступ до певних дій у системі, але не до певних даних.

RBAC надає структуру авторизації з наступними критеріями:

- дозволи — операції роботи з даними, такі як read-only, read-write тощо;
- ролі — набір дозволів, які застосовуються до користувачів;
- користувачі — аутентифікований користувач, який має одну або кілька ролей. При відсутності ролей, додаток забороняє доступ користувачеві.

2.2.2 Контроль доступу на основі атрибутів (ABAC)

Ключовою перевагою ABAC над RBAC є те, що він надає доступ на основі атрибутів кожного компонента системи замість ролей. Для виконання авторизації значення всіх атрибутів беруться у момент перевірки прав і порівнюються з очікуваними значеннями. Виконання всіх умов забезпечує доступ до ресурсу. Завдяки використанню цієї моделі можливо описати правило будь-якої

складності. Крім того, правила ABAC можуть оцінювати атрибути суб'єктів і ресурсів, які ще мають бути інвентаризовані системою авторизації. ABAC має і свої обмеження, цей тип системи важко налаштувати через спосіб визначення та підтримки політик. Важко провести попередній аудит і визначити дозволи, доступні для конкретного користувача. Може бути неможливо визначити рівень ризику для конкретного користувача.

2.3 OAuth 2.0 і OpenID Connect

OAuth 2.0 — це протокол безпеки, який дозволяє одній програмі отримати дозвіл на доступ до даних користувача з іншого сайту. Послідовність дій для надання дозволу називають делегованою авторизацією. За допомогою цього стандарту надається дозвіл сервісу отримати дані або використовувати функції іншої програми від імені користувача, не використовуючи пароль. OAuth 2.0 визначає, як має реалізовуватися взаємодія між сервісами задля забезпечення безпечної авторизації [14].

OAuth 2.0 заснований на використанні базових веб-технологій: HTTP-запитах, редіректах тощо. Тому використовувати OAuth можливо на будь-якій платформі з доступом до інтернету та браузеру: на сайтах, у мобільних та desktop-додатках, плагінах для браузерів. Безпека OAuth 2.0 ґрунтується на протоколі SSL. Це полегшує процес розробки, але при цьому потребує додаткових обчислювальних ресурсів та адміністрування.

За результатом авторизації користувач отримує Access Token, який використовується для доступу до захищених ресурсів. Звернення до них відбувається за допомогою HTTPS із зазначенням ключа у заголовках запиту.

У протоколі описано кілька варіантів авторизації, які підходять для різних ситуацій:

- авторизація для додатків з серверною частиною — сайтів та веб-додатків;
- авторизація для клієнтських додатків — мобільних програм та desktop-додатків;
- авторизація за логіном та паролем;
- відновлення попередньої авторизації.

Стандарт визначає наступні ролі:

- Resource Owner – користувач, який заходить у додаток і надає йому дозвіл використовувати свої персональні дані з іншого сайту;
- Client — додаток або веб-сайт, який використовується користувачем і який взаємодіє з Authorization Server та Resource Server для отримання закритих даних користувача;
- Authorization Server - сервер який перевіряє логін/пароль користувача;
- Resource Server — зберігає закриту інформацію користувача, яку можна отримати за допомогою API. Authorization Server та Resource Server можуть бути поєднані в одну систему.

Загальний алгоритм роботи зображено на рисунку 2.4.

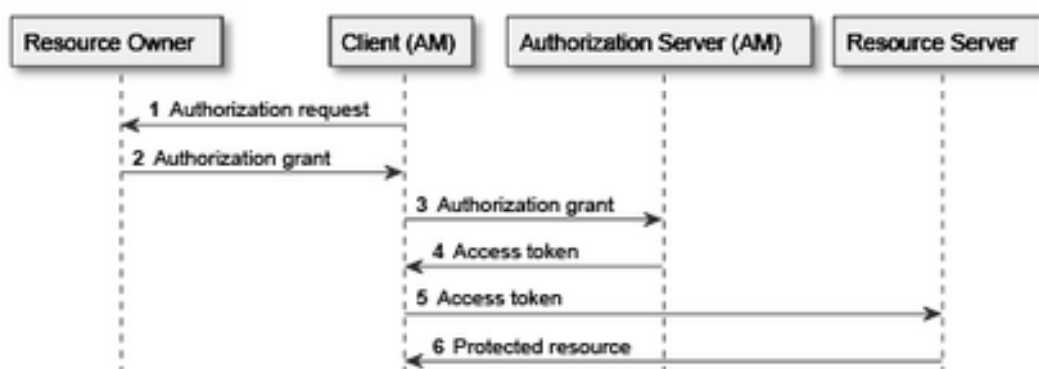


Рисунок 2.4 — Алгоритм роботи OAuth 2.0

За допомогою OAuth 2.0 відбувається лише авторизація користувача. Про нього відомо лише про його Access Token, за допомогою якого можна отримати певні дані з серверу ресурсів. Але він нічого не говорить про особу користувача, що не дозволяє надати йому доступ вже на головному веб-додатку. Тому

необхідний OpenID Connect — він додає інформацію про особу користувача (identity), що дозволяє реалізувати його аутентифікацію.

OpenID Connect також додає можливість динамічної реєстрації та виявлення сервісів. Це дозволяє будувати SSO системи (Single Sign-On), у яких використовується один логін для багатьох не пов'язаних між собою сервісів.

OpenID Connect (OIDC) — відкритий стандарт OpenID, який розширює OAuth 2.0 такими основними можливостями:

- у первинному запиті додається додатковий атрибут `scope=openid`;
- в результаті роботи алгоритму Client отримує ID Token, окрім Access та Refresh Token.

2.4 Огляд існуючих готових рішень

На сьогоднішній день існує достатня кількість готових варіантів реалізації аутентифікації та авторизації для систем на базі мікросервісної архітектури. Деякі з таких рішень виконують конкретну задачу, таку як контроль доступу, або аутентифікація, а деякі виступають в якості загальної платформи, за допомогою якої можна розгортати будь-які необхідні компоненти. Отже, розглянемо найпопулярніші рішення.

2.4.1 Kong API Gateway

Kong API Gateway є одним з найпоширеніших API-шлюзів. Він є платформонезалежним інструментом, який надає готові рішення на будь-яких платформах. Серед основних його компонентів можна виділити модулі балансування трафіку, вбудовані secure breaker, healthcheck, активний та пасивний моніторинг upstream, різні механізми аутентифікації та авторизації користувача. А також є підтримка користувацьких модулів, завдяки тому що ця платформа має відкритий вихідний код [15].

Сам Kong написаний на мові Lua і будується на основі Nginx модулю lua-nginx-module. Він може виступати як в якості безпосередньо API-шлюзу, так і API-middleware, чи навіть в деяких випадках в якості Service Mesh.

На рисунку 2.5 зображено механізм роботи Kong API Gateway.

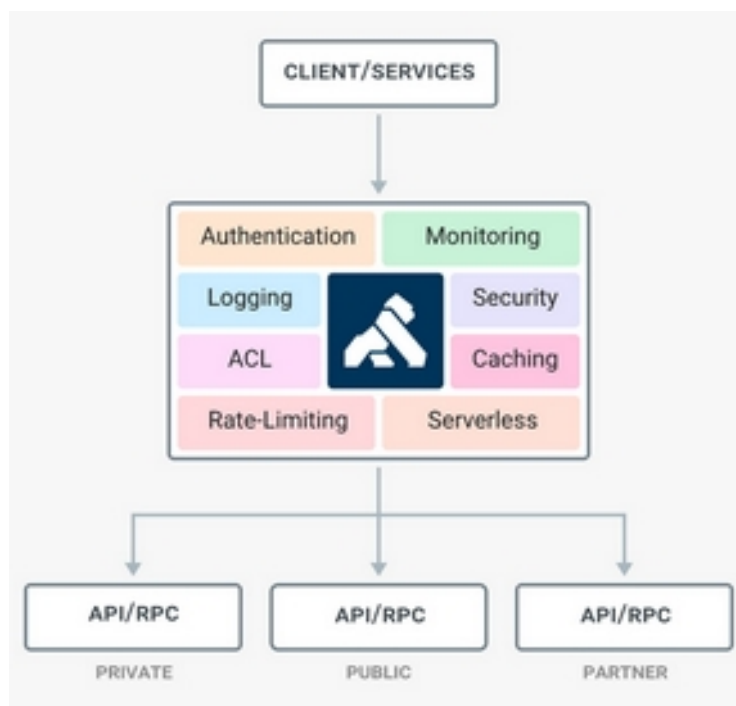


Рисунок 2.5 — Архітектура API-шлюзу Kong

Коли Kong запущено, кожен запит, що надсилається до додатку, спочатку потраплятиме до Kong, після чого буде надсилатися проксі-сервером до остаточного API. У проміжках між запитам та відповідями Kong виконає будь-який встановлений плагін, розширюючи базовий функціонал додатку.

Kong надає вибір із багатьох можливих компонентів для реалізації необхідного функціоналу у своєму мікросервісному додатку, як наприклад:

- серед методів аутентифікації можна вибрати Базову аутентифікацію, JWT, LDAP, OAuth 2.0;
- плагіни безпеки, які дозволяють створити додатковий рівень безпеки: ACL, CORS, динамічний SSL, обмеження IP;
- плагіни реалізації контролю доступу на базі ролей та атрибутів.

2.4.2 Amazon API Gateway

Amazon API Gateway — це служба Amazon Web Services (AWS) для створення, публікації, підтримки, моніторингу та захисту RESTfull API, HTTP API і WebSocket API. Amazon API Gateway є хмарним API-шлюзом, що робить його повноцінним незалежним сервісом з широкими можливостями для масштабування та інтеграції з іншими хмарними сервісами.



Рисунок 2.6 — Архітектура API-шлюзу Amazon [16]

На рисунку 2.6 показано, як за допомогою Amazon API Gateway надається інтегрований і функціональний інструментарій для створення безсерверних додатків AWS. API-шлюз виконує всі завдання, пов'язані з прийняттям і обробкою API-викликів, серед яких можна виділити керування трафіком, авторизацію та контроль доступу, моніторинг і керування версіями API.

Шлюз API є точкою входу для програм, які отримують доступ до даних, бізнес-логіки або функцій із серверних служб, таких як робочі навантаження, що виконуються в Amazon Elastic Compute Cloud (Amazon EC2), код, що виконується в AWS Lambda, будь-які веб-програми або комунікаційні програми в режимі реального часу.

Серед особливостей можна виділити наступні:

- підтримка API із збереженням стану (WebSocket) і без збереження стану (HTTP і REST);
- створення API RESTful за допомогою API HTTP або API REST;
- інтерфейси API WebSocket для розробки додатків, які потребують двостороннього зв'язку в режимі реального часу;
- приватна інтеграція з AWS ELB та AWS Cloud Map;
- API-ключі для сторонніх розробників;
- генерування клієнтських SDK багатьма мовами, включаючи JavaScript, iOS та Android;
- авторизація за допомогою AWS Lambda.

2.4.3 Tyk API Gateway

Tyk — це корпоративний API-шлюз із відкритим вихідним кодом, який підтримує такі протоколи як REST, GraphQL, TCP і gRPC. Tyk побудовано на мові Golang, яка розроблена для високої пропускну здатності та паралелізму. На відміну від Lua у Kong, чи NGINX, Golang є більш сучасною і є легшою у програмуванні. Окрім Golang, Tyk має інтерпретатори для запуску плагінів іншими мовами, такими як Javascript і Lua.

Цей шлюз розроблено з нуля, без залежностей від яких-небудь проксі-серверів. Для розподіленого обмеження швидкості та зберігання токенів використовується база даних Redis у Tyk.

Серед особливостей можна виділити наступне:

- Tyk пропонує два варіанти розгортання — як хмарне рішення SaaS, так і локальне розгортання;
- використання протоколів на вибір: REST, SOAP, GraphQL, gRPC і TCP;
- аутентифікація: OIDC, JWT, Bearer Tokens, Базова аутентифікація, сертифікати клієнта тощо;
- розширювана архітектура за допомогою проміжних плагінів Tyk, або власних плагінів, розроблюваних будь-якою мовою, яка підтримує gRPC;

- обмеження швидкості та квоти: можливість захистити вихідні канали від перевантаження та застосування обмежень для кожного окремого споживача;
- широкі налаштування контролю доступу до окремих вузлів з використанням чорного чи білий списків, ігнорування доступу;
- IP AllowListing — блокування доступу до ненадійних IP-адрес для більш безпечної взаємодії;
- безперервне перезавантаження — конфігурації Tук можна динамічно змінювати, а службу перезавантажувати, не впливаючи на активні запити
- власний декларативний API Kubernetes — використання оператора Tук з відкритим вихідним кодом.

2.5 Проектування розроблюваних алгоритмів

Здійснивши аналіз існуючих підходів та методів реалізації алгоритмів аутентифікації і контролю доступом, а також розглянувши актуальні рішення, було виявлено і визначено декілька основних шаблонів проектування. Серед них базовим підходом є використання API-шлюзу як єдиної вхідної точки розроблюваного веб-додатку. Цей підхід є ключовим при проектуванні архітектури усього проєкту, і серйозно впливає на алгоритми аутентифікації і авторизації. Також важливою особливістю є використання токенів як основного методу спілкування між користувачем з веб-додатком, а також у середині системи. Інші підходи на сьогоднішній день є застарілими та використовуються лише при умові підтримки старих проєктів, тому токени також важливо урахувати.

Отже, надалі необхідно розробити алгоритми роботи API-шлюзу, а також системи аутентифікації та авторизації користувача з урахуванням визначених умов та загальноприйнятих норм.

2.5.1 API-шлюз

Загалом перед API-шлюзом як правило ставлять багато задач по управлінню роботою сервісу, кешуванню, виконанню балансування навантаження з метою оптимізації використання ресурсів тощо. Але при проєктуванні саме цієї системи стоїть завдання лише розробити алгоритми, що напряду пов'язані з користувацькою аутентифікацією. Таким чином основними функціональними компонентами такого API-шлюзу повинні бути системи ідентифікації, аутентифікації і авторизації користувача.

Виходячи з зазначених вище обмежень необхідно визначити основні етапи, через які повинен пройти запит у API-шлюзі та як отриманий запит повинен бути оброблений. Отже процес роботи API-шлюзі виглядає наступним чином:

- 1) Користувач повинен відправляти запит в уніфікованій, єдиній формі. І, як правило, API-шлюзи для реалізації доступу до мікросервісів повинні використовувати принципи маршрутизації з повною підтримкою REST. В таких умовах, коли користувач робить запит, він буде потрапляти на один з маршрутів, який налаштований таким чином, щоб оброблятися проміжними компонентами системи;
- 2) Далі у шлюзі повинен бути реалізований функціонал, який розділятиме запити, що потребують захищеного доступу та ті, до яких можна звертатися без додаткової верифікації;
- 3) В наступному етапі необхідно проводити попередню ідентифікацію користувача, виконуючи перевірку чи насправді він існує. Переконавшись, що дійсно облікові дані цього користувача наявні у базі даних, необхідно провести його аутентифікацію, на меті якої стоїть підтвердження його особи;
- 4) При успішному проходженні попереднього етапу користувача належить перевірити на наявність у нього дозволу на доступ до захищеного ресурсу;
- 5) Далі якщо користувачу необхідно передати які-небудь дані, тоді наступним етапом буде їх обробка та внесення до запиту безпосередньо на мікросервіс. Існує декілька підходів, за допомогою яких реалізують передачу

даних всередині мікросервісної системи, але для підвищення безпеки та надійності при використанні системи для внутрішнього спілкування між API-шлюзом і мікросервісами, а також між мікросервісами безпосередньо буде використовуватись підхід з шифруванням даних. На справді існує два основних способи реалізації цього підходу, кожен із яких значно впливає на побудову архітектури усієї системи, а саме API-ключі або JWT. API-ключі є простими й повністю прозорими. Вони не зберігають ніякої важливої інформації і не шифрують свій зміст. Такий ключ являє собою нечитабельний унікальний ідентифікатор. І хоча може здатися, що використання таких ключів є значно простішим і ефективнішим рішенням, але проблеми з'являються при масштабуванні системи. В свою чергу JWT, хоча й потребує більшої обчислювальної потужності заради свого створення та верифікації, є більш ґручким інструментом, який чітко ідентифікує як користувача, так і інший сервіс, чого не може зробити API-ключ. Тому для внутрішнього спілкування в середині системи будуть використовуватись саме JWT;

6) Після того як на базі отриманих від користувача даних було створено внутрішній токен, його необхідно внести у заголовок запиту для подальшого використання у кінцевому мікросервісі;

7) Фінальним етапом роботи API-шлюзу є безпосередньо переадресація на сам запитуваний мікросервіс.

На рисунку 2.7 зображено весь описаний життєвий цикл спроектованого API-шлюзу, зображеного у вигляді алгоритму.



Рисунок 2.7 — Алгоритм роботи API-шлюзу

2.5.2 Сервіс керування сесіями

Одну з ключових ролей в системі відіграє сервіс, який умовно можна назвати сервісом аутентифікації. Задача, яка стоїть перед цим блоком полягає у тому, щоб виконувати операції, пов'язані зі створенням та управлінням користувацькою сесією. Таким чином, цей блок повинен мати можливість створювати користувачів, створювати та знищувати сесії, а також підтримувати існуючі. В кінцевому рахунку саме завдяки цьому сервісу користувач має можливість отримати доступ до мікросервісів системи.

Для реалізації цього сервісу було обрано підхід з централізованим управлінням сесансами. При такому підході сесанси зберігаються у єдиному місці

безпосередньо на сервері. Для зберігання та обміну даними буде використовуватись токени, що дозволить зробити систему безстановою. Обрані рішення дозволяють системі бути достатньо гнучкою і дає їй можливість при потребі ефективно горизонтально розширюватись.

При цьому існує декілька підходів до реалізації аутентифікації на базі JWT. Основними при цьому є використання одного токenu — лише Access, або двох токенів разом з Refresh. Використання схеми з двома токенами значно впливає на безпечність і одночасно на зручність використання системи користувачам. Другий токен в такій схемі є довгоживучим і використовується для того, щоб автоматично час від часу оновлювати перший. Це дозволяє користувачу не турбуватись через те, що його сесія сплине з часом життя його токenu. Також другий токен підвищує безпечність завдяки тому, що перехват Access токenu зломисниками надасть лише тимчасовий доступ до даних користувача, так як знову ж таки після спливання часу життя Access токenu сервер видасть нову пару. Втрата Refresh токenu призведе до того, що поточна сесія користувача припиниться і він виконає новий вхід чим створить нову пару токенів. Звісно створення додаткового токenu підвищує часові витрати на обробку операцій користувача, але підвищення надійності системи того варті. Таким чином, схема Refresh і Access токенів обмежує час, на який атакуючий зломисник може отримати доступ до сервісу. Порівняно з одним токеном, яки може використовуватися тижнями, і ніхто про це не дізнається.

Зберігання даних зі сторони серверу буде відбуватися у базі даних, у якій будуть зберігатися інформація про безпосередньо користувача, а також про його активні сесії. Користувач, в свою чергу, отриману пару токенів буде зберігати у своїх запитах, як правило це відбувається у cookie, заголовках або у тілі відповіді. Refresh токен прийнято зберігати у cookie, в той час як зберігання Access токenu в більшості залежить від конфігурації клієнту.

Отже, поточний сервіс повинен мати наступні контролери для роботи з сесією користувача:

- контролер реєстрації — створення нового облікового запису користувача;
- контролер входу — створення та збереження нової сесії та видача пари tokenів користувачу;
- контролер виходу — видалення сесії та знищення tokenів;
- контролер оновлення пари tokenів — створення нових tokenів при спливу життя Access tokenу.

Почнемо з контролеру реєстрації. Метою цього алгоритму є створення нового користувача, який надалі зможе користуватися захищеними ресурсами розроблюваної системи. Життєвий цикл цього контролеру виглядає наступним чином:

1) Спочатку сервіс отримує маршрут, який був заданий користувачем. Далі необхідно отримати внутрішній token з даними, які він передає. Якщо token дійсний, тобто його зашифрований підпис співпадає, тоді відбувається витягування з нього даних. Цей етап є універсальним для кожного з алгоритмів, так як він передбачає верифікацію даних ще до того, як сервісом будуть виконуватись які-небудь дії, тому надалі цей етап буде опускатися;

2) Перед початком створення користувача необхідно перевірити, чи не існує вже він, або чи є дані цього користувача унікальними. На цьому етапі відбувається саме така перевірка і, якщо все гаразд, тоді відбувається наступний етап;

3) Далі необхідно створити самого користувача. При цьому процесі отримані з tokenу дані повинні потрапити у базу даних, де зберігаються користувачі;

4) При успішному завершенні процесу реєстрації користувачу повертається відповідне повідомлення, на чому реєстрації і завершується.

Рисунок 2.8 демонструє описаний алгоритм дій при реєстрації нового користувача.



Рисунок 2.8 — Алгоритм реєстрації користувача

Наступним розглянутим контролером буде контролер входу. Основна його мета полягає у управлінні користувацькими сесіями та видачі користувачу токенів, за допомогою яких він безпосередньо буде проходити процес аутентифікації для доступу до захищених ресурсів. Це етап є вкрай важливим з точки зору оптимізації через ряд ресурсозатратних етапів через які необхідно пройти. Отже розглянемо процес роботи цього контролеру:

- 1) Визначення шляху, отримання та перевірка внутрішнього токена;
- 2) У наступному етапі необхідно провести ідентифікацію користувача.

При цьому відбувається перевірка, чи існує запис про цього користувача у базі даних. Якщо він дійсно існує, тоді за допомогою отриманого паролю

здійснюється перевірка автентичності і, якщо обидва процеси завершуються успішно, тоді відбувається наступний етап;

3) Далі необхідно завершити застарілі сесії, які з новим входом втрачають свою актуальність. При цьому умови для визначення того, які сесії вважати застарілими, можуть бути різними, як, наприклад, безпосередньо обмеження на кількість одночасних сеансів, так і по параметрам користувача, як `device_id`;

4) Найважливішим етапом є створення пари Access і Refresh токенів. Кожен з токенів потребує унікального набору даних для підвищення своєї надійності і процес формування токенів є достатньо тривалим. Тут також важливо визначити і те, скільки кожний токен буде дійсним. Як правило для надійності час життя Access токена встановлюють на 60 хвилин, а Refresh — 30 днів. Тривалість життя може бути різною, але така схема є ідеальним компромісом між частотою звертань до сервера і потенційним часом доступу зломисника до даних користувача;

5) Далі необхідно зберегти сесію на сервері. Робити це можна завдяки збереженню Refresh токена у базі даних системи. Як саме відбувається перевірка чи є користувацька сесія справжньою буде описано під час розглядання процесу оновлення токенів;

6) Завершальним етапом є повернення користувачу пари створених ключів. Разом з ними також необхідно передати і супровідну інформацію таку як час життя Access токена.

На рисунку 2.9 зображено розглянутий алгоритм створення сесії користувача.

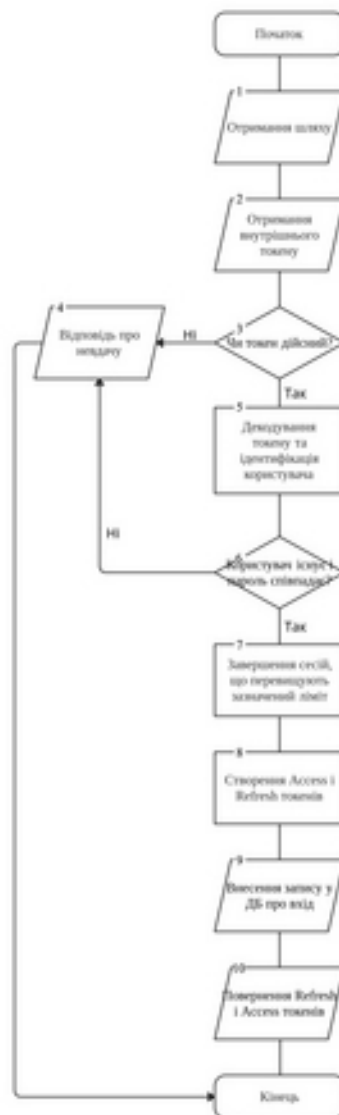


Рисунок 2.9 — Алгоритм створення сесії користувача

Далі розглянемо контролер виходу користувача. Перед ним стоїть задача припинення існуючих сесій, а також видалення наявних у користувача токенів. Покроково описати принцип роботи контролеру можна наступним чином:

- 1) Визначення шляху, отримання та перевірка внутрішнього токена;
- 2) Далі за рахунок існуючого Refresh токена необхідно знайти запис про поточну сесію у базі даних. Якщо така сесія існує, тоді її необхідно знищити, тим самим і припиниться дія сесії;
- 3) Після успішного завершення попередньої операції необхідно знищити Refresh токен і у користувача, який за визначеною схемою зберігається у cookie, а також відправити повідомлення про успішне завершення операції користувачу.

На рисунку 2.9 зображено алгоритм роботи контролеру виходу користувача.

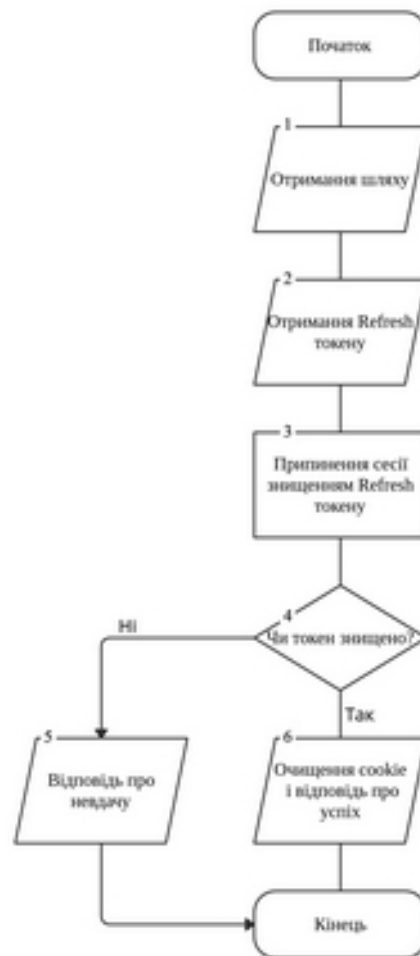


Рисунок 2.9 — Алгоритм видалення сесії користувача

Настав час розглянути алгоритм оновлення токенів. Виклик цієї частини програми відбувається тоді, коли час життя виданого раніше Access токєну сплинув і для продовження подальшої роботи користувачу необхідні нові токєни. При цьому виклик повинен відбуватися самостійно на стороні клієнту, відправною точкою чого може бути або встановлений на час життя Access токєну таймер, або при умові чергового запиту на сервер, коли у відповідь клієнт отримає повідомлення о припиненні терміну дії токєну.

Отже, розглянемо як саме відбувається процес роботи контролеру оновлення токенів користувача:

- 1) Визначення шляху, отримання та перевірка Refresh токена для того, щоб упевнитись, чи дійсно сесія існує і є коректною;
- 2) Далі завдяки Refresh токена відбувається пошук користувача у базі даних для отримання усіх додаткових необхідних даних для формування нової пари токенів;
- 3) Якщо створення відбулося успішно, тоді користувачу повертаються нові токени з усією додатковою інформацією.

На рисунку 2.10 зображено описаний алгоритм.



Рисунок 2.10 — Алгоритм оновлення токенів сесії користувача

2.5.3 Сервіс авторизації

Далі розглянемо сервіс авторизації. Мета, яка стоїть перед ним, полягає у тому, щоб однозначно визначати та контролювати права користувачів мікросервісної системи. Цей компонент логічно можна поділити на дві основні частини:

- центр керування політиками, який надає змогу визначати, керувати, тестувати та налаштовувати усі доступні правила контролю доступу системи;
- центр прийняття рішень на вхідний запит існує для того, щоб надавати рішення про видачу доступу, оцінюючи використовувані політики контролю доступу.

Таким чином сервіс успадковує централізовану модель, у якій присутня єдина точка прийняття рішень. Такий підхід робить систему досить гнучкою, що виражається у можливості оновлення правил контролю доступом незалежно від компонентів системи, у внесенні змін у політики без впливу на мікросервіс, у універсальності деяких правил для цілої групи мікросервісів, зменшуючи таким чином загальну кількість політик і правил. Також це надає змогу створювати правила для окремих мікросервісів, виносячи цю логіку за межі основного коду сервісу. Так що вибраний підхід надає найбільшу кількість можливостей для створення швидкого і оптимізованого сервісу авторизації.

Розглянемо умовну послідовність дій, яку повинен виконувати цей сервіс:

- 1) Спочатку центр керування політиками повинен визначити наявні правила контролю доступу для усіх присутніх у системі ролей. Це відбувається за допомогою отримання даних з інформаційного, у якому зберігаються усі визначені атрибути та політики доступу;
- 2) Далі коли користувач робить запит, він потрапляє на один з визначених захищених шляхів для перевірки прав доступу;
- 3) Після цього відбувається перевірка достовірності переданого користувачем внутрішнього токена, у якому зберігаються дані, які будуть використовувати для прийняття остаточного рішення;

4) У результаті на основі отриманого шляху, політик контролю доступу та переданих користувацьких даних у центрі прийняття рішень відбувається винесення остаточного рішення про надання користувачу доступу до захищеного ресурсу;

5) При успіху сервіс повертає повідомлення про успіх, на базі якого API-шлюз пропускає користувача до запитуваного ресурсу.

Життєвий цикл розробленого сервісу зображено на рисунку 2.11.

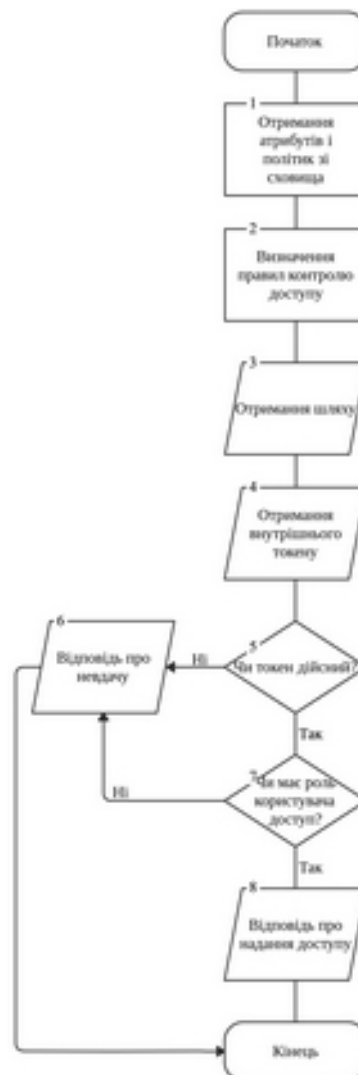


Рисунок 2.11 — Алгоритм сервісу авторизації

2.5.4 Загальна схема веб-додатку

Спроектувавши окремі компоненти мікросервісного додатку, необхідно описати і загальний алгоритм роботи системи. Розглянемо процес запиту користувача до захищеного мікросервісу:

1) Спочатку, коли клієнт виконує запит, він перехоплюється API-шлюзом, у якому відбувається первинна його обробка з метою визначення типу контенту, мети, кінцевої точки, URL-параметрів тощо;

2) Далі відбувається перевірка користувацького токена, якщо він присутній та дійсний, тоді відбувається наступний етап. Але якщо ні, тоді користувач отримує помилку і муситиме пройти процес аутентифікації, в результаті якого він отримає персональний токен доступу;

3) Після аутентифікації необхідно проводити авторизацію. Запит користувача потрапляє на сервіс авторизації, де за допомогою визначених політик та інформації о користувачі виконується перевірка прав доступу. Якщо користувач має право взаємодіяти з захищеним ресурсом, тоді відбувається наступний етап;

4) Запит користувача повертається до API-шлюзу, де його готують до переадресації безпосередньо на мікросервіс. Після чого він має можливість взаємодіяти безпосередньо з ним.

У загальному виді спроектовану систему зображено на рисунку 2.12.

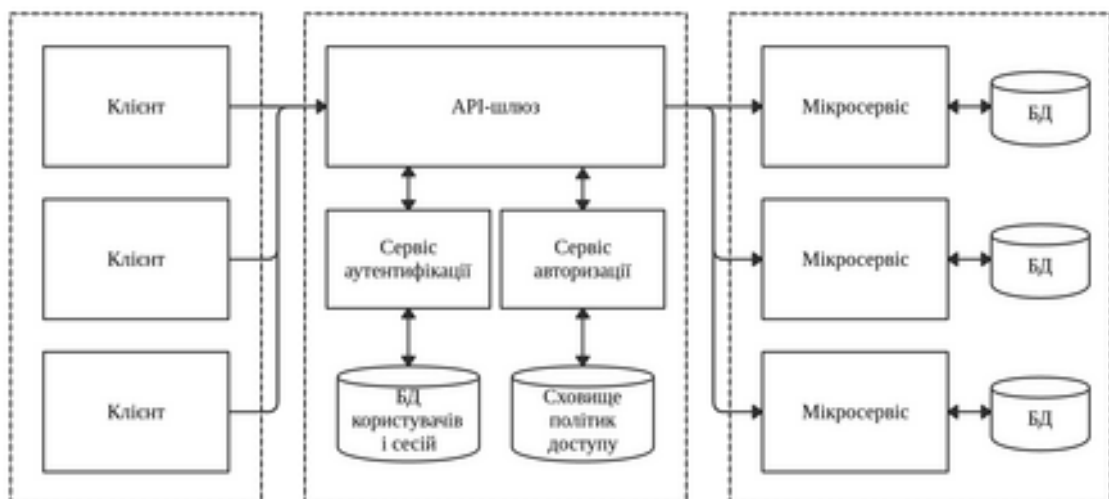


Рисунок 2.12 — Загальна схема системи

2.6 Висновки проектування алгоритмів

У поточному розділі було розглянуто існуючі реалізації алгоритмів роботи мікропроцесорних систем, а також на їх основі було спроектовано покращені версії цих алгоритмів.

На початку розділу було розглянуто існуючі концепції технічної реалізації аутентифікації для систем з мікросервісною архітектурою. Було розглянуто алгоритми їх роботи, особливості реалізації, визначені існуючі недоліки та переваги. Окремо було розглянуто протокол OAuth 2.0, а також заснований на ньому OpenID Connect.

В ході роботи було проведено огляд готових варіантів реалізації аутентифікації та авторизації для мікросервісних систем. Було виділено декілька категорій їх класифікації, наприклад за місцем розгортання, або за наявністю імплементації окремих проху-серверів. Було розглянуто архітектуру цих систем, визначено особливості та наявні можливості, на які здатна та чи інша система.

За допомогою отриманих даних було визначено основні функціональні компоненти, з яких повинна складатися розроблювана система, а також були визначені слабкі місця, які необхідно покращити. В результаті було спроектовано нові вдосконалені алгоритми мікросервісної системи.

3 РЕАЛІЗАЦІЯ АЛГОРИТМІВ ТА ТЕСТУВАННЯ СИСТЕМИ

Наступним етапом роботи є реалізація та тестування спроектованої системи. Використовувані технології та методики значно впливають на кінцеву ефективність та продуктивність як окремих компонентів системи, так і одразу всього додатку. Але у цьому розділі важливо приділити увагу не тільки безпосередньо методикам розробки, оптимізації процесів шифрування, передачі даних та використання пам'яті, але й розгортанню системи, так як це також в значній мірі впливає на часові витрати системи.

3.1 Реалізація алгоритму API-шлюзу

Почнемо з реалізації API-шлюзу. Як було визначено раніше, для створення усіх компонентів системи було обрано фреймворк Express, що базується на Node.js. Створення додатку на основі цього фреймворку виглядає наступним чином:

```
const express = require('express');
const app = express();
```

Для коректної роботи системи необхідно встановити змінні оточення.

Зробити це можна наступним чином:

```
require('dotenv').config();
```

При цьому для використання змінних, вони повинні знаходитись у спеціальному файлі, у цьому випадку під назвою `.env.development`. Цей файл містить змінні для формування токенів, номер порту API-шлюзу, а також режим роботи сервісу. Отже вміст цього файлу є наступним:

```
PORT=8081
NODE_ENV=development
JWT_SECRET=eyJpc3MiOiJhc2RhZGEiLCJpYXQiOiJlY2NTgZ0Tk3MTUsImV4cCI6MTY4OTkzNTcx
JWT_INNER_SECRET=nfsGw3nfi3n9scKksjj1ldpqqNzdhe3
```

Наступним кроком у системі визначаються проміжні функції, метою яких є оптимізації та обробка вхідних даних. У лістингу 3.1 спочатку виконується стискання тіла вхідного запиту, далі додаються обробники різних типів контенту. Далі за допомогою методу `helmet()` додаються HTTP-заголовки, які використовуються, щоб допомогти захистити запити. Потім встановлюються CORS — механізм, що використовується, щоб надати можливість клієнту отримувати дозволи на доступ до вибраних ресурсів з іншого домену. В кінці функції додається логгер.

Лістинг 3.1 — Обробники вхідного запиту

```
module.exports = (app) => {
  app.use(compression());
  app.use(express.urlencoded({
    extended: true,
  }));
  app.use(express.json());
  app.use(helmet());
  app.use(cors());
  app.use(expressWinston.logger({
    transports: [
      new winston.transports.Console(),
    ],
    requestWhitelist: ['url', 'method', 'httpVersion',
'originalUrl', 'query'],
    format: winston.format.combine(
      winston.format.json(),
    ),
  }));
};
```

Шлюз повинен мати можливість працювати як з HTTP, так і з HTTPS. При чому для створення HTTPS серверу сертифікати отримуються з файлу конфігурації. Реалізація цього функціоналу продемонстровано у лістингу 3.2.

Лістинг 3.2 — Обробники вхідного запиту

```
if (config.protocol === 'http') {
  this.http = app.listen(config.http.port || 8081,
config.http.host, logUrl('http'));
}
if (config.protocol === 'https') {
  const key = fs.readFileSync(config.https.key, 'utf8');
```

```

    const cert = fs.readFileSync(config.https.cert, 'utf8');
    const ca = config.https.ca &&
fs.readFileSync(config.https.ca, 'utf8');
    this.https = https
      .createServer({ key, cert, ca }, app)
      .listen(config.https.port || 8443, logUrl('https'));
  }

```

Для доступу до окремих мікросервісів у шлюзі використовується вбудований у Express клас Router. А сама маршрутизація реалізується в окремому файлі та підключається наступним чином:

```

const routes = require('./src/router');
app.use(routes);

```

Маршрутизація у цьому файлі є динамічною та генеруватимуться під час запуску шлюзу зі конфігураційного списку за наступною схемою:

- host : String — вказує на цільовий хост проксі;
- path : String — цільовий URL-шлях для запитів;
- auth : Boolean — визначає захищені ресурси, які вимагають

аутентифікації для кінцевої точки.

Сама маршрутизація у конфігураційному файлі реалізується на базі JSON. Схему маршрутизації зображено у лістингу 3.3.

Лістинг 3.3 — Схема маршрутизації

```

routes: {
  get: [
    {
      host: 'hostname',
      path: '/some/path',
      auth: true,
    },
  ],
  post: [
    { ... }
  ],
  put: [
    { ... }
  ],
  delete: [
    { ... }
  ],
}

```

Роутер являє собою цикл, у якому із даних, отриманих з конфігураційного файлу, формується кожен окремий шлях. Масив `routeConfig`, із якого формується маршрут містить маршрут, проміжні функції, а також кінцеву точку запиту. Саме тут виконується перевірка валідності токена та наявних прав доступу. Цикл зображено у лістингу 3.4.

Лістинг 3.4 — Цикл формування маршрутів

```
each(routes, (route, k) => {
  each(route, (r) => {
    const { host, path, auth } = r;

    const routeConfig = [
      path,
      auth && checkJwt({ secret: jwtSecret }),
      auth && accessControl(k),
      redirect(host, path),
    ];

    router[k](...routeConfig.filter(e => e));
  });
});
```

Функція `accessControl()` виконує запит до сервісу авторизації. При цьому разом із запитом передаються наявні заголовки, серед яких повинен бути и токен аутентифікації `Bearer Authentication`. Якщо сервіс поверне відповідь про успіх, тоді завдяки методу `next()` відбудеться виклик наступної функції, а інакше при забороні доступу відбудеться помилка. Код функції `accessControl` продемонстровано у лістингу 3.5.

Лістинг 3.5 — Функція контролю доступу

```
const accessControl = (type) => (req, res, next) => {
  axios[type](`${authorization.host}${req.path}`, {
    headers: req.headers,
  })
  .then((res) => {
    next();
  })
  .catch((err) => {
    next(err.response.data);
  })
}
```

Після виконання усіх проміжних функцій, якщо при цьому не відбулося ніяких помилок і доступ все ж таки було надано, до запит потрапляє до фінальної функції `redirect()`, яка і виконує безпосередню переадресацію. У цій функції викликається метод `requestProxy()`, який в якості параметрів отримує адрес переадресації та додатково встановлюється заголовок із внутрішнім токеном. Код функції продемонстровано у лістингу 3.6.

Лістинг 3.6 — Функція переадресації на мікросервіс

```
const redirect = (host, path) => (req, res, next) => {
  const proxy = requestProxy({
    url: `${host}${path}`,
    headers: {
      'content-type': 'text/plain',
      'i-token': jwt.sign(req.body, jwtInnerSecret),
    },
  });
  proxy(req, res, next);
}
```

3.2 Реалізація сервісу керування сесіями

Першою перепорою на шляху отримання доступу до захищеного ресурсу є наявність у користувача токена аутентифікації. Для його отримання необхідно звертатись до сервісу керування сесіями, який оброблює запити, що пов'язані як раз з аутентифікацією. Отже розглянемо процес розробки цього сервісу.

В плані налаштування сервера цей сервіс має не значні відмінності від налаштування API-шлюзу, тому цей етап буде пропущено. Почнемо з маршрутизації, яка має наступний вигляд:

```
router.post('/signin', tokenChecker, Controller.signIn);
router.delete('/signout', Controller.signOut);
router.post('/signup', tokenChecker, Controller.signUp);
router.get('/refresh', Controller.refresh);
```

Для кожного шляху указується свій HTTP-метод, шлях та контролер, що оброблює запит та повертає результат клієнту. Також при обробці запиту використовується проміжна функція `tokenChecker()`, мета якої полягає у виконанні попередньої верифікації вхідного внутрішнього токenu та у декодуванні його вмісту, який далі буде передано у контролер запиту. Вміст проміжної функції `tokenChecker` показано у лістингу 3.7.

Лістинг 3.7 — Функція перевірки токenu

```
module.exports = (req, res, next) => {
  const token =
req.headers[process.env.JWT_INNER_SECRET_HEADER_NAME];

  if (token) {
    const { JWT_INNER_SECRET } = process.env;

    jwt.verify(token, JWT_INNER_SECRET, async (err, decoded) =>
{
      if (err) {
        next(createHttpError(403, err));
      }
      req.body = decoded;
      next();
    });
  } else {
    next(createHttpError(403, 'No token provided.'));
  }
};
```

Так вміст токenu потрапляє до об'єкту `req`, який, в свою чергу, використовуватиметься у контролері запиту. Але якщо токен не буде знайдено або під час верифікації буде виявлено, що його було скомпрометовано, тоді функція поверне клієнту помилку.

Робота сервісу полягає у керуванні користувачами і сесіями, що виконується завдяки роботі з базою даних. Тому база даних сервісу повинна мати відповідні таблиці.

Структуру таблиці `users` описано у таблиці 3.1.

Таблиця 3.1 – Структура таблиці users

Поле	Тип	Значення за замовчуванням	Примітка	Ключ
id	integer	-	not null, autoincrement, unique	primary
username	string(128)	-	not null, unique	-
role	enum('basic', 'moderator', 'admin')	'basic'	not null	-
email	string	-	not null, unique	-
password_hash	text	-	not null	-
createdAt	date	current_timestamp	not null	-
updatedAt	date	current_timestamp	not null	-

Структура таблиці refresh_tokens показано у таблиці 3.2.

Таблиця 3.2 – Структура таблиці refresh_tokens

Поле	Тип	Значення за замовчуванням	Примітка	Ключ
id	integer	-	not null, autoincrement, unique	primary
userId	integer	-	not null	foreign
value	text	-	not null	-
ua	string	-	-	-
expiresIn	date	-	not null	-
createdAt	date	current_timestamp	not null	-
updatedAt	date	current_timestamp	not null	-

Завдяки використанню Sequelize взаємодія з базою відбувається за допомогою об'єктно-орієнтованої парадигми. Sequelize використовується для представлення даних у вигляді об'єктів, що значно спрощує їх керування. Для початку роботи необхідно виконати налаштування підключення до бази даних, що виконується наступною командою:

```
npx sequelize-cli init
```

В результаті буде отримано файл config/config.json, який було додатково налаштовано наступним чином:

```
{
  "development": {
    "username": "postgres",
    "password": "password",
    "database": "dp-ms",
    "host": "localhost",
    "dialect": "postgres",
    "operatorsAliases": "Op",
    "migrationStorage": "json",
    "seederStorage": "json"
  }
}
```

Далі необхідно створити саму базу даних, міграції та моделі, що виконується наступними командами:

```
npx sequelize-cli db:create
npx sequelize-cli model:generate --name User --attributes
username:string,role:string,email:string,password:string
npx sequelize-cli model:generate --name RefreshToken --
attributes userId:integer,value:text,ua:string,expiresIn:date
```

В результаті виконання у проєкті будуть створені файли міграції та моделі, які також необхідно дещо відредагувати. Розглянемо файл моделі користувача. Модель існує для опису таблиці на рівні додатку. На ній виконується превентивна валідація даних, опис полів таблиці, створюються асоціації з іншими таблицями. У лістингу 3.8 продемонстровано властивості моделі, які відповідають полям таблиці.

Лістинг 3.8 — Властивості моделі користувача

```
User.init({
  username: {
    allowNull: false,
    unique: true,
    type: DataTypes.STRING(128),
```

```

    validate: {
      allowNull: true,
      notEmpty: true,
      is: /^[a-z0-9_-]{3,15}$/i,
    }
  },
  role: {
    allowNull: false,
    type: DataTypes.ENUM('basic', 'moderator', 'admin'),
    defaultValue: 'basic',
  },
  email: {
    allowNull: false,
    unique: true,
    type: DataTypes.STRING,
    validate: {
      allowNull: true,
      notEmpty: true,
      isEmail: true,
    }
  },
  password: {
    field: 'password_hash',
    allowNull: false,
    type: DataTypes.TEXT,
  },
}, {
  sequelize,
  modelName: 'User',
  tableName: 'users',
  underscored: true,
});

```

Завдяки хукам виконується хешування паролю користувача для запобігання збереженню важливих персональних даних у таблиці. У лістингу 3.9 продемонстровано хуки, що викликаються перед створенням та оновленням даних користувача, а також саму функцію хешування.

Лістинг 3.9 — Хешування пароля користувача

```

User.beforeCreate(async (user, options) => {
  user.password = await user.hashPassword(user);
});
User.beforeUpdate(async (user, options) => {
  user.password = await user.hashPassword(user);
});

async hashPassword(user) {
  if (user.changed('password')) {

```

```

        const saltRounds = parseInt(process.env.SALT_ROUNDS);
        return await bcrypt.hash(user.password, saltRounds);
    }
    return user.password;
};

```

А завдяки функції `comparePassword()` відбувається перевірка коректності паролю:

```

    async comparePassword(password) {
        return await bcrypt.compare(password,
this.getDataValue('password'));
    }

```

Так як користувачі та сесії пов'язані зв'язком один до багатьох, необхідно створити зв'язок і у моделі. Це, в тому числі, дасть змогу використовувати так звані “магічні методи”, які надаватимуть можливість звертатись до пов'язаних даних. Для встановлення зв'язків між моделями необхідно у статичному методі `associate()` створити відповідний зв'язок. Відповідний метод продемонстровано у лістингу 3.10.

Лістинг 3.10 — Асоціації моделі User

```

static associate({ RefreshToken }) {
    User.hasMany(RefreshToken, {
        foreignKey: 'userId',
    });
}

```

Наступним розглянемо модель сесій. У лістингу 3.11 продемонстровано властивості цієї моделі.

Лістинг 3.11 — Властивості моделі сесій

```

RefreshToken.init({
    userId: {
        field: 'user_id',
        allowNull: false,
        type: DataTypes.INTEGER,
    },
    value: {
        allowNull: false,
        type: DataTypes.TEXT,
    },
    ua: {

```

```

    type: DataTypes.STRING,
  },
  expiresIn: {
    field: 'expires_in',
    allowNull: false,
    type: DataTypes.DATE,
    set(expire) {
      this.setDataValue('expiresIn', new Date(expire * 1000));
    }
  },
}, {
  sequelize,
  modelName: 'RefreshToken',
  tableName: 'refresh_tokens',
  underscored: true,
});

```

Для знищення застарілих токенів, використовується функція `destroyExpiredTokens()`, яка викликається у cron в заданий час знаходячи и знищуючи поля сесій, життя яких минуло пізніше поточного моменту. Вміст функції показано у лістингу 3.12.

Лістинг 3.12 — Функція знищення застарілих токенів

```

static async destroyExpiredTokens() {
  const destroyedCount = await RefreshToken.destroy({
    where: {
      expiresIn: { [Op.lte]: new Date() }
    }
  });
  console.log(`<----- Destroyed ${destroyedCount} refresh
tokens`);
}

```

Перевірка існування переданого токена у базі відбувається завдяки функції `isTokenExists()`. Її вміст продемонстровано у лістингу 3.13.

Лістинг 3.13 — Функція перевірки існування токена

```

static async isTokenExists(token) {
  return RefreshToken
    .findOne({ where: { value: token } })
    .then(() => true)
    .catch(() => false);
}

```

Асоціація сесій має схожий вид як і у користувачів за одним лиш виключенням: кожна сесія має лише одного користувача, у чому і полягає відмінність. Лістинг 3.14 містить вміст методу асоціації.

Лістинг 3.14 — Асоціації моделі RefreshToken

```
static associate({ User }) {
  RefreshToken.belongsTo(User, {
    foreignKey: 'userId',
  });
}
```

Тепер настав час розглянути безпосередньо наявні контролери. І почнемо з контролеру реєстрації. Із об'єкту `req` він отримує вилучені раніше дані із токена. За допомогою цих даних буде виконано запит до бази даних зі спробою створити нового користувача. Якщо це матиме успіх, клієнт отримає відповідне повідомлення, а інакше — повідомлення про помилку. У лістингу 3.15 продемонстровано вміст розглянутого контролеру.

Лістинг 3.15 — Контролер реєстрації

```
module.exports.signUp = async (req, res, next) => {
  try {
    const { body } = req;
    await User.create(body);
    res
      .status(200)
      .send(getResponse('Signed up successfully!'));
  } catch (error) {
    next(createHttpError(401, 'User existed'));
  }
};
```

Далі розглянемо контролер входу, завдяки якому створюється нова сесія. До цього контролеру завдяки проміжній функції `tokenChecker` (лістинг 3.7) потрапляють вже готові дані користувача. Використовуючи ці дані спочатку відбувається ідентифікація користувача пошуком його у базі даних. Після цього порівнюється його пароль, якщо він вірний, тоді контролер виконує знищення старих сесій та створює пару нових токенів. В кінці робиться запис Refresh токена

до бази, чим відбувається збереження поточної сесії, і в кінці отримані токени повертаються користувачу. У лістингу 3.16 показано описаний контролер входу користувача до системи.

Лістинг 3.16 — Контролер входу

```
module.exports.signIn = async (req, res, next) => {
  try {
    const { body: { email, password, ua } } = req;

    const user = await User.findOne({ where: { email } });

    if (user && await user.comparePassword(password)) {
      await destroyOverLimitTokens(user);

      const { accessToken, accessExp } = await
getAccessToken({ id: user.id, email, username: user.username, role:
user.role });
      const { refreshToken, refreshExp } = await
getRefreshToken({ email, ua, username: user.username });

      await user.createRefreshToken({
        value: refreshToken,
        expiresIn: refreshExp,
        ua,
      });

      res
        .cookie(REFRESH_TOKEN_NAME, refreshToken,
getTokenCookieOptions(refreshExp))
        .status(200)
        .send(getResponse('Logged in.', { token: accessToken,
expiresIn: accessExp }));
    } else {
      next(createHttpError(401, 'Incorrect provided login
data.'));
    }

  } catch (error) {
    next(error);
  }
};
```

Для того, щоб знищувати застарілі сесії, які виходять за ліміт, використовується функція `destroyOverLimitTokens()`. В якості параметра вона отримує екземпляр користувача, завдяки чому вираховується кількість застарілих токенів, після чого це значення порівнюється із заданим заздалегідь і

вираховується необхідна кількість сесій на знищення. Вміст функції знищення перевищуючих заданий ліміт токенів показано у лістингу 3.17.

Лістинг 3.17 — Функція знищення застарілих токенів

```
module.exports.destroyOverLimitTokens = async (user) => {
  const refreshTokensCount = await user.countRefreshTokens();

  if (refreshTokensCount >= MAX_AUTH_COUNT) {
    const destroyCount = refreshTokensCount - MAX_AUTH_COUNT +
1;

    await RefreshToken.destroy({
      where: { userId: user.id },
      limit: destroyCount,
    });
  }
};
```

Наступним розглянутим контролером буде контролер виходу. Завдяки йому відбувається знищення поточної сесії користувача. Спочатку із заголовка запиту дістається Refresh токен та відбувається його знищення. При умові, якщо це успішно вдалось зробити, користувач отримує відповідне повідомлення та його поточний токен очищається з cookie. Описаний процес роботи контролеру виходу продемонстровано у лістингу 3.18.

Лістинг 3.18 — Контролер виходу

```
module.exports.signOut = async (req, res, next) => {
  try {
    const refreshToken = req.cookies[REFRESH_TOKEN_NAME];

    const isTokenDestroyed = await
RefreshToken.destroy({ where: { value: refreshToken } });

    if (isTokenDestroyed) {
      res
        .clearCookie(REFRESH_TOKEN_NAME,
getTokenCookieOptions())
        .status(200)
        .send(getResponse('Logged out. '));
    } else {
      next(createHttpError(401, 'Incorrect provided
token. '));
    }
  }
};
```

```

    }
  } catch (error) {
    next(error);
  }
};

```

Останнім контролером є контролер оновлення сесії. Цей контролер є дещо схожим по своїй суті і побудові на контролер входу, але й з рядом ключових відмінностей. Спочатку у ньому відбувається перевірка існування Refresh токена, при успіху на базі отриманих з токена даних відбувається пошук користувача, для якого згодом створюється пара нових tokenів. Далі відбувається оновлення сесії видаленням старого Refresh токена з бази та додаванням нового. В кінці користувачу повертається відповідь з новою парою tokenів. У лістингу 3.19 показано цей контролер.

Лістинг 3.19 — Контролер оновлення сесії

```

module.exports.refresh = async (req, res, next) => {
  try {
    const { cookies, ua } = req;
    const refreshToken = cookies[REFRESH_TOKEN_NAME];
    const isTokenExists = await
RefreshToken.isTokenExists(refreshToken);

    if (isTokenExists) {
      const email = decode(refreshToken).email;
      const user = await User.findOne({ where: { email } });

      const { accessToken, accessExp } = await getAccessToken({
id: user.id, email, username: user.username, role: user.role });
      const { refreshToken, refreshExp } = await
getRefreshToken({ email, ua, username: user.username });

      await RefreshToken.destroy({ where: { value: refreshToken
} });

      await user.createRefreshToken({
        value: refreshToken,
        expiresIn: refreshExp,
        ua,
      });

      res
        .status(200)
        .cookie(REFRESH_TOKEN_NAME, refreshToken,
getTokenCookieOptions(refreshExp))

```

```

        .send(getResponse('Updated access token.', { token:
accessToken, expiresIn: accessExp }));
    } else {
        next(createHttpError(401, 'Incorrect provided
token.'));
    }

    } catch (error) {
        next(error);
    }
};

```

3.3 Реалізація сервісу авторизації

Після успішного проходження аутентифікації у користувача повинен з'явитися Access токен. Його мета — отримання доступу до захищених ресурсів і саме з його допомогою виконується перевірка прав доступу користувача. Так що наступним необхідно розглянути процес розробки сервісу авторизації.

Розроблюваний сервіс авторизації складається з двох основних частин, де перша — це розділ отримання та визначення правил та політик системи, завдяки яким відбувається контроль доступу. Цей блок повинен виконуватись на початку роботи сервісу, тобто при його запуску. Друга частина — маршрутизатор з шляхами для яких необхідно застосувати визначені політики.

Для початку розглянемо процес визначення політик сервісу. Складається він з етапу отримання політик зі сховища, після чого виконується визначення, які операції може виконувати кожна з існуючих ролей. Завдяки реалізованим у класі AccessControl методам створюються усі необхідні обмеження доступу. У лістингу 3.20 показано процес створення правил доступу.

Лістинг 3.20 — Визначення правил доступу

```

const ac = new AccessControl();
const { basic, moderator, admin } = roles;
const { profile, posts } = resources;

exports.permissions = (() => {

```

```

    ac.grant(basic)
      .readOwn(profile)
      .updateOwn(posts)
      .readAny(posts);

    ac.grant(moderator)
      .extend(basic)
      .readAny(profile);

    ac.grant(admin)
      .extend(basic)
      .extend(moderator)
      .updateAny(posts)
      .deleteAny(posts);

    return ac;
  })();

```

Далі коли користувач виконує запит, він потрапляє на один із визначених шляхів, що відповідають маршрутам API-шлюзу, але тут визначені лише необхідні шляхи. При цьому в якості контролеру тут виступає функція `grantAccess()`, завдяки якій можливо визначити права доступу до окремого маршруту. Виклик цього контролеру має наступний вигляд:

```
Controller.grantAccess('readAny', 'profile')
```

В якості першого аргументу сюди необхідно передати тип запитуваної дії, а в якості другого — тип ресурсу. Таким чином для кожного маршруту однозначно визначаються умови доступу. У самому контролеру спочатку відбувається верифікація токenu аутентифікації, далі за отриманою з цього токenu роллю виконується перевірка, чи має право користувач отримати доступу до ресурсу. В кінці користувачу повертається повідомлення з результатом авторизації. Код описаного контролеру наведено у лістингу 3.21.

Лістинг 3.21 — Контролер визначення прав доступу

```

exports.grantAccess = function (action, resource) {
  return async (req, res, next) => {
    const token = req.headers.authorization.split(' ')[1];

    if (token) {
      const { ACCESS_TOKEN_SECRET } = process.env;

```

```

    jwt.verify(token, ACCESS_TOKEN_SECRET, async (err, decoded)
=> {
    if (err) {
        next(createHttpError(403, err));
    }
    try {
        const permission = permissions.can(decoded.role)
[action](resource);

        if (!permission.granted && Number(req.params.id) !==
decoded.id) {
            return res
                .status(401)
                .send('You don\'t have enough permission to perform
this action')
        }
        return res
            .status(200)
            .send('OK');
    } catch (error) {
        next(error)
    }

    });
} else {
    next(createHttpError(403, 'No token provided.'));
}
}
};

```

3.4 Тестування системи

Завершальним етапом розробки системи є проведення фінального тестування отриманого результату. Для виконання запитів до мікросервісів буде використовуватись програмне забезпечення під назвою Postman. Postman — це інструмент для роботи з API, який дозволяє розробнику надсилати запити до сервісів та працювати з їхніми відповідями.

Отже, почнемо з тестування несанкціонованого доступу до захищеного ресурсу. При спробі виконати такий запит, клієнту буде повернено відповідну помилку, що показано на рисунку 3.1.

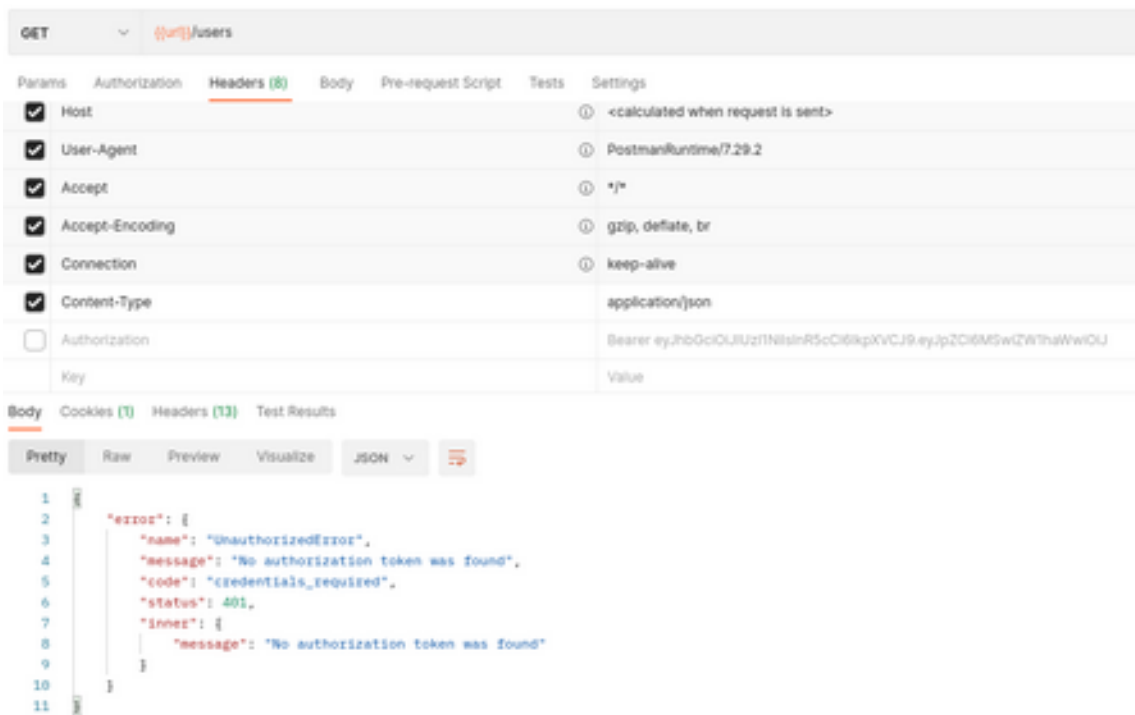


Рисунок 3.1 — Відповідь на несанкціонований доступ

Тому для доступу користувачу необхідно пройти реєстрацію. Це досягається виконанням POST запиту по шляху `/auth/signup`. Користувач у формі вносить свої дані для реєстрації та відправляє їх на сервер. Результат успішної реєстрації показано на рисунку 3.2.

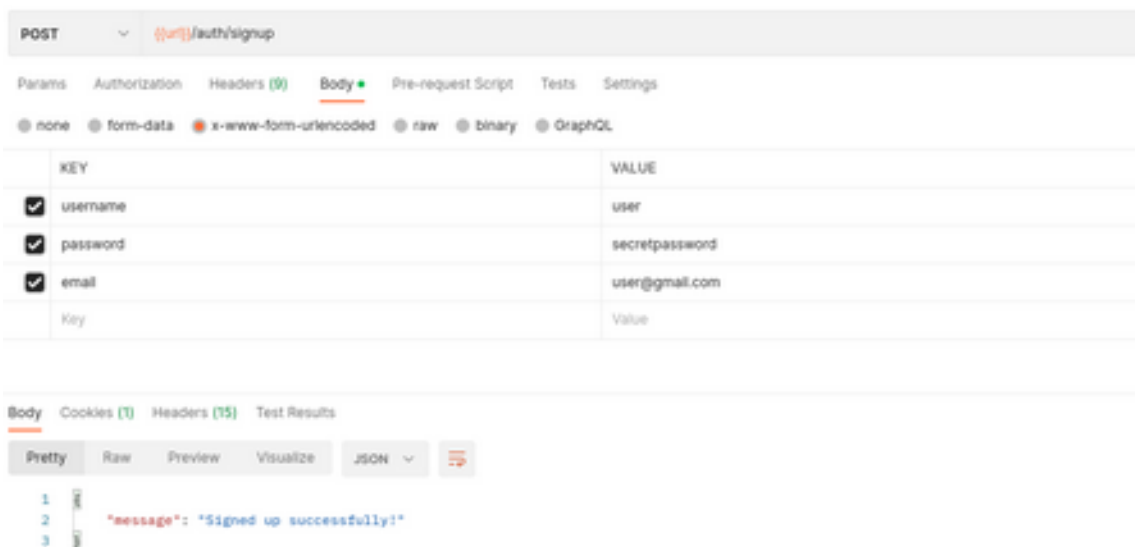


Рисунок 3.2 — Відповідь про успішну реєстрацію

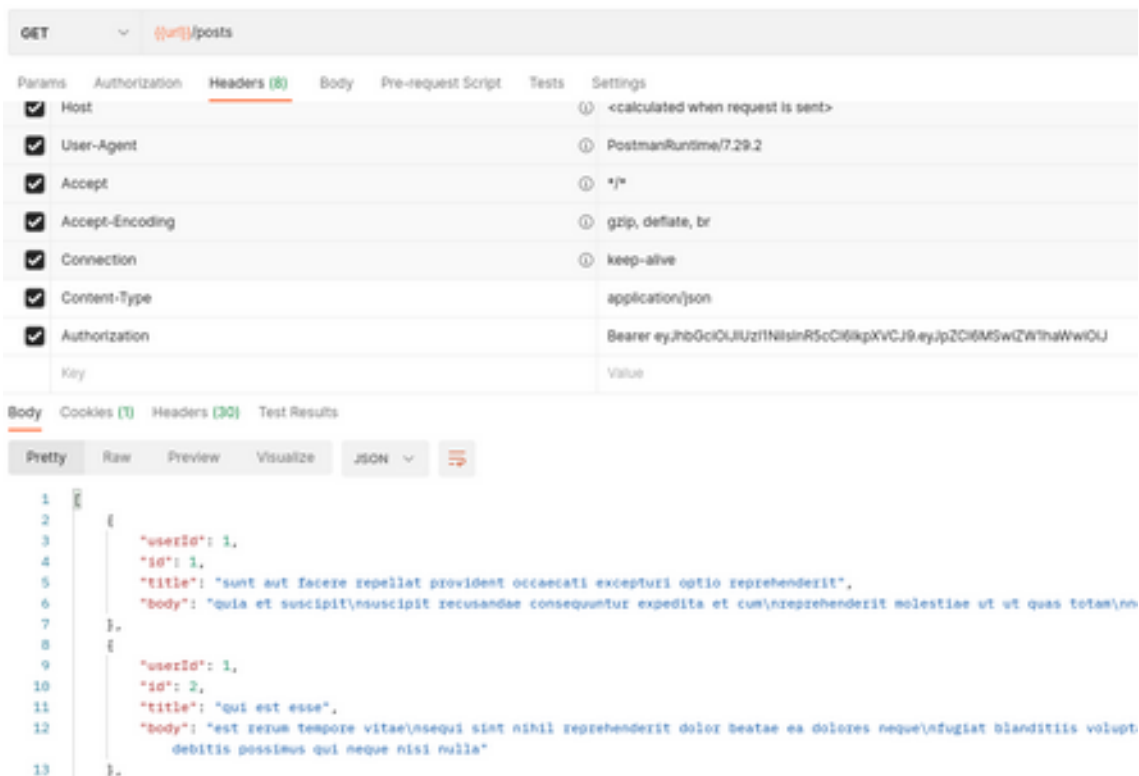


Рисунок 3.4 — Виклик з отриманням даних постів

Але в рамках системи у користувача немає доступу до перегляду інших користувачів окрім себе, що показано на рисунку 3.5.

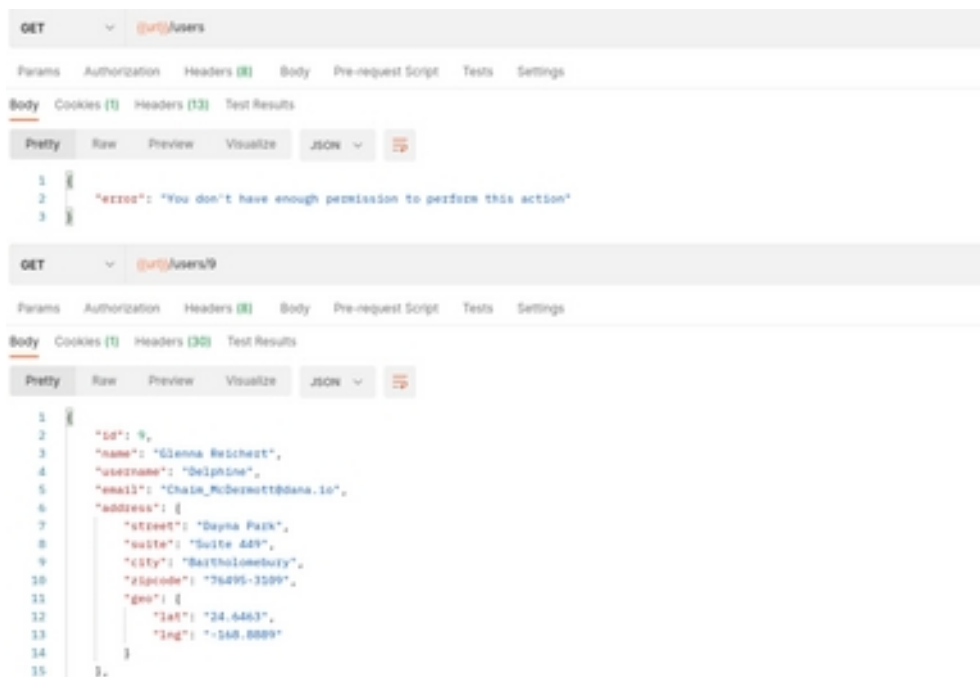


Рисунок 3.5 — Виклик з отриманням даних користувачів

компонентів аутентифікації мікропроцесорних систем. В результаті було отримано три сервіси, в яких реалізуються зазначені алгоритми, а саме: API-шлюз, який виконує роль єдиної точки входу, керує та оброблює вхідний запит від користувача. Серед основних його задач можна виділити маршрутизацію, верифікацію користувача та переадресацію на сервіси керування сесіями та авторизації. У сервісу авторизації відбувається перевірка прав користувача на доступ до захищеного ресурсу. Сервіс керування сесіями виконує завдання, пов'язані зі створенням та керуванням користувачем, його сесіями, а також його аутентифікацією.

Далі було проведено тестування отриманих сервісів системи, в результаті чого система продемонструвала повну працездатність.

4 ДОСЛІДЖЕННЯ СИСТЕМИ

Ключовим етапом роботи є проведення дослідження ефективності розроблених алгоритмів мікросервісної системи. У цьому розділі відображаються результати навантажувальних тестів, мета яких визначити в кількісних характеристиках продуктивність розробленої системи. В рамках дослідження будуть порівнюватися запропонована схема мікросервісної системи з класичним варіантом реалізації з моделлю шлюзу пограничного рівня.

4.1 Методи оцінювання продуктивності мікропроцесорної системи

Під продуктивністю мікросервісної архітектури звичайно розуміють деякий набір показників, за допомогою яких виконується оцінювання ефективності роботи з урахуванням часових обмежень системи з мікросервісною архітектурою

загалом та окремих її частин. Це визначення охоплює деякі загальні показники, які можна використовувати та будь-якої комп'ютерної системи, та показники, що можливо застосовувати лише в рамках існуючої предметної області, як наприклад, середня кількість проходжень аутентифікації за обмежений проміжок часу.

Надійність мікросервісної системи можна визначити як набір атрибутів, які описують здатність системи виконувати визначені завдання з належним рівнем якості за встановлених умов за встановлений період часу. З цього слідує, що здатність системи виконувати визначену кількість завдань виступає і показником її продуктивності, і показником її надійності. Тому система, що не в змозі дотримуватись заданих для неї тимчасових обмежень, не може вважатись надійною.

В рамках аналізу продуктивності мікросервісну архітектуру можна розглядати у двох видах: як окремий сервіс та їх композицію. Перший випадок є тривіальним завданням по причині того, що область аналізу продуктивності комп'ютерних і програмних систем має достатньо визначених стандартів.

Але в свою чергу розрахунок продуктивності розподілених систем є значно складнішим завданням. Під продуктивністю комплексу різнорідних сервісних систем зазвичай розуміється поєднання надійності, вартості, відмовостійкості, часу виконання завдань та інших показників.

Загалом методи оцінки продуктивності мікросервісних систем можна поділити на два напрямки, а саме: з використанням математичного чи імітаційного моделювання. У першому напрямку наголошується на аналізі структури системи, оцінюючи, наприклад, продуктивність залежно від типових конструкцій, як то циклів чи умовних конструкцій. Але методи аналізу цієї групи є достатньо складним у реальному застосуванні, при чому складність зростає експоненційно з підвищенням складності самої системи. В свою чергу метод імітаційного моделювання значно краще підходить для оцінювання якості роботи мікросервісних систем, але при цьому виникає проблема відсутності універсальних методів та технологій.

4.2 Визначення показників оцінювання продуктивності системи

Отже необхідно визначитись з показниками продуктивності, за якими буде об'єктивно досліджуватись ефективність розроблена система. Значна кількість цих показників є універсальною для багатьох комп'ютерних систем, що дозволяє їх класифікувати за деякими критеріями. Так показники, за якими буде відбуватися оцінювання можна поділити на наступні категорії:

- часові витрати — до цієї групи входять показники, які відображають різні часові характеристики роботи мікросервісної системи;
- кількісні показники — до групи входять показники, що визначають продуктивність системи кількісно.

Таким чином, за визначеними категоріями можна сформулювати можна представити скласти групи наборів показників. У таблиці 4.1 представлено часові показники оцінювання.

Таблиця 4.1 — Часові показники

Назва	Опис
Час виконання	Ключова характеристика, яка визначає час, що необхідний для повного виконання завдання.
Час очікування	Час, який запит витратив в очікуванні обробки.
Час отримання відповіді	Час, витрачений на отримання даних відповіді від сервісу.
Тривалість ітерації	Час, необхідний для завершення однієї повної ітерації, включаючи час, витрачений на налаштування та демонтаж.

Наступною категорією оцінювання є набір кількісних показників. Дані показники наведено у таблиці 4.2.

Таблиця 4.2 — Часові показники

Назва	Опис
Паралельність	Кількість паралельних завдань, які може виконувати система.
Пропускна здатність	Кількість завдань, що може оброблятися системою за одиницю часу.
Навантаженість системи	Показники навантаження ЦП та ОЗУ серверу, на якому розташований сервіс.

4.2 Оцінка продуктивності системи

Тепер, коли основні показники оцінювання мікросервісної системи були визначені, необхідно провести дослідження продуктивності роботи розробленої системи, і за результатами тестування буде надано об'єктивну оцінку її роботи.

Проводитись замір показників буде за різної кількості користувачів. В якості платформи для тестування буде використовуватися К6. Вона надає змогу створювати високонавантажені тести зі створенням значної кількості віртуальних користувачів. Ці користувачі являють собою паралельні потоки, які симулюють навантаження на систему. Таким чином, при проведенні дослідження будуть братися заміри з різною кількістю віртуальних користувачів, симулюючи різні стресові рівні навантаження, починаючи з низького рівня, та закінчуючи значною кількістю паралельних потоків. На жаль через те, що симулювання користувачів відбуватиметься локально на тій самій машині, де розташована і досліджувана система, це буде мати вплив на продуктивність цієї системи, але при цьому сама відносна тенденція продуктивності роботи відносно контрольної системи все одно залишиться на тому ж рівні, так як обидві системи знаходяться в однакових умовах.

Під час тестування система буде проходити повний цикл своєї роботи, виконуючи реєстрацію користувача, після чого відбувається вхід. Далі

віртуальний користувач виконує доступу до захищеного ресурсу, після чого виконується запит на оновлення токенів доступу. В кінці життєвого циклу віртуальний користувач виконує вихід із системи. Схожий життєвий цикл має і контрольна система.

Ці та навантажувальні тести для наочності можна поділити на дві категорії — тести з низької інтенсивністю навантаження, а також з високою інтенсивністю. Відмінність полягатиме лише в кількості використовуваних одночасних віртуальних користувачів.

Отже почнемо тестування з оцінки часових витрат для повного виконання завдання системами. До цього показника входять усі часові витрати для виконання повного життєвого циклу, включаючи час, необхідний для запиту до бази даних, час для обробки та формування даних користувача у кодї системи, час для виконання HTTP-запитів. На рисунку 4.1 зображено діаграму витраченого часу на виконання одного повного циклу роботи програми.

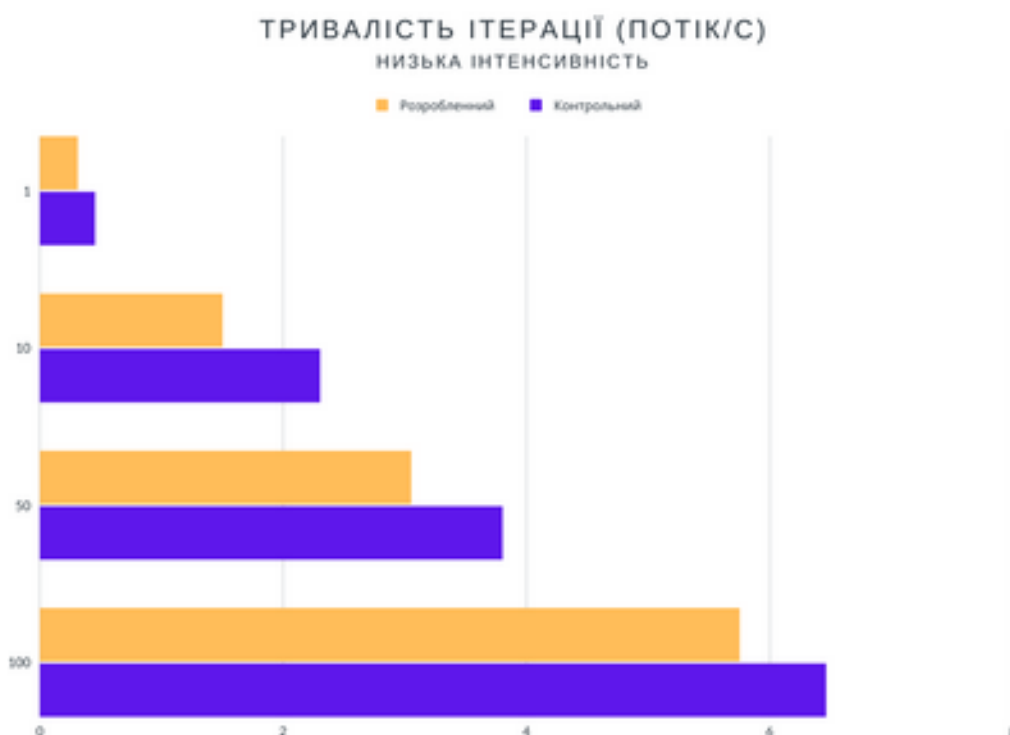


Рисунок 4.1 — Час одної повної ітерації при низькому навантаженні

Провівши порівняння отриманих даних можна зробити висновок, що при одному користувачу швидкість роботи вища на 31.3%, при десятих — 34.8%, але вже при п'ятдесяти розрив починає скорочуватися і різниця вже у 19,7%, а при сотні потоків різниця падає до 11%. Діаграма з відповідними показниками зображена на рисунку 4.2.

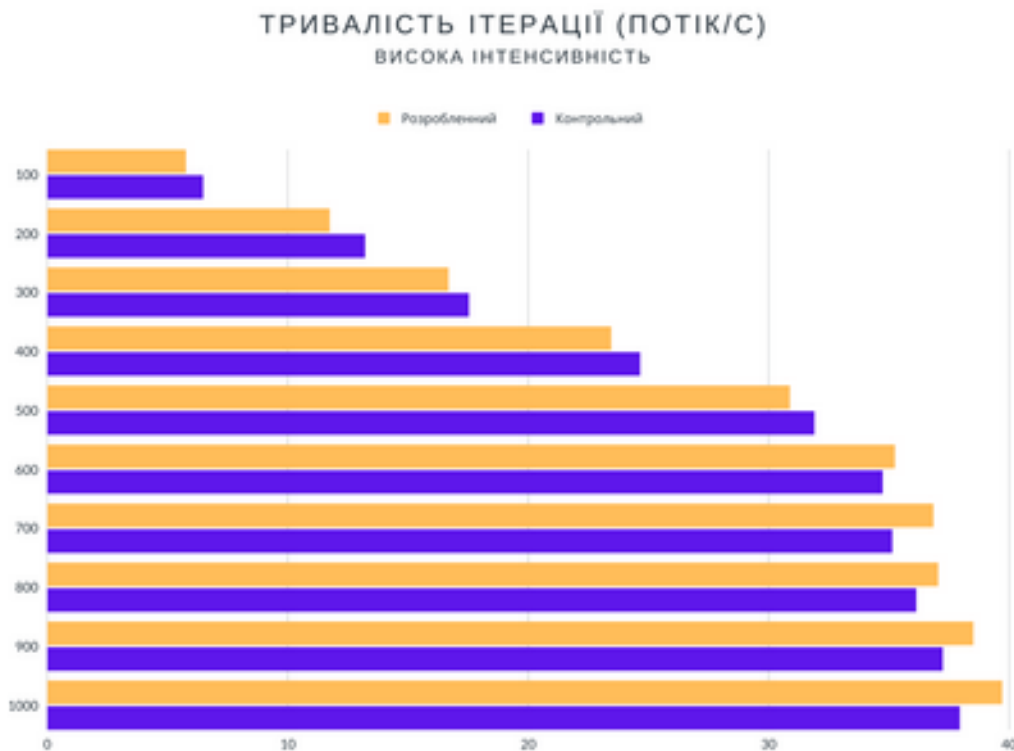


Рисунок 4.2 — Час одної повної ітерації при високому навантаженні

З цієї діаграми можна побачити, що при середніх показниках навантаження розроблена все ще показує кращий час роботи, але починаючи з шістсот потоків контрольна мікросервісна система починає працювати більш стабільно з меншими витратами часу. При цьому при двох сотнях віртуальних користувачів розроблена система все ще має перевагу у 11.1% від контрольної системи, але далі система поступово втрачає свою перевагу до рівня 4.85% при трьох сотнях користувачів та до -4.63% при одній тисячі.

Наступним кроком розглянемо дані, які відображають відносний час для виконання HTTP-запиту. Цей показник включає у себе час, який необхідно для відправлення запиту, час, який витрачається на очікування відповіді, а також час,

за який системою виконується отримання даних. Діаграма з цими показниками зображена на рисунку 4.3.

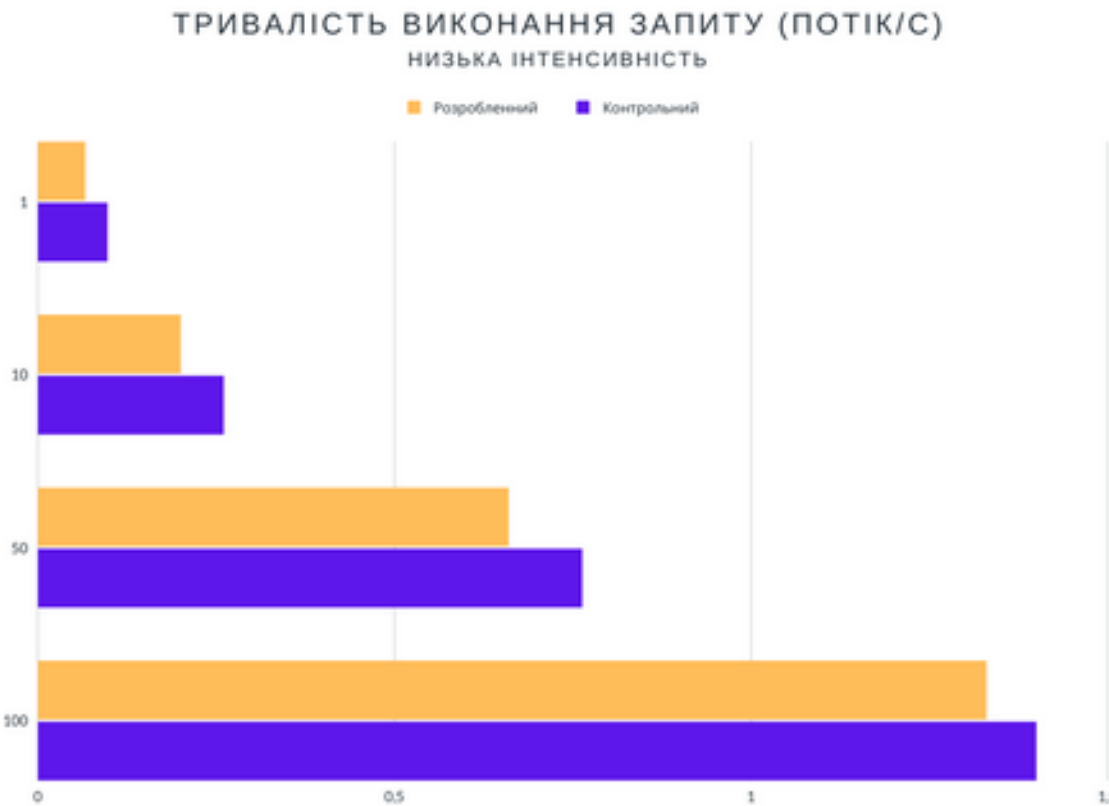


Рисунок 4.3 — Час виконання запиту при низькому навантаженні

Отриманий результат є дещо схожим на попередні показники часових витрат, але з меншою різницею. Так при одному потоці виконання запиту відбувається на 31.96% швидше, при десяти потоках — на 23%, при п'ятдесяти — 13.5%, а при сотні — 8.27%. Результат має схожу тенденцію при меншій різниці, що можна пояснити кращою оптимізацією саме процесів обробки запитів, ніж процесу передачі та приймання даних у мережі.

Наступним розглянемо час виконання запиту при високому навантаженні. Відповідна діаграма зображена на рисунку 4.4.



Рисунок 4.4 — Час виконання запиту при високому навантаженні

Отримані дані відображають схожу тенденцію з часом повної ітерації. Спочатку при середніх показниках часові витрати менші за показники контрольної системи, але зі збільшенням навантаження розроблена система знову ж таки починає відставати. Так при двох сотнях паралельних потоків різниця становить 7%, далі до шести сотень розроблена має незначну перевагу у близько 4%, після чого різниця йде у негативну сторону.

Кількість паралельно оброблюваних запитів у системі при цьому є приблизно схожою і становить в середньому для тестів з високим навантаженням близько 82.2%. Такий результат досягається завдяки особливостям роботи самої платформи, на який будуються обидві системи — Node.js. Насправді вона не підтримує паралелізму, а виконує усі запити асинхронно в одному потоці, за рахунок чого розподіляє ресурси серверу більш ефективно. Тому в рамках наявних навантажувальних тестів досягти значного навантаження саме на паралелізм не вдасться і системи працюють однаково добре.

Останнім вагомим фактором порівняння є використовувані ресурси системи. Контрольна система складається з багатьох модулів, які дозволяють значно ефективніше керувати їй вхідними запитами особливо при значних навантаженнях, але при цьому навіть при низькому навантаженні вона навантажує процесор в 2.6 рази сильніше за розроблену систему, в той час як використання пам'яті в неї на 37% нижче. Це відношення залишається однаковим незалежно від кількості потоків, а отже не залежить від рівня навантаження.

4.3 Висновки проведеного дослідження

За отриманими результатами можна зробити висновок, що розроблена система має значну перевагу над контрольною системою при невеликих навантаженнях — не більше двох сотень потоків, досягаючи в мінімумі переваги у 11%. При середніх навантаженнях від трьох до п'яти сотень розроблена система вже має незначну перевагу над контрольною системою, а далі при зростанні навантаження система починає показувати гірший результат за контрольну, де вже навпаки невелике відхилення буде в її користь. При цьому розроблена система використовує значно менше процесорних ресурсів, за рахунок чого є менш вимогливою до умов розгортання.

Для розуміння отриманого результату необхідно провести порівняння реалізованої та контрольної системи. Серед основних відмінностей можна виділити наступне:

- 1) Контрольна система використовує для спілкування протокол gRPC замість архітектурного стилю REST. gRPC — це фреймворк, який був розроблений для виклику віддалених процедур (RPC), і який працює поверх HTTP/2. У REST в якості формату передачі даних використовується JSON, тоді як у gRPC — Protobuf, який використовує бінарну серіалізацію даних, що значно підвищує швидкість роботи.

gRPC має багато переваг перед REST: у нього вище швидкість передачі даних за рахунок використання Protobuf, чи, наприклад, HTTP/2, який приносить додаткові переваги, такі як стиснення заголовків HTTP і мультиплексування запитів.

Але gRPC є і ряд недоліків. Через необхідність початкової додаткової ініціалізації цього протоколу, використання при низькій та середній кількості потоків не дає особливої переваги перед REST. Також gRPC є значно складнішим у реалізації та підтримці, через що підвищується вартість розробки. REST має більшу поширеність, завдяки чому можливе спілкування з зовнішнім API без зайвих проблем.

2) Використання асинхронного API. Синхронні запити блокують виконання запитів і можуть викликати серйозні вузькі місця у продуктивності. Асинхронний зв'язок стосується обміну даними між службами, що не потребують негайної відповіді. Завдяки протоколу gRPC у контрольній системі реалізується така потокова передача даних у реальному часі. Здатність gRPC керувати двонаправленою потоковою передачею дозволяє системі відправляти та отримувати повідомлення в режимі реального часу, не чекаючи відповіді клієнта, що дозволяє понизити час очікування.

3) В той самий час розроблена система для передачі даних між мікросервісами використовує внутрішній токен, в кому зберігаються дані. Хоча це дозволяє підвищити надійність системи завдяки верифікації даних, що передаються, та обмеженню доступу до внутрішніх мікросервісів з зовні, але через витрати часу, які необхідні для створення, а потім і читання токenu, зі значним підвищенням паралельних завдань обробки токenu зростає і затримка роботи. При високих навантаженнях використання механізмів безпеки, що передбачають велику кількість обмінів даними для перевірки справжності користувача, негативно позначається на продуктивності.

4) Методи кешування та стискання даних. Важливу роль відіграють і методи кешування, які дозволяються прискорити час отримання доступу до даних. Реалізація кешування відбувається на різних рівнях системи, наприклад на

рівні API-шлюзу, сервісах керування користувачами, сесіями, доступом, чи на рівні бази даних. Кешуванню піддаються усілякі дані, починаючи з відповідей користувачу та закінчуючи Access токенами. Разом з цим працюють і різні підходи до стискання даних, які відправляються на передачу. Процес компресії має багато особливостей реалізації, що значною мірою впливають на продуктивність системи, тому швидкодія великою мірою залежить від використовуваних підходів стискання даних. Так у розробленій системі використовується методи компресії з низькою оптимізацією, що в свою чергу впливає на результати у багатопотокових тестах навантаження.

З отриманих результатів можна зробити висновок, що найкращими умовами застосування розробленої мікросервісної системи є невеликі та невибагливі сервери з відносно низьким рівнем паралельного трафіку. В таких умовах запропонована конфігурація матиме найбільшу ефективність.

ВИСНОВКИ

В результаті виконання роботи було проведено дослідження технологій аутентифікації та авторизації для систем на базі мікросервісної архітектури. В ході роботи було спроектовано чотири алгоритми, на базі яких було розроблено мікросервісну систему.

В ході аналізу засобів розробки було проведено порівняльний аналіз мов програмування та систем управління базами даних. Також було проаналізовано існуючі моделі і методи реалізації аутентифікації та авторизації на базі мікросервісних систем і проведено аналіз існуючих на сьогоднішній день рішень з визначенням їх основних особливостей.

З результатів аналізу було визначено ключові алгоритми для аутентифікації і авторизації у мікросервісних системах, та проведено проектування покращених варіантів цих алгоритмів з урахуванням підвищення їх безпеки та ефективності роботи. Таким чином було спроектовано алгоритм роботи API-шлюзу, алгоритм створення користувача, алгоритм створення сесії користувача, алгоритм видалення сесії користувача, алгоритм оновлення токенів сесії користувача, алгоритм авторизації та визначено загальну схему розроблюваної системи.

За спроектованими алгоритмами було розроблено систему, працездатність якої було перевірено за допомогою середовищ Postman і К6. Була перевірена коректність роботи основних компонентів системи та правильність взаємодії сервісів між собою та користувачем. Було визначено повну працездатність розробленої системи.

В ході дослідження отриманої системи було визначено ефективність її роботи відносно контрольної системи. Було визначено ефективні умови застосування, за яких система буде мати найвищу продуктивність своєї роботи. Так система показала себе в середньому на 20% ефективніше при значеннях кількості паралельних потоків до двох сотень одиниць, на 5% при значеннях до п'яти сотень, але при подальшому зростанні навантаження система має нижчу

ефективність в середньому на 4% від контрольної системи. При цьому розроблені алгоритми показують себе значно ефективніше у використанні процесорних потужностей у 2.6 рази, хоча використання пам'яті у системи на 37% вище.

Виходячи з отриманих результатів з дослідження методів аутентифікації та авторизації в мікросервісних системах, можна підсумувати та зазначити, що поставлена мета магістерської роботи з покращення існуючих підходів була успішно досягнена.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

- 1 Microservices Adoption in 2020 by Mike Loukides and Steve Swoyer [Електронний ресурс] – Режим доступу до ресурсу: <https://www.oreilly.com/radar/microservices-adoption-in-2020/>.
- 2 OWASP API Security TOP 10 [Електронний ресурс] – Режим доступу до ресурсу: <https://owasp.org/www-project-api-security/>.
- 3 Microservices vs. monolithic architecture [Електронний ресурс] – Режим доступу до ресурсу: <https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith>.
- 4 Authentication & Authorization in Microservices Architecture - Part I [Електронний ресурс] – Режим доступу до ресурсу: <https://dev.to/behalf/authentication-authorization-in-microservices-architecture-part-i-2cn0>.
- 5 Мікросервісна архітектура [Електронний ресурс] – Режим доступу до ресурсу: <https://medium.com/@IvanZmerzlyi/microservices-architecture-461687045b3d>.
- 6 Security Strategies for Microservices-based Application Systems (PDF) [Електронний ресурс] – Режим доступу до ресурсу: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204.pdf>.
- 7 XACML [Електронний ресурс] – Режим доступу до ресурсу: <https://uk.wikipedia.org/wiki/XACML>.
- 8 Authentication and authorization in microservice-based systems: survey of architecture patterns (PDF) [Електронний ресурс] – Режим доступу до ресурсу: https://cyberrus.com/wp-content/uploads/2020/10/32-43-438-20_4.Barabanov.pdf.
- 9 PolyglotPersistence [Електронний ресурс] – Режим доступу до ресурсу: <https://martinfowler.com/bliki/PolyglotPersistence.html>.
- 10 Microservices [Електронний ресурс] – Режим доступу до ресурсу: <https://www.jetbrains.com/lp/devecosystem-2021/microservices/>.

11 Pluggable Authentication Modules (PAM) [Электронный ресурс] – Режим доступа до ресурсу: <https://www.oreilly.com/library/view/practical-unix-and/0596003234/ch04s05.html>.

12 Microservices Authentication and Authorization Solutions [Электронный ресурс] – Режим доступа до ресурсу: <https://medium.com/tech-tajawal/microservice-authentication-and-authorization-solutions-e0e5e74b248a>.

13 RBAC vs. ABAC: What's the Difference? [Электронный ресурс] – Режим доступа до ресурсу: <https://www.dnsstuff.com/rbac-vs-abac-access-control>.

14 Современные стандарты идентификации: OAuth 2.0, OpenID Connect, WebAuthn [Электронный ресурс] – Режим доступа до ресурсу: <https://habr.com/ru/post/491116/>.

15 KONG — The Microservice API Gateway [Электронный ресурс] – Режим доступа до ресурсу: <https://medium.com/@far3ns/kong-the-microservice-api-gateway-526c4ca0cfa6>.

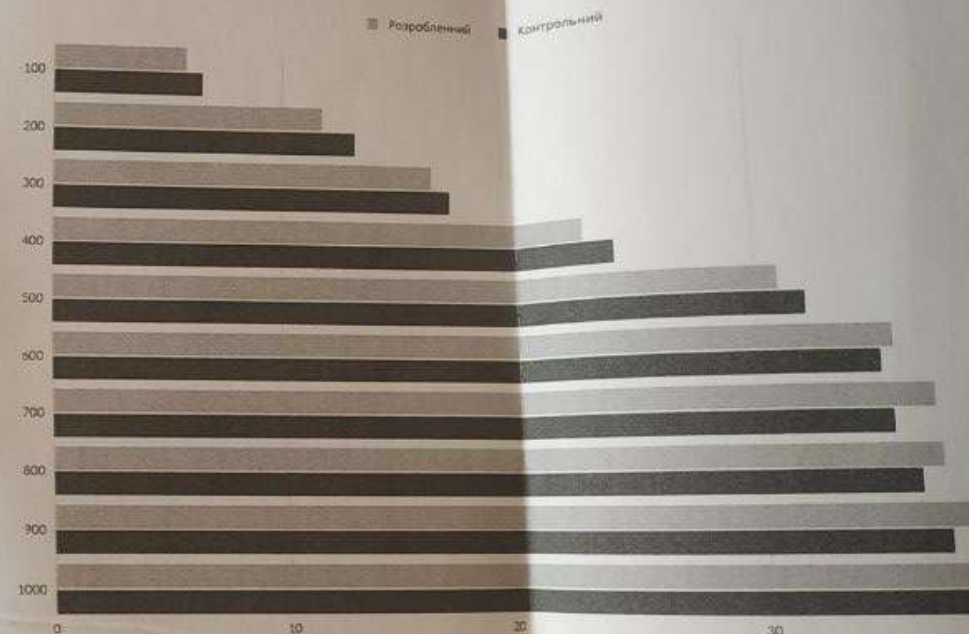
16 What is Amazon API Gateway? [Электронный ресурс] – Режим доступа до ресурсу: <https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html/>.

17 Tyk API Gateway [Электронный ресурс] – Режим доступа до ресурсу: <https://github.com/TykTechnologies/tyk>.

■ Разработанный ■ Контрольный



Розроблений



					13.02070849.00007 ПЛ2	
					Комп'ютерна система аутентифікації користувачів на основі мікросервісної архітектури	
					Діаграми дослідження витрат часу на виконання повної ітерації роботи системи	
Зам.	Лист	№ докум.	Піпн.	Дата	Лист	Місяц
Розроб.		Кострикин М.М.	<i>[Signature]</i>			
Перев.		Серульський С.Ю.	<i>[Signature]</i>			
Т. контр.					Лист	
Н. контр.		Щербак Н.В.	<i>[Signature]</i>		Национальный университет «Запорізька політехніка»	
Затв.		Кудерметов Р.К.	<i>[Signature]</i>		КНТ-31	

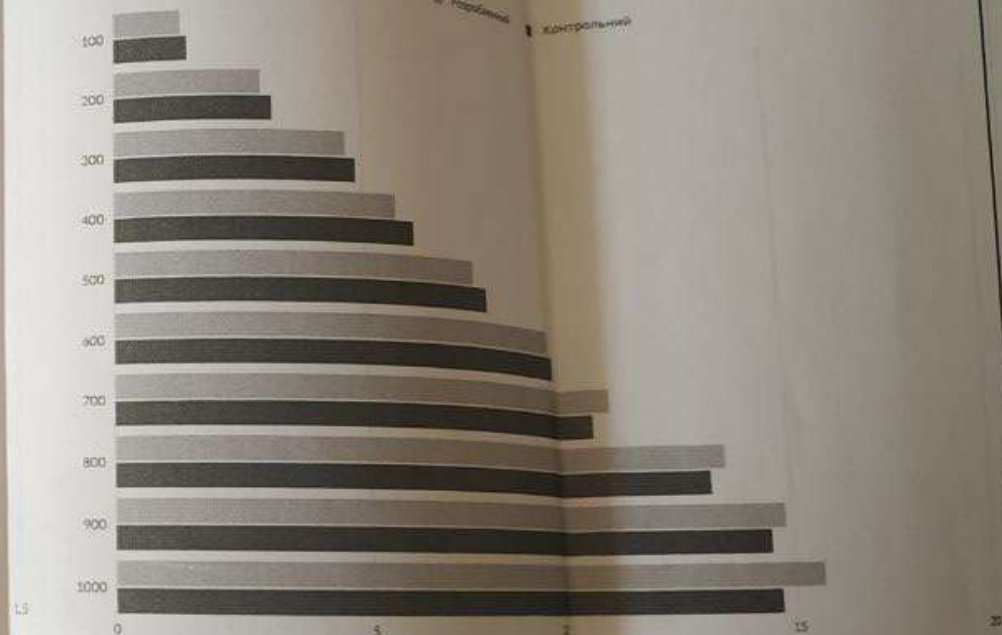
ТРИВАЛІСТЬ ВИКОНАННЯ ЗАПИТУ (ПОТІК/С) НИЗЬКА ІНТЕНСИВНІСТЬ

Розроблений Контрольний



ТРИВАЛІСТЬ ВИКОНАННЯ ЗАПИТУ (ПОТІК/С) ВИСОКА ІНТЕНСИВНІСТЬ

Розроблений Контрольний



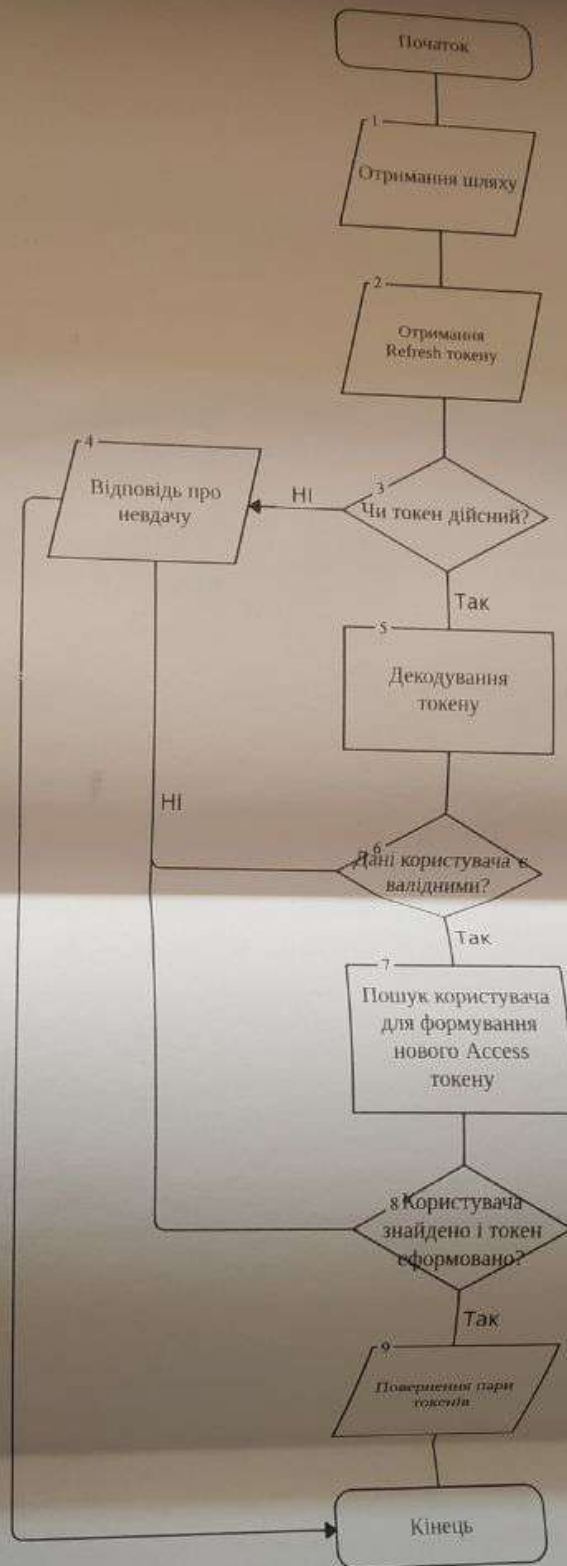
13 02070849.00007 ПЛЗ

Зам.	Лист.	№ докум.	Пит.	Зам.
Розроб.		Кострикин М.М.		
Перев.		Скрупський С.Ю.		
Т. контр.				
Н. контр.		Шербак Н.В.		
Зам.		Кудерметов Р.К.		

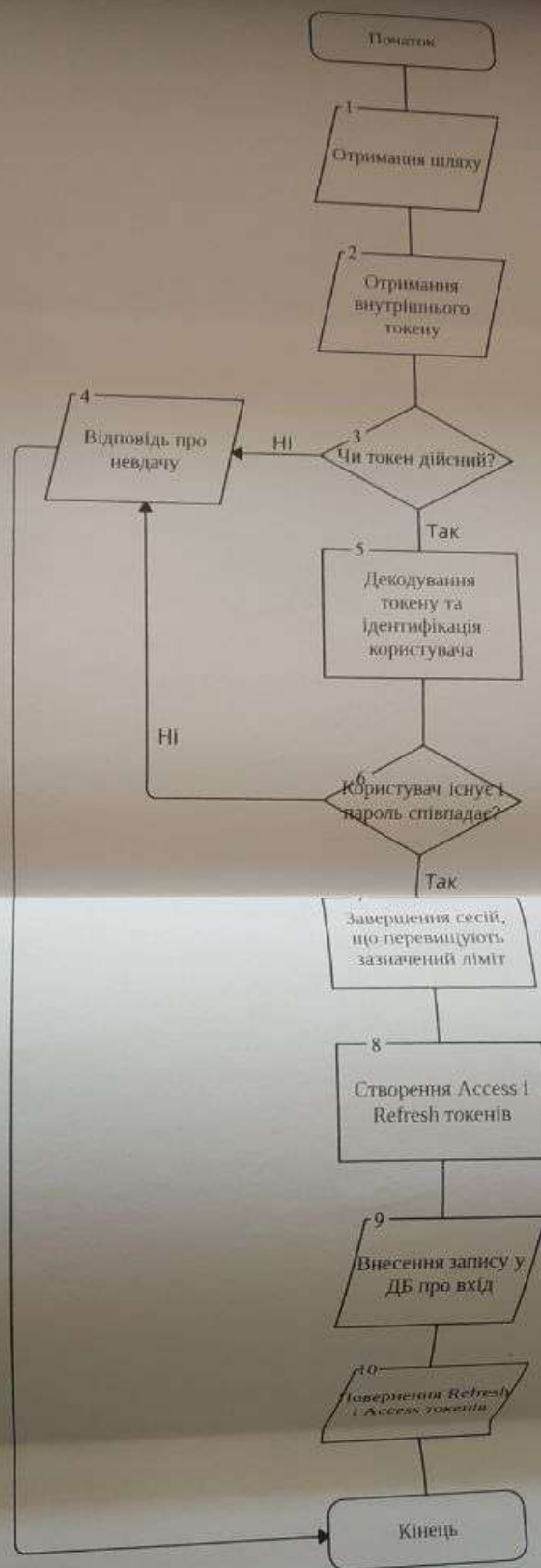
Комп'ютерна система аутентифікації користувачів на основі мікросервісної архітектури
Діаграма дослідження витрат часу на виконання запитів системою

Лист	Місяц	Місяць

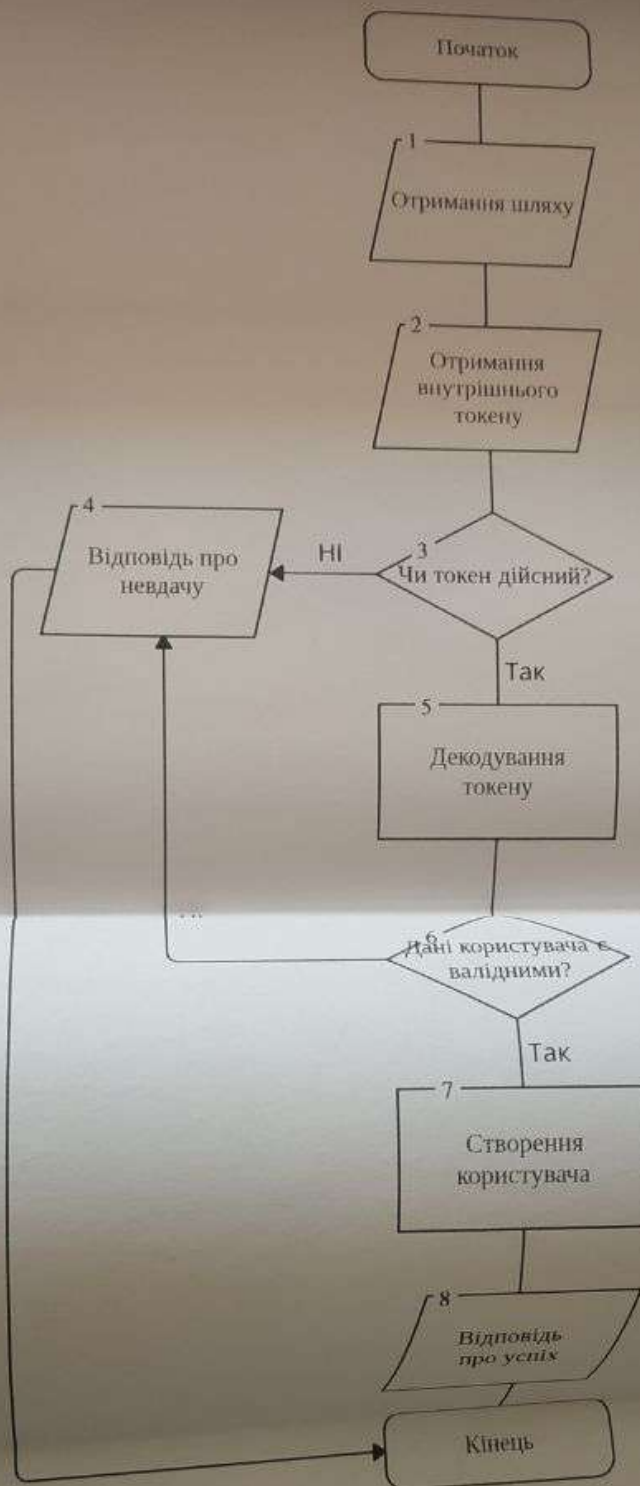
Лист Лист
Національний університет «Запорізька політехніка»
КНТ-5114



13.02070849.00007 A1				
Зам.	Лист	Аб докум.	Підп.	Датум
Розроб.	Костриків М.М.			
Перев.	Скороходов С.Ю.			
Т. контр.				
Н. контр.	Щербак П.В.			
Затв.	Кудаченко Р.Б.			
Комп'ютерна система аутентифікації користувачів на основі мікросервісної архітектури				
Алгоритм оновлення токена сесії				
Лист				
Місяц				
Рік				
Національний університет «Запорізька політехніка»				
КНТ-511м				



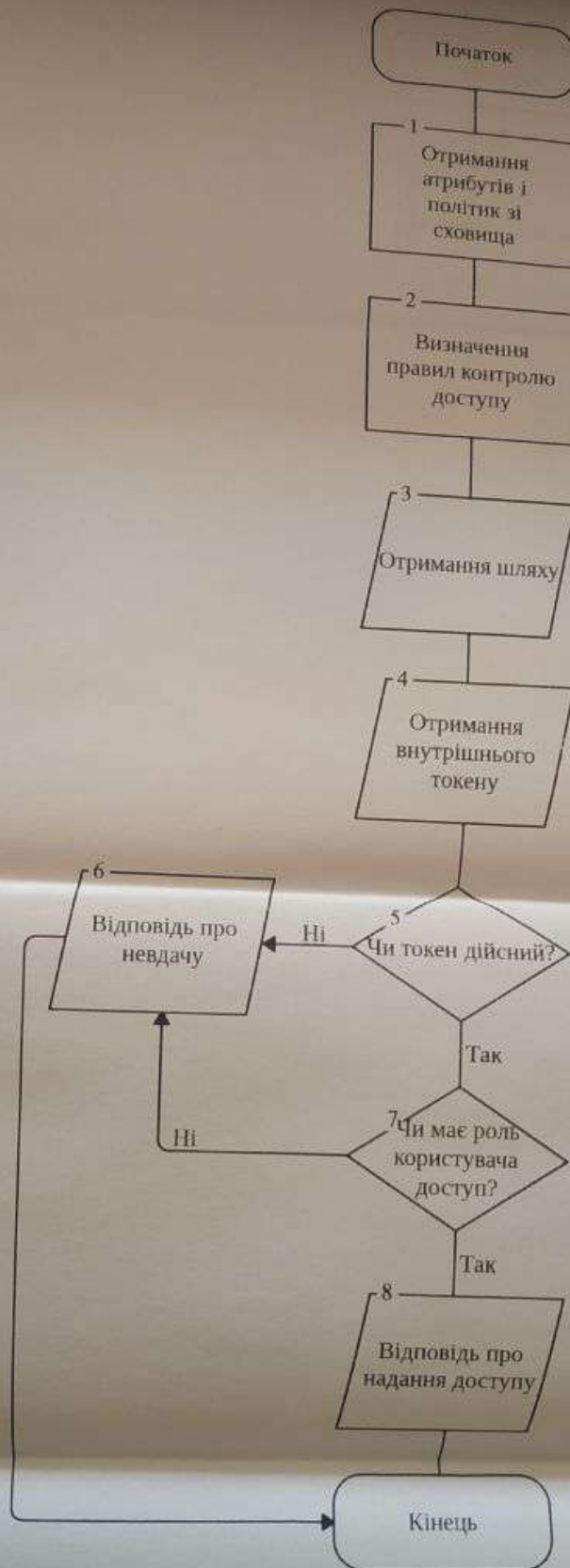
13.02070849.00007 A1				
Зам.	Лист	№ докум.	Підп.	Дата
Розроб.		Костришин М.М.	<i>[Signature]</i>	
Перев.		Скрупський С.Ю.	<i>[Signature]</i>	
Головир.				
Н.контр.		Щербак Н.В.	<i>[Signature]</i>	
Зам.		Худяков Р.Р.	<i>[Signature]</i>	
Комп'ютерна система аутентифікації користувачів на основі мікросервісної архітектури				
Алгоритм створення сесії				
Лист	Маса	Масив		
Листа				
Національний університет «Запорізька політехніка»				
КНТ-511М				



13.02070849.00007 A1				
Комп'ютерна система аутентифікації користувачів на основі мікросервісної архітектури				
Алгоритм створення користувача				
Лист				
Маса				
Масштаб				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				
Лист				



13.02070849.00007 A1					Лист	Маса	Масштаб
Зам.	Лист.	М. докум.	Шлях	Лист	Комп'ютерна система аутентифікації користувача на основі мікросервісної архітектури		
Розроб.	Костришин М.М.						
Перев.	Скрупський С.Ю.				Алгоритм видалення сесії		
Т.контр.							
Н.контр.	Шербак Н.В.				Національний університет «Запорізька політехніка» КНТ-511м		
Затв.	Кудерметов Р.К.						



					13.02070849.00007 A1					
Зам.	Лист	№ докум.	Підп.	Дата	Комп'ютерна система аутентифікації користувачів на основі мікросервісної архітектури			Лист	Маса	Масштаб
Розроб.		Богданюк М.М.			Алгоритм перевірки прав доступу					
Перев.		Скрявський С.Ю.								
Г.контр.								Національний університет «Запорізька політехніка»		
Н.контр.		Шарбон Н.В.						КІТ-511м		
Чит.		Будаківський Р.Б.								