Міністерство освіти і науки України
Харківський національний університет ім. В. Н. Каразіна
Факультет комп'ютерних наук
Спеціальність 125 «Кібербезпека»
Освітня програма «Кібербезпека»

"Допущено до захисту"
В.о. завідувача кафедри БІСТ
Мелкозьорова О.М.

_____
«_____» 2024 р.

Пояснювальна записка

до кваліфікаційної роботи бакалавра

На тему "Розробка засобу безпечної аутентифікації користувачів у розподіленій інформаційній системі"

оцінка « » Керівник проф. Олійников Р.В.

Голова ЕК Рецензент доцент Родінко М.Ю.

Лемешко О.В. _____ Виконавець: студент групи КБ-42

_____ Косаренко І.В.

Харків - 2024

РЕФЕРАТ

Пояснювальна записка включає в себе 60 сторінок, 20 рисунків, 26 джерел, 1 таблиця, 2 додатки.

Метою роботи є розробка безпечного механізму аутентифікації користувачів у розподіленій системі на основі сучасних методів забезпечення захисту, включаючи ідентифікацію, аутентифікацію та авторизацію.

Об’єктом дослідження є процеси та методи аутентифікації користувачів у розподіленій системі.

Предметом дослідження є новітня інформаційно-комунікаційна система з методами аутентифікації та авторизації.

Дипломна робота містить в собі теоретичні відомості про розподілені системи, їх принципи та методи використання, мікросервісна архітектура, як реалізізація розподіленої системи. Було висвітлено інформацію про основні властивості мікросервісів, їх компоненти та особливості. Також робота містить в собі основні принципи безпеки користувачів у розподіленій системі, а саме принципи роботи ідентифікації, аутентифікації та авторизації. В дипломній роботі було розглянуто сучасний протокол безпеки OAuth 2.0, що є популярним рішенням для забезпечення безпеки в інформаційних системах.

У процесі роботи було розроблено додаток на основі мікросервісної архітектури використання протоколу аутентифікації та авторизації OAuth 2.0. Під час виконання практичної частини дипломної роботи, було використано сучасні технології та принципи побудови інформаційно-комунікаційної системи.

Ключові слова: ЗАСТОСУНОК, ЗАХИСТ, БЕЗПЕКА, МЕТОДИ БЕЗПЕКИ, ІДЕНТИФІКАЦІЯ, АУТЕНТИФІКАЦІЯ, АВТОРИЗАЦІЯ.

ABSTRACT

The explanatory note includes 60 pages, 20 figures, 26 sources, 1 table, and 2 appendices.

The aim of the study is to develop a secure user authentication mechanism in a distributed system based on modern security methods, including identification, authentication, and authorization.

The object of the research is the processes and methods of user authentication in a distributed system.

The subject of the research is the latest information and communication system with authentication and authorization methods.The thesis contains theoretical information about distributed systems, their principles, and methods of use, and microservice architecture as an implementation of a distributed system. It highlights the main properties of microservices, their components, and features. The work also includes the basic principles of user security in a distributed system, specifically the principles of identification, authentication, and authorization. The thesis examines the modern OAuth 2.0 security protocol, which is a popular solution for ensuring security in information systems.

During the course of the work, an application based on microservice architecture utilizing the OAuth 2.0 authentication and authorization protocol was developed. The practical part of the thesis employed modern technologies and principles for building an information and communication system.

Keywords: APPLICATION, PROTECTION, SECURITY, SECURITY METHODS, IDENTIFICATION, AUTHENTICATION, AUTHORIZATION.

ЗМІСТ

ПЕРЕЛІК ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

ІЗОД – Інформація з Обмеженим Доступом
ІКС – Інформаційно Комунікаційна Система
СКБД – Система Керувань Базою Даних
API – Application Programming Interface
CI / CD – Continious Integration / Continious Deployment
DDL – Data Definition Language
DevOps – Development Operations
DOS – Denial Of Service
gRPC – Google Remote Procedure Call
IPC – Inter-Process Communication
JWT – JSON Web Token
MFA – Multi Factor Authentication
RBAC – Role Based Access Control
REST – Representative State
SIEM – Security Information and Event Managment
SQL – Structured Query Language
URL – Uniform Resource Locator
XML – EXtensible Markup Language
YAML – Yet Another Markup Language

ВСТУП

Актуальність. Тема безпеки користувачів у інформаційній системі залишається дуже актуальною в сучасному цифровому світі. Цінність інформації стає важливим фактором у будь-якій організації, тому захист особистої інформації та даних користувачів стає все важливішим.

З появою глобального Інтернету, виникла велика кількість кількість кібератак, включаючи хакерські атаки, проникнення в систему, DOS-атаки, перехоплення каналів зв'язку. Це створює небезпеку для ІЗОД, тобто право доступу до якої обмежено встановленими правовими нормами і (чи) правилами [1]. Порушення цілісності може спричинити дуже великі економічні втрати для власників інформаційної системи. Витік або порушення цілісності ІзОД (спотворення, модифікація, руйнування, знищення) можуть бути результатом реалізації загроз безпеці [2].

На сьогоднішній день суспільство стає обізнаніше у темі кібербезпеки. Коли людина може втратити цінну інформацію, приходе усвідомлення про ризики, які пов'язані зі збереженням та передачею особистих даних у мережі. Це змушує створювати нові ідеї та підходи кіберзахисту даних, тому що вимоги зростають пропорційно кількості користувачів в глобально-інформаційному середовищі.

Не мало людей користується інтернетом, соціальними мережами, веб-сервісами, хмарними технологіями, Інтернетом речей та іншою інформаційною інфраструктурою. Без надійного захисту та ефективної інформаційної системи особисті дані мають великий ризик бути скомпрометовані зловмисниками, що призводить до поганих наслідків суспільству.

Наразі все більше і більше ресурсів вкладаються в розвиток кібербезпеки. Захист інформації з обмеженим доступом став пріорітетною задачею багатьох країн та суспільств. Наразі постійно створюються нові стандарти та регламенти захисту персональних даних, а також покращуються підходи до

кіберзахисту, отримується новий досвід, суспільство отримує позитивні експертні висновки, які дозволяють використовувати вже опрацьовану та надійну систему захисту.

Сучасні інформаційно-комунікаційні системи представляють собою велику мережу телекомунікацій та комп'ютерної інфраструктури, що об'єднані між собою. Це великий зв'язок між елементами, що дозволяє обмінюватись інформацією на будь-який куток світу. Проте, це викликає багато вразливостей у кіберсередовищі. Зловмисники мають дуже багато можливостей провести кібератаку та захопити конфіденційну інформацію.

У епоху глобального інтернету та розподілених систем суспільство повинно надавати велику увагу безпеці та захисту особистої інформації. Зростання кількості цифрових сервісів і платформ, що збирають і обробляють особисті дані користувачів, робить безпеку ще більш важливою. Нестача адекватного захисту може призвести до серйозних наслідків, таких як крадіжка ідентифікаційної інформації, фінансові шахрайства, або навіть порушення приватності.

У цій дипломній роботі увага приділяється темі безпеки користувачів у розподіленій системі. Основною задачею є розробка безпечної розподіленої системи на основі сучасних методів аутентифікації та авторизації.

Також у роботі особлива увага приділяється ідентифікації, аутентифікації та авторизації користувачів у системі. Розглядається перспективи розвитку розподіленої системи та основні проблеми безпеки у ній. Також буде запропоновано практичне рішення на базі проаналізованих цілей та вимог.

Для досягнення мети дипломної роботи було сформовано наступні завдання:

- дослідити актуальність та розвиток розподілених систем у сучасному світі
- проаналізувати теоретичні дані про безпеку конфіденційної інформації та засоби її захисту у розподілених системах, включаючи інформацію

про основі принципи безпеки даних у інформаційно-комунікаційних системах, сучасні методи аутентифікації та авторизації

- розробити захищену систему, спираючись на теоретичні дані та вимоги до системи

1 АКТУАЛЬНИЙ СТАН І ПЕРСПЕКТИВИ РОЗВИТКУ РОЗПОДІЛЕНИХ СИСТЕМ НА БАЗІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

1.1 Огляд сучасного стану розподілених систем

Комп’ютерні системи переживають велику революцію. З 1945 року, коли почалася комп’ютерна ера, приблизно до 1985 року комп’ютери були великими та дорогими. Навіть міні-комп’ютери коштували щонайменше десятки тисяч доларів кожен. В результаті більшість організації мали лише кілька комп’ютерів. Через відсутність шляхів для підключення, вони працювали окремо один від одного [3].

Через деякий час, комп’ютери почали набувати популярність. Приток ресурсів та уваги до тоді незнайомої технології вплинуло на безпрецедентний розвиток комп’ютерів. З середини 1980-х років виникло два визначних відкриття. Перший прорив у інформаційних технологіях було створення потужних мікропроцесорів. Спочатку це були 8-бітні, а згодом 16 та 32-бітні, що дозволяли виконувати понад 1 млрд. інструкцій в секунду. Також впала і вартість комп’ютерів, що дозволило швидко разповсюдити їх у кожну організацію та кожен дім.

Друге значне відкриття стосується мережевих технологій. Локальні мережі дозволили об’єднувати десятки та сотні комп’ютерів та обмінюватись величезними обсягами даних за секунди. З часом локальні мережі перетворились в глобальні та дозволили комп’ютерам комунікувати між країнами та регіонами.

За результатами цих технологій з’явилась можливість з легкістю та великою швидкістю обмінюватись інформацією між комп’ютерами. Розподілені системи – це комп’ютерні системи, що складаються з набору взаємопов’язаних комп’ютерів, які спільно працюють для досягнення спільної мети. Ці системи можуть бути фізично розташовані на різних вузлах мережі та мати різні характеристики, такі як операційна система, обладнання тощо.

Основна ідея розподілених систем полягає у тому, що вони спільно працюють над вирішенням задачі, розділяючи навантаження та ресурси між собою. Визначення розподілених систем має два основних аспекти: система повинна складатись з автономних компонентів (комп'ютерів) та користувачі розподіленої системи не повинні знати, що система багатокомпонента, тобто користується, як одна система. Це означає, що для користувачів важливо забезпечити враження єдиної системи, незважаючи на те, що вона може бути розгорнута на різних пристроях чи комп'ютерах, що працюють у різних мережах.

Перший аспект, про автономність компонентів системи, означає, що кожен компонент може працювати незалежно від інших, має власні ресурси та може функціонувати відокремлено. Це дозволяє системі бути більш масштабованою та ефективною, адже різні компоненти можуть працювати паралельно, обробляючи різні завдання.

Другий аспект полягає в тому, що користувачі не повинні знати про розподіленість системи. Для них важливо мати доступ до функцій і можливостей системи, незалежно від того, на яких пристроях вони працюють чи з якої точки доступу вони підключені до мережі. Це забезпечує єдність і зручність взаємодії з системою для користувачів, що може бути критично важливим для успішного впровадження розподілених систем у практичних застосуваннях.

Для кращого розуміння принципу роботи розподіленої системи, важливо відокремити її характеристики. Однією з важливих характеристик є відмінності між різними комп'ютерами в одній системі та способи як вони комунікують, як правило, приховано від користувачів. Те саме стосується внутрішньої організації розподіленої системи. Ще одна важлива характеристика полягає в тому, що користувачі та програми можуть взаємодіяти з розподіленою системою узгоджено однаковим способом, незалежно від того, де і коли відбувається взаємодія.

В принципі, розподілені системи також мають відносно легко розширюватися або масштабуватись. Ця характеристика є прямим наслідком наявності незалежних комп'ютерів, але водночас приховує, як ці комп'ютери насправді беруть участь у системі в цілому [3, с. 2].

1.2 Перспективи напряму розвитку розподілених систем

Розвиток технологій розподілених систем сьогодні визначає не лише подальше вдосконалення інформаційного середовища, але й трансформацію бізнес-моделей та підвищення продуктивності в різних галузях. Новітні технології, такі як хмарні рішення, контейнеризація та мікросервісна архітектура, дозволяють компаніям створювати динамічні та адаптивні системи, які забезпечують швидкість реакції на зміни на ринку. Використання таких технологій веде до збільшення ефективності використання ресурсів, оптимізації робочих процесів та поліпшення якості обслуговування клієнтів. Безпека також стає пріоритетним аспектом у розробці розподілених систем, і сучасні інструменти та підходи дозволяють створювати відмовостійкі та захищені від кіберзагроз рішення. Таким чином, інноваційні технології у сфері розподілених систем відкривають нові можливості для підприємств у всьому світі, допомагаючи їм ефективно використовувати ресурси, швидше реагувати на зміни та залишатися конкурентоспроможними.

Однією з ключових інновацій у сфері розподілених систем є мікросервісна архітектура. У порівнянні з традиційними монолітними додатками, мікросервісна архітектура дозволяє розбити додаток на невеликі, незалежні сервіси, які можуть функціонувати та масштабуватися незалежно один від одного. Це створює більш гнучкі та розширювані системи, оскільки кожен мікросервіс може бути розроблений, впроваджений та масштабований окремо. З точки зору розподіленої системи, мікросервіси можуть бути розгорнуті на різних серверах чи контейнерах, що дозволяє створювати гнучкі та відмовостійкі архітектури. Крім того, мікросервіси можуть комунікувати між собою за допомогою API, що дозволяє легко інтегрувати нові функції та забезпечує більшу модульність системи. Таким чином, мікросервісна

архітектура сприяє подальшому розвитку розподілених систем, забезпечуючи більшу гнучкість, масштабованість та відмовостійкість.

Особлива увага приділяється саме у розвиток способів та методологій безпеки розподілених систем. Разом із стрімким прогресом у технологіях та парадигмах, відповідні небезпеки умисних та неумисних порушень безпеки збільшилися експоненційно, настільки, що серед більш нових тенденцій у розподіленому обчисленні безпека стала стримуючим фактором для його прийняття [4].

1.3 Проблемні питання розвитку безпечних розподілених систем на базі мікросервісної архітектури

Розподілені системи, побудовані на базі мікросервісної архітектури, стають все більш популярними у сучасному інформаційному середовищі, проте вони також стикаються з рядом викликів та проблем, які потрібно вирішувати для забезпечення їх ефективності та безпеки. Потреба захищати цілісність та конфіденційність інформації та інших ресурсів, які належать як індивідуумам, так і організаціям, є всеосяжною як у фізичному, так і в цифровому світі [5].

Один із головних викликів – це забезпечення безпеки та відсутність точок вразливості в розподіленій системі. З урахуванням того, що мікросервіси функціонують автономно, інтеграція різних сервісів може створювати додаткові потенційні точки атаки для зловмисників. Тому важливо розробляти та впроваджувати ефективні механізми захисту, такі як мережеві файрволи, системи моніторингу безпеки та ідентифікації, щоб мінімізувати ризик порушення безпеки даних та системи в цілому.

Сама основна модель безпеки розподіленої системи формулюється так: безпека розподіленої системи може бути досягнута шляхом захисту процесів і каналів, які використовуються для їх взаємодії та захисту об’єктів, які вони інкапсулюють від несанкціонованого доступу. Об’єкти у системі маються на увазі будь-яка інформація з обмеженим доступом, наприклад, конфіденційна, яка використовується різними користувачами різними способами. Наприклад, деякі об’єкти можуть містити особисті дані користувача, такі як його поштова

скринька, номер телефону, дата народження, податковий номер та інші. Щоб це підтримувати, потрібно мати права доступу, які вказують на рівень доступу, щоб виконувати ті чи інші операції.

Ще однією проблемою є масштабованість та управління розподіленими системами. Зі зростанням кількості мікросервісів у системі стає складніше відслідковувати та керувати ними. Важливо мати ефективні засоби моніторингу та управління, які дозволяють оперативно виявляти та усувати неполадки, а також автоматизувати процеси розгортання та масштабування мікросервісів для забезпечення ефективного використання ресурсів.

Не менш важливим викликом є забезпечення високої доступності та надійності системи. Розподілені системи мають бути готові до різних видів відмов та забезпечувати безперервну доступність сервісів для користувачів. Для цього потрібно впроваджувати механізми реплікації даних, автоматичного відновлення та балансування навантаження, що дозволяє забезпечити стабільну роботу системи навіть у випадку відмов окремих компонентів.

Розробка та впровадження ефективних рішень для подолання цих викликів є ключовими завданнями для подальшого розвитку безпечних розподілених систем на базі мікросервісної архітектури. Вирішення цих проблем відкриватиме нові можливості для застосування.

1.4 Постановка задач досліджень

Для подолання мети та виконання усіх задач дипломної роботи, потрібно сформулювати задачи, які будуть описувати проблеми безпеки розробки засобу безпечної аутентифікації користувачів у розподіленій інформаційній системі. Виходячи зі сформованих проблем, зазначених у минулому підрозділі потрібно:

- Проаналізувати основні концепції та принципи побудови мікросервісної архітектури, їх взаємодія із компонентами всередині системи
- Визначити вимоги щодо безпеки і надійності мікросервісних архітектур, описати основні методи та засоби захисту користувачів, дослідити такі поняття, як: ідентифікація, аутентифікація, авторизація

- Розробити архітектурну модель розподіленої системи з відповідними вимогами та поставленими задачами, в яку входить способи збереження інформації, способи забезпечення консистентності даних. Створити програмне забезпечення на базі мікросервісної архітектури.

2 ОПИС МІКРОСЕРВІСНИХ АРХІТЕКТУР

2.1 Основні принципи мікросервісної архітектури

Концепція мікросервісної архітектури сформувалася як відповідь на зростання складності традиційних монолітних додатків у великих корпоративних середовищах. Перші застосування подібних ідей виникли в середині 2000-х років, коли компанії шукали способи розробки, що дозволяли б швидше реагувати на зміни в бізнес-вимогах та підвищувати ефективність розробки програмного забезпечення.

Мікросервісна архітектура – це підхід до розробки програмного забезпечення, де програми представляються як колекція сервісів, які незалежні один від одного та слабко зв’язані [6]. Щоб краще зрозуміти суть цієї технології, потрібно відрізняти монолітну архітектуру та мікросервісну.

Монолітна архітектура — це традиційна модель програмного забезпечення, яка побудована як уніфікована одиниця, самодостатня та незалежна від інших програм. Слово «моноліт» часто приписують чомусь великому та значному, що не далеко від істини монолітної архітектури для розробки програмного забезпечення [7]. Монолітна архітектура — це окрема велика обчислювальна мережа з єдиною кодовою базою, яка об’єднує всі бізнес-завдання. Щоб внести зміни в програму такого типу, потрібно оновити увесь додаток, отримавши доступ до бази коду та створивши та розгорнувши оновлену версію інтерфейсу служби. Це робить оновлення обмежувальними та забирає багато часу (див. рис. 2.1).

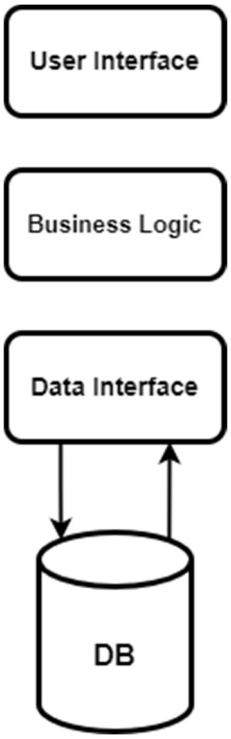

Рисунок 2.1 – схема монолітної архітектури

Головна різниця між мікросервісною та монолітною архітектурою – способ організації та взаємодії компонентів програмного забезпечення. Мікросервіси надають більшу гнучкість, масштабованість та незалежність, що робить їх популярним вибором для розробки сучасних розподілених систем. У той час, як монолітні додатки можуть бути більш простими для розробки та розгортання в невеликих проектах. На рисунку 2.2 зображено схема компонентів мікросервісної системи.

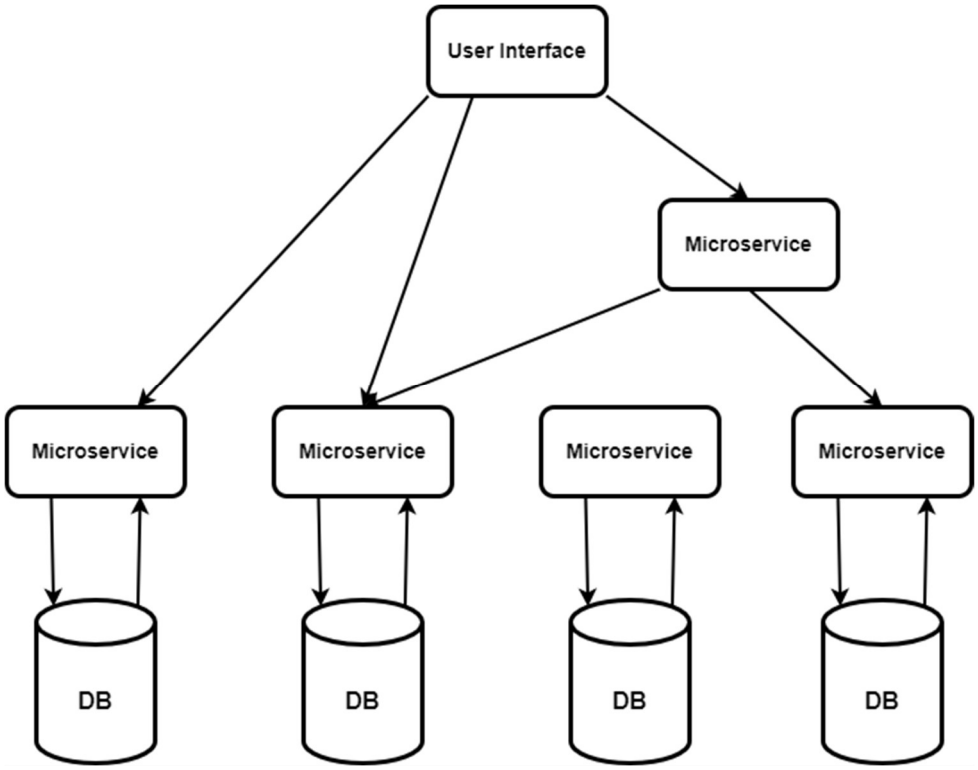

Рисунок 2.2 – схема мікросервісної архітектури

Мікросервісна архітектура має велику кількість переваг, що робить монолітну систему менш надійною, ефективною та безпечною. Серед розподілених систем, мікросервісна є самою найразповсюдженою та їй властиві найбільшою мірою переваги технології розподілених систем та клієнто-орієнтовних систем.

Різноманітність технологій. Маючи систему, що складається з кількох взаємодіючих служб, ми можемо вирішити використовувати різні технології всередині кожного. Це дозволяє нам вибрати правильний інструмент для кожної роботи, а не доводиться вибирати більш стандартизований, універсальний підхід, який часто не може ефективно розкрити увесь потенціал ресурсів компонента [8, с. 20].

Якщо одна частина нашої системи потребує покращення продуктивності, ми можемо вирішити використовувати інший стек технологій, який краще здатний досягти необхідного рівня продуктивності. Ми також можемо вирішити, що спосіб зберігання наших даних має змінитися для різних частин нашої системи. Наприклад, для соціальної мережі ми можемо зберігати

взаємодії наших користувачів у графо-орієнтованій базі даних для відображення сильно взаємопов’язаної природи соціального графа, але можливо, публікації, які роблять користувачі, можуть зберігатися в документо-орієнтованому сховищі даних.

Відмовостійкість. Ключовим поняттям у розробці стійкості є реакція системи на відмову. Якщо один компонент системи виходить з ладу, але цей збій не провокує відмови інших компонентів, ви можете ізолювати проблему та решта системи може продовжувати працювати. В монолітній системі, якщо сервіс дає збій, все перестає працювати. З монолітним ми можемо працювати на кількох машинах, щоб зменшити ймовірність відмови, але з мікросервісами, ми можемо створювати умови, що допомогають зменшити ризики збою у системі [8, с. 22].

Однак, нам потрібно бути обережними. Щоб наші мікросервісні системи могли належним чином прийняти цю покращену стійкість, нам потрібно зрозуміти нові джерела невдач, яким доводиться мати справу з розподіленими системами. Мережі можуть і будуть виходити з ладу, як і машини. Нам потрібно знати, як із цим впоратися та як на це вплине на кінцевого користувача нашого програмного забезпечення.

Масштабування. Маючи великий, монолітний сервіс, ми повинні масштабувати всі його компоненти разом. Якщо одна невелика частина системи виявляється обмеженою в продуктивності, це стає проблемою для всієї програми [8, с. 23]. У великому монолітному додатку масштабування відбувається як для цілого, навіть якщо лише деякі його частини потребують більше потужності. У випадку з мікросервісами, ми можемо масштабувати лише конкретні частини системи, які потребують більшої продуктивності, тим самим дозволяючи іншим частинам працювати на меншому, менш потужному обладнанні.

Розгортання. Зміна навіть одного рядка в довгій монолітній програмі з мільйонами рядків коду вимагає повного перезапуску програми для внесення змін. Це може призвести до значного втручання та підвищення ризику

випуску(розгортання програми в продуктив) [8, с. 24]. В практиці, такі складні та ризиковані випуски відбуваються рідко через страх перед можливими проблемами. На жаль, це означає, що зміни накопичуються між випусками, і коли нова версія програми потрапляє в експлуатацію, вона включає в себе велику кількість змін. І чим більше часу пройшло між випусками, тим більший ризик того, що ми можемо зробити щось неправильно.

За допомогою мікросервісів ми можемо внести зміни до одного сервісу та розгорнути їх незалежно від іншої частини системи. Це дозволяє нам швидше розгортати наш код. Якщо виникає проблема, її можна швидко ізолювати до окремого сервісу, що робить швидке відкатування простим. Це також означає, що ми можемо швидше надавати нові функціональні можливості нашим клієнтам.

Організаційна узгодженість. Багато людей стикалися з проблемами, пов'язаними з великими командами та обсягами коду. Ці проблеми можуть погіршуватися, коли команда розподілена. Ми також знаємо, що невеликі команди, які працюють над меншими обсягами коду, зазвичай є більш продуктивними [8, с. 25]. Мікросервіси дозволяють нам краще вирівняти нашу архітектуру з організацією, допомагаючи нам мінімізувати кількість людей, які працюють над будь-яким обсягом коду, щоб досягти оптимального балансу між розміром команди та продуктивністю. Ми також можемо переміщувати власність сервісів між командами, щоб намагатися зберегти співпрацю людей над одним сервісом в одному місці.

Композиційність. Однією з ключових переваг розподілених систем є можливість використання функціональності у різних контекстах. За допомогою мікросервісів ми можемо надавати нашу функціональність для різних цілей та застосувань. Це може бути особливо важливо, коли ми думаємо про те, як наші користувачі використовують наше програмне забезпечення [8, с. 26]. Вже минули часи, коли ми могли думати у вузьких категоріях лише про наш веб-сайт для настільних комп'ютерів або мобільний додаток. Тепер у нас є

можливості представляти клієнтам інформаційні послуги різними способами, наприклад веб-додаток, мобільний додаток, нативні пристрої та інше.

2.2 Компоненти мікросервісної архітектури

Кожна мікросервісна система будується з компонентів, які у купі представляються як злагоджена система та мають усі властивості, представлені у минулому підрозділі. Ще одною ознакою мікросервісів є наявність компонентів, які у силу властивостей мікросервісів мають змогу добавлятись, прибиратись або конфігуруватись окремо, не заважаючи іншим компонентам. Компоненти представляють собою окремі системи, які допомогають додати нову логіку або процес, щоб покращити життєвий цикл програмного забезпечення.

Ці компоненти спільно створюють гнучку та масштабовану архітектуру, яка дозволяє легко розробляти, випробовувати, впроваджувати та масштабувати складні додатки. Розглянемо основні:

- Сервіси. Основні будівельні блоки мікросервісної архітектури. Кожен сервіс відповідає за виконання певної функціональності. Вони можуть бути написані різними мовами програмування та використовувати технології, що найкраще підходять для їхнього завдання.
- Комунікація. Сервіси взаємодіють між собою через мережу, використовуючи зазвичай легкі та прозорі протоколи. Цей компонент особливо важливий, тому що сервіси не можуть існувати як монолітні додатки. Комунікація дозволяє системі обмінюватись даними та створювати консистентну інформаційну базу. Мікросервіси можуть обмінюватись інформацією за допомогою різних методів. Зазвичай, через популярні та захищені протоколи обміну даними, наприклад: HTTP / REST, gRPC, Message Brokers, GraphQL.
- Контейнеризація. Сервіси можуть бути упаковані в контейнери, які забезпечують їх ізольоване середовище виконання. Це дозволяє забезпечити консистентність роботи сервісів на різних середовищах від розробки до виробництва. Кожен контейнер містить усе необхідне

для виконання додатку: код, середовище виконання, бібліотеки, конфігураційні файли та інші ресурси.

- Реєстрація. Сервіси повинні бути здатні знаходити один одного в мережі. Для цього можуть використовуватись різне програмне забезпечення. Реєстрація сервісів допомогає об’єднувати сервіси в окрему мережу та забезпечувати конфіугацію у ній.
- Моніторінг та логування [9]. Моніторинг та логування є критичними аспектами будь-якої розподіленої системи, включаючи мікросервісну архітектуру. Ці процеси дозволяють вам відстежувати стан вашої системи, виявляти проблеми та реагувати на них. Основним терміном моніторингу систему є – лог. Лог – це структуровані або неструктуровані записи про події, які відбуваються в системі [10]. Журнали можуть включати інформацію про запити до серверів, помилки, винятки та інші важливі події. У мікросервісної архітектури логування є складним процесом, оскільки логування відбувається у великій кількості мікросервісів. Вирішити цю проблему дозволяє багато програмного забезпечення, наприклад, Kibana. Також важливо відслідковувати за станом середовища, де розташовані мікросервіси. Основними загрозами для мікросервісів може бути виток або нестача оперативної пам’яті, ресурсів центрального процесора та інше. Відслідковувати події за цим допомогає метрики – це числові дані, які відображають стан різних частин системи. Метрики можуть бути процесорними часами, використанням пам’яті, швидкістю відповіді запитів та іншими показниками продуктивності. Зазвичай ця інформація структурована та представлена у вигляді графиків та діаграм.
- Управління конфігурацією [9]. Сервіси повинні бути здатні працювати в різних середовищах з різною конфігурацією. Цей компонент дозволяє швидко конфігурувати програму під середовище. Оскільки мікросервіс не повинен бути прив’язаним до середовища, він має бути

легко сконфігурован та швидко доставлен у роботу, щоб запобігти використання зайвих людських ресурсів та часу.

- Масштабованість. Один з ключових переваг мікросервісної архітектури – здатність до масштабування. Кожен сервіс може бути масштабований окремо, забезпечуючи гнучкість та ефективність використання ресурсів. Існує два види масштабованості: вертикальна та горизонтальна. Вертикальна масштабованість досягається завдяки підвищенню спроможностей комп'ютеру, тобто підвищенню потужності системи через обладнання. Горизонтальна масштабованість здійснюється через збільшення числа додатків однієї програми, які розгорнуті у одній мережі та доступ до них відбувається через балансувальник.
- Доставка. Цей процес є способом розгортання та оновлення програми у мережі. Наприклад, поступила нова доробка до програмного забезпечення і її потрібно швидко запустити у роботу. Ручний процес поставки доробки займає дуже багато часу та ресурсів, тому швидка доставка дуже сильно економить людські ресурси та час. Для забезпечення безперервної доставки та швидкого впровадження змін часто використовуються практики DevOps та автоматизовані інструменти CI / CD.

Отже, компоненти в мікросервісній архітектурі є надзвичайно корисною в розробці мікросервісів. Вона дозволяє розділити систему на невеликі, самостійні модулі, що полегшує розуміння та розвиток системи. Кожен компонент може бути розроблений, тестований, розгорнутий та масштабований незалежно, що забезпечує більшу гнучкість та швидкість в розробці. Крім того, компоненти легко можна замінювати або модифікувати, що сприяє збереженню системи актуальною та готовою до змін у майбутньому. Швидкість розробки, розширюваність та масштабованість, а також можливість легко тестувати кожен компонент окремо роблять компонентну архітектуру ідеальним вибором для розробки мікросервісних додатків.

2.3 Комунікація та взаємодія між мікросервісами

На відміну від моноліту, архітектура мікросервісів структурує додаток як набір сервісів. Ці сервіси часто повинні співпрацювати для обробки запиту. Оскільки екземпляри сервісів зазвичай є процесами, що працюють на кількох машинах, вони повинні взаємодіяти за допомогою міжпроцесної комунікації (IPC). Вона відіграє набагато важливішу роль в архітектурі мікросервісів, ніж у монолітному додатку.

Насьогодні немає невеликого вибору технологій, які можуть бути використані для комунікації між додатками. Найпоширенішим вибором є REST (з JSON). Проте важливо пам’ятати, що не існує універсального рішення. Під час проектування системи, треба ретельно зважувати переваги та недоліки кожної технології. Кожна технологія може краще підходити для цього випадку, а в іншому призвести до втрати ресурсів [11, с. 66].

Існує багато різних технологій міжпроцесної комунікації на вибір. Сервіси можуть використовувати синхронні механізми комунікації на основі запитів та відповідей, такі як REST на основі HTTP або gRPC. Крім того, вони можуть використовувати асинхронні механізми обміну повідомленнями, такі як AMQP або STOMP. Також існує різноманіття форматів повідомлень. Сервіси можуть використовувати зрозумілі для людини, текстові формати, такі як JSON або XML. Альтернативно, вони можуть використовувати більш ефективні бінарні формати, такі як Avro або Protocol Buffers.

Існує два найпоширеніших способи взаємодії між мікросервісами: синхронний, асинхронний. При синхронному способі клієнт очікує своєчасної відповіді від сервісу і може навіть блокуватися, поки чекає. При асинхронному клієнт не блокується, і відповідь, якщо вона є, не обов’язково надсилається негайно [11, с. 67]. Розберемо популярні типи комунікації між сервісами:

1) Запит-відповідь (Request-Response)

   Цей спосіб є синхронним. Клієнт сервісу робить запит до сервісу і чекає на відповідь. Клієнт очікує, що відповідь надійде своєчасно. Він може навіть блокуватися під час очікування. Цей стиль взаємодії зазвичай

призводить до тісного зв’язку між сервісами. Основними прикладами є REST на основі HTTP, де клієнт надсилає HTTP-запит (GET, POST, PUT, DELETE) і отримує відповідь з даними у форматі JSON або XML. Також популярним методом є gRPC, де клієнт робить RPC-запит і отримує відповідь у більш ефективному бінарному форматі, такому як Protocol Buffers.

2) Повідомлення (Messaging)

   Спосіб комунікації повідомленнями є асинхронним. Один сервіс надсилає повідомлення іншому сервісу через посередника (message broker), і відповідь не очікується негайно. Основними технологіями є AMQP (Advanced Message Queuing Protocol), де сервіс надсилає повідомлення до черги, яка обробляється іншим сервісом, або STOMP (Simple Text Oriented Messaging Protocol), де у тілі запиту присутні текстові повідомлення.

3) Публікація-Підписка (Publish-Subscribe)

   Цей спосіб є асинхронним. Один сервіс публікує повідомлення, і всі підписані сервіси отримують його [11, с. 68]. Дуже часто використовується технології, як RabbitMQ або Apache Kafka.

4) Відправка команд (Command Sending)

   Відправка команд на інший сервіс є асинхронним варіантом обробки даних. Один сервіс надсилає команду іншому сервісу, вказуючи, що потрібно виконати певну дію. Приклад: CQRS (Command Query Responsibility Segregation), де команди надсилаються до сервісів для зміни стану системи.

5) Події (Event-Driven Communication)

   Цей спосіб є асинхронним. Комунікація на основі подій – це шаблон, в якому сервіси генерують та споживають повідомлення на основі подій, які відбуваються у системі. Подія – це повідомлення про зміну стану або даних сервісу, таке як нове замовлення, підтвердження платежу або оновлення запасів. Прикладом є шаблон проектування Event Sourcing, де

події зберігаються у вигляді журналу подій, а інші сервіси підписуються на ці події для обробки [12].

Ці типи взаємодії допомагають досягти гнучкості, масштабованості та стійкості у мікросервісній архітектурі, дозволяючи сервісам ефективно спілкуватися між собою залежно від вимог конкретної задачі.

2.4 Висновки по розділу

Отже, мікросервісна архітектура – це найпоширеніший вид розподілених систем, який має свої переваги від монолітних додатків. У розділі було розглянуто основні особливості мікросервісів, їх компоненти та концепція комунікації між ними. Цей підхід має численні переваги, зокрема більшу надійність, ефективність та безпеку порівняно з монолітними системами. Він також дозволяє використовувати різноманітні технології всередині кожного сервісу та легше масштабувати систему. У цілому, мікросервісна архітектура стала популярним вибором для розробки сучасних розподілених систем, завдяки своїм перевагам у гнучкості, незалежності та ефективності.

Для досягнення мети дипломної роботи, треба зазначити, що мікросервісна архітектура потребує в ретельному захисті. Згідно з згаданими особливостями, захист цієї системи набагато складніший, тому при виборі архітектури безпеки, потрібно знати та аналізувати сучасні та перевірені методи забезпечення захисту розподіленої системи.

3 ВИМОГИ ЩОДО БЕЗПЕКИ І НАДІЙНОСТІ МІКРОСЕРВІСНИХ АРХІТЕКТУР

3.1 Основні принципи безпеки у мікросервісних архітектурах

Система управління інформаційною безпекою повинна забезпечувати збереження конфіденційності, цілісності й доступності інформації [13]. Це також стосується і принципів безпеки кожної мікросервісної архітектури, хоча сама структура захищеності мікросервісів досить складна.

Основні принципи безпеки користувачів у мікросервісах включають у себе різноманітні аспекти, спрямовані на забезпечення захищеності їх даних і систем. Перш за все, це принцип забезпечення конфіденційності. Це означає, що дані користувачів повинні бути захищені від несанкціонованого доступу та перегляду. Для цього можуть використовуватися шифрування даних та інші методи захисту.

Другий принцип – це цілісність даних. Це означає, що дані повинні залишатися недоторканими і не піддаватися змінам без дозволу користувача або системного адміністратора. Забезпечення цілісності даних може включати в себе застосування контрольних сум та хеш-функцій.

Третім принципом є доступність. Це означає, що система повинна бути доступною для користувачів у разі необхідності. Це може бути досягнуто шляхом використання механізмів резервного копіювання та відновлення, а також шляхом моніторингу та управління навантаженням.

Також важливим принципом безпеки є постійний аудит та моніторінг стану захищеності інформаційної системи. Наприклад, процес SIEM дозволяє швидко реагувати на події безпеки та за потребою втручати спеціалістів задля вирішення проблеми [14]. Описані в минулих розділах технології та компоненти мікросервісної архітектури дозволяють повністю притримуватись усім принципам безпеки. Але постійне реагування на зміни в інформаційному

середовищі та періодичне вдосконалення системи безпеки є важливим аспектом, щоб максимально знизити ризики реалізації загроз.

Нажаль, немає універсального методу забезпечення безпеки мікросервісів. Замість цього розробники повинні враховувати кілька факторів при проектуванні безпеки та знаходити баланс між безпекою та ресурсів організації.

Розробляти систему захищеності у мікросервісних архітектурах набагато складніше, ніж монолітний додаток. Мікросервіси складаються з різних компонентів, які потрібно захищати кожен окремо. Налаштування безпеки не тільки доступу користувачів до ресурсу, а й мережі, комунікації всієї системи, що робить безпеку набагато складнішою. Також впровадження аудиту та моніторингу є набагато складнішим процесом, ніж моноліт, тому що потрібно не тільки збирати записи та метрики, а ще й збирати в одне місце, агрегувати, індексувати.

3.2 Ідентифікація користувачів у мікросервісних системах

Поступово зростаючи в онлайні, користувачі постійно ідентифікуються, аутентифікуються та авторизуються. Ці терміни часто використовуються взаємозамінно, але вони не однакові і працюють по-різному для досягнення конкретних завдань. Поняття аутентифікація та авторизація розберемо в наступному підрозділі.

Ідентифікація – перший крок у більшості онлайн-транзакцій і вимагає від користувача «ідентифікувати» себе, зазвичай, надаючи ім’я, електронну адресу, номер телефону або ім’я користувача [16]. Іншими словами, це процес встановлення того, хто саме намагається отримати доступ.

Розберемо приклад, що таке ідентифікація. Коли користувач хоче потрапити у банківський застосунок у веб-додатку, на його екрані з’являється два поля: логін та пароль. Коли користувач вводить свій логін (ідентифікаційне ім’я, електронна пошта, номер телефону та інше), це називається ідентифікація. Далі він проходить інші етапи розпізнавання його ідентичності.

Ідентифікація – дуже важливий етап в процесі авторизації та ідентифікації. Без нього неможливо відокремити користувача у системі та надати йому доступ до ресурсів та прав.

3.3 Методи аутентифікації та авторизації у мікросервісах

Аутентифікація та авторизація – це два фундаментальних поняття в галузі кібербезпеки, які часто помилково використовуються як синоніми. Хоча вони тісно пов’язані, між ними є важлива різниця.

Процес аутентифікації – це спосіб, яким користувач може довести, що вони все ще та сама особа, яку вони стверджували під час фази ідентифікації [16]. Це відповідає введенню пароля для користувача у інформаційній системі, щоб впевнитись, що це справді він. Наприклад, коли ви входите в систему за допомогою імені користувача та пароля, ви проходите аутентифікацію. Система перевіряє ваші облікові дані, щоб переконатися, що ви дійсно є тим, за кого себе видаєте.

Популярні методи, якими можна завершити процес аутентифікації:

- Паролі. Імена користувачів та паролі є найпоширенішими факторами аутентифікації. Якщо користувач вводить правильні дані, система вважає особу дійсною та надає доступ.
- Одноразові PIN-коди. Надають доступ лише для однієї сесії або транзакції.
- Аутентифікаційні програми. Генерують захисні коди через сторонню організацію, яка надає доступ.
- Біометрія. Користувач надає відбиток пальця або сканування райдужної оболонки ока для отримання доступу до системи.

В деяких випадках системам потрібно успішне підтвердження кількох факторів, перш ніж надавати доступ. Ця вимога багатофакторної аутентифікації (MFA) часто використовується для підвищення безпеки понад те, що можуть забезпечити самі паролі [15].

Авторизація – це процес визначення того, що користувач може робити в системі [16]. Це відповідає перевірки прав та дій, які може робити користувач у системі. Наприклад, навіть якщо ви успішно аутентифікувалися в системі, вам

все одно можуть бути заборонені певні дії, такі як доступ до конфіденційних даних або внесення змін до налаштувань. В безпечних середовищах авторизація завжди повинна відбуватися після автентифікації. Користувачі повинні спочатку підтвердити справжність своєї особи, перш ніж адміністратори організації нададуть їм доступ до запитуваних ресурсів.

Авторизація в більшість випадках реалізує модель авторизацію на основі ролей (RBAC) – це метод керування доступом, який використовує поняття ролей для визначення прав користувачів у системі [17]. Кожній ролі призначається набір дозволів, які визначають, які дії користувач може виконувати. Наприклад, роль «адміністратор» може мати дозвіл на доступ до всіх даних і налаштувань системи, тоді як роль «користувач» може мати дозвіл на доступ лише до певної частини даних і не може змінювати налаштування.

Отже, аутентифікація та авторизація це різні системи та підходи до безпеки інформаційно-комунікаційної системи. Аутентифікація допомагає визначити чи це ця людина, яка себе представляє, а авторизація лише перевіряє, що може робити користувач та які права у нього є на виконання дій у системі. [18]

У таблиці 3.1 наведено головні відмінності між цих двох термінів та головні особливості використання.

Таблиця 3.1 – відмінності аутентифікаціїї та авторизації

	Автентифікація	Авторизація
Що робить?	Перевіряє облікові дані	Видає права
Як це працює?	Через пароль, біометрія, PIN-код, інші системи	Через налаштування безпеки у системі

Продовження таблиці 3.1

Це видно користувачу?	Так	Ні
Користувач це може змінити?	Так	Ні

3.4 Протокол OAuth 2.0

Протокол OAuth 2.0 є досить популярним фреймворком у сучасному глобальному Інтернеті. Використовують цей фреймворк не тільки масштабні корпорації як Google, Facebook, Microsoft, а й невеликі стартапи. Цей протокол служить для об'єднання одного сервіса з іншим і дає змогу комунікувати не тільки веб-сайтам, а ще й мобільним, нативним додаткам, хмарним сервісам та іншим. Протокол дозволяє створити ще одну захисну оболонку над інформаційною системою. Він використовується усюди: від охорони здоров'я до банків, від енергетичної сфери до соціальних мереж.

З точки зору інформаційної безпеки, протокол Oauth 2.0 – це протокол делегування, який дозволяє власнику ресурсу дозволити програмному додатку отримувати доступ до цього ресурсу від його імені, не видаючи себе за нього. Замість того, щоб надавати додатку своє ім'я користувача та пароль, власник ресурсу надає додатку спеціальний токен (token), для доступу у систему [19, с. 3-4]. Система авторизації Oauth 2.0 дозволяє сторонньому додатку отримати обмежений доступ до HTTP-сервісу, або від імені власника ресурсу, організовуючи взаємодію затвердження між власником ресурсу та HTTP-сервісом, або дозволяючи сторонньому додатку отримати доступ власноруч.

В традиційному клієнт-серверному додатку щоб потрапити на захищений ресурс, тобі потрібно представити свої конфіденційні дані самому ресурсу, а потім серверу потрібно правильно зберігати дані кожного користувача [20, с. 4].

А якщо таких додатків багато, то потрібно з кожним ділитись своєю інформацією. Це робить таку систему вразливою до атак, тому що відповідальність за облікові та інші дані переміщується на всі сервери. Також додаткам потрібно підтримувати аутентифікацію по паролям, не дивлячись, що паролі не завжди можуть бути дати потрібний захист. Додатки отримують забагато інформації, що призводить до того, що користувач вже не може обмежувати доступ до своєї інформації та контролювати потік даних.

Oauth вирішує ці проблеми, вводячи шар авторизації та розділяючи роль клієнта і власника ресурсу. У Oauth клієнт запитує доступ до ресурсів, якими керує власник ресурсу і які розміщені на сервері ресурсу, та отримує інший набір облікових даних, ніж у власника ресурсу.

Замість використання облікових даних власника ресурсу для доступу до захищених ресурсів, клієнт отримує токен доступу — рядок, що позначає певний обсяг, тривалість і інші атрибути доступу. Токени доступу видаються стороннім клієнтам сервером авторизації з дозволу власника ресурсу. Клієнт використовує токен доступу для доступу до захищених ресурсів, розміщених на сервері ресурсу.

Наприклад, кінцевий користувач (власник ресурсу) може надати сервісу друку (клієнту) доступ до своїх захищених фотографій, які зберігаються на сервісі обміну фотографіями (сервері ресурсу), без передачі свого імені користувача та пароля сервісу друку. Натомість вона аутентифікується безпосередньо на сервері, якому довіряє сервіс обміну фотографіями (сервер авторизації), який видає сервісу друку облікові дані для конкретної делегації (токен доступу) [19, с. 4].

У специфікації Oauth 2.0 є 4 ролі:

- Власник ресурсу. Будь-який користувач, який здатний надати облікові дані захищеному ресурсу
- Сервер ресурсів. Сервер, що розміщує захищені ресурси, здатний приймати та відповідати на запити до захищених ресурсів, використовуючи токени доступу.

- Клієнт. Застосунок, який робить запити до захищених ресурсів від імені власника ресурсу та з його дозволу.
- Авторизаційний сервер. Сервер, що видає токени доступу клієнту після успішної аутентифікації власника ресурсу та отримання дозволу [20, с. 6].

Протокол Oauth 2.0 пропонує різнити типи дозволу та процесу авторизації (authorization grant). Специфікація визначає чотири типи: код авторизації (authorization code), внутрішній (implicit), пароль власника ресурсу (resource owner password credentials) та облікові дані клієнта (client credentials) [20, с. 8]. Кожен з типів авторизації має свої переваги. Це залежить від архітектури системи та політики безпеки. Кожен розробник повинен сам обирати найбільш вдалу реалізацію протоколу Oauth 2.0. Але всі способи досягають своєї мети – забезпечення шару безпеки в розподілених системах. На рисунку 3.1 зображена абстрактна діаграма процесу аутентифікації та авторизації користувача для отримання доступу до захищеного ресурсу.

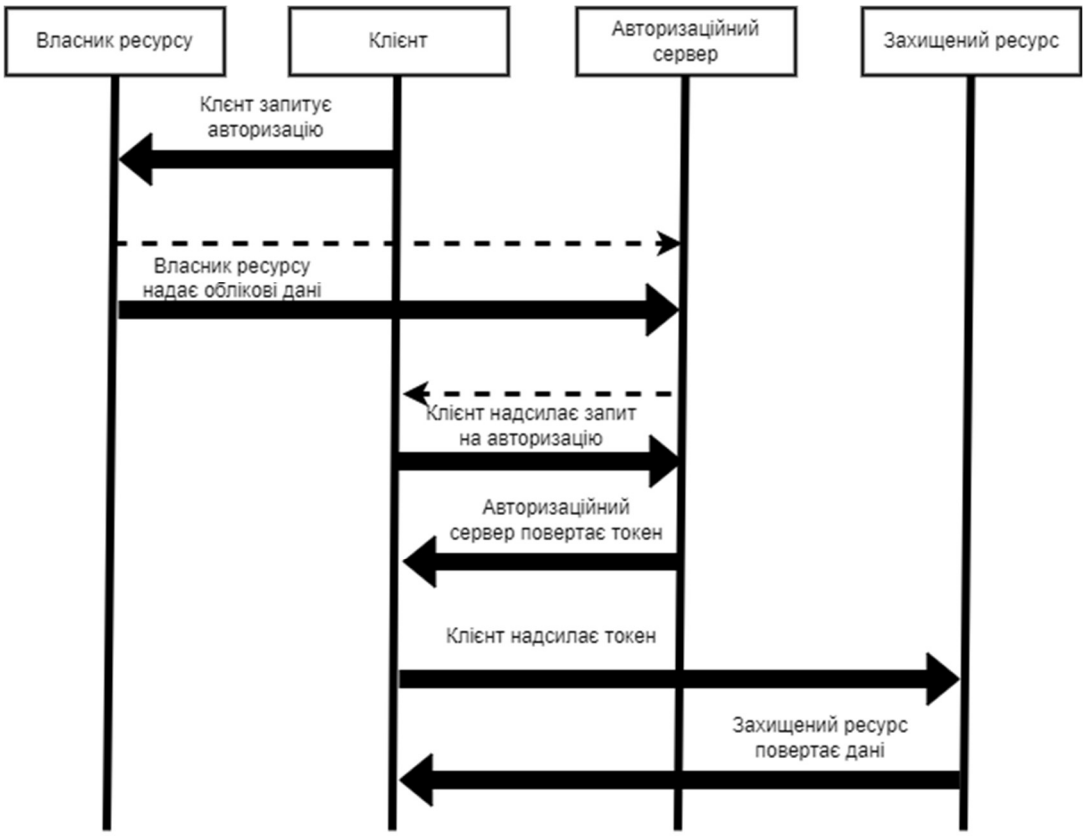

Рисунок 3.1 – абстрактна діаграма процесу Oauth 2.0 авторизації

Процес типу код авторизації отримується за допомогою сервера авторизації, який виступає посередником між клієнтом і власником ресурсу. Замість того, щоб запитувати дозвіл безпосередньо у власника ресурсу, клієнт спрямовує власника ресурсу до сервера авторизації, який, у свою чергу, повертає власника ресурсу до клієнта з кодом авторизації.

Перш ніж повернути власника ресурсу до клієнта з кодом авторизації, сервер авторизації аутентифікує власника ресурсу та отримує дозвіл. Оскільки власник ресурсу аутентифікується лише на сервері авторизації, облікові дані власника ресурсу ніколи не передаються клієнту.

Код авторизації надає кілька важливих переваг у сфері безпеки, таких як можливість аутентифікації клієнта, а також передачу токена доступу безпосередньо клієнту без проходження через користувацький агент власника ресурсу, що потенційно могло б призвести до його розкриття іншим особам, включаючи власника ресурсу.

Внутрішній тип — це спрощений процес отримання коду авторизації, оптимізований для клієнтів, реалізованих у браузері за допомогою скриптових мов, таких як JavaScript [20, с. 8]. В внутрішньому процесі, замість видачі клієнту коду авторизації, клієнту безпосередньо видається токен доступу (в результаті авторизації власника ресурсу). Тип дозволу називається внутрішнім, оскільки проміжні облікові дані (такі як код авторизації) не видаються (і пізніше не використовуються для отримання токена доступу).

Внутрішні дозволи підвищують швидкодію та ефективність деяких клієнтів (таких як клієнти, реалізовані як застосунки в браузері), оскільки зменшують кількість циклів запитів, необхідних для отримання токена доступу.

Процес типу пароль власника ресурсу являє собою процес, де власник ресурсу передає свої облікові дані і це безпосередньо є правом на отримання токену доступа. Цей тип рекомендується впроваджувати, якщо є висока довіра між власником ресурсу та клієнтом, або впровадження інших типів не мають сенсу.

Хоча цей тип дозволу вимагає прямого доступу клієнта до облікових даних власника ресурсу, облікові дані власника ресурсу використовуються лише для одного запиту і обмінюються на токен доступу. Цей тип дозволу може усунути потребу клієнта зберігати облікові дані власника ресурсу для майбутнього використання, обмінюючи облікові дані на довготривалий токен доступу або токен оновлення.

Облікові дані клієнта (або інші форми аутентифікації клієнта) можуть бути використані як дозвіл на авторизацію, коли обсяг авторизації обмежений захищеними ресурсами, що знаходяться під контролем клієнта, або захищеними ресурсами, раніше узгодженими з сервером авторизації. Облікові дані клієнта використовуються як дозвіл на авторизацію, зазвичай коли клієнт діє від свого імені (клієнт також є власником ресурсу) або запитує доступ до захищених ресурсів на основі авторизації, попередньо узгодженої з сервером авторизації [20, с. 9].

OAuth-токени є непрозорими для клієнта, що означає, що клієнту немає потреби (і часто можливості) дивитися на сам токен [20, с. 10]. Завданням клієнта є перенесення токену, запитуючи його у сервера авторизації та представляючи його захищеному ресурсу. Токен не є непрозорими для всіх у системі: завданням сервера авторизації є видавати токен, а завданням захищеного ресурсу – перевіряти токен. Таким чином, обидві сторони повинні мати можливість розуміти сам токен та його значення. Проте, клієнт абсолютно не усвідомлює всього цього. Цей підхід дозволяє клієнту бути набагато простішим, ніж це було б інакше необхідно, а також надає серверу авторизації та захищеному ресурсу неймовірну гнучкість у розгортанні цих токенів.

Протокол Oauth 2.0 дозволяє розміщати в токен важливу інформацію, яка може бути доступна серверу ресурсів. Авторизаційний сервер дозволяє клієнту вказувати обсяг (scope) запиту на доступ за допомогою параметра запиту «scope». У свою чергу, сервер авторизації використовує параметр відповіді «scope», щоб повідомити клієнта про обсяг виданого токена доступу.

Значення параметра «scope» виражається у вигляді списку чутливих до регістру рядків, розділених пробілами. Рядки визначаються сервером авторизації. Якщо значення містить кілька рядків, розділених пробілами, їх порядок не має значення, і кожен рядок додає додатковий діапазон доступу до запитуваного обсягу.

3.5 Висновки по розділу

Безпека в мікросервісній архітектурі є важливим аспектом для створення безпечної інформаційно-комунікаційної системи. Головними принципами безпеки системі, як і в будь-якій іншій, є конфіденційність, доступність та цілісність.

При розробці мікросервісної архітектури, потрібно правильно будувати безпеку, виходячи з вимог та ресурсів організації, тому що не існує універсального методу забезпечення захищеності даних. Потрібно правильно обирати технології та способи безпеки, щоб знизити ймовірність реалізації загроз.

Ідентифікація, аутентифікація та авторизація є важливими компонентами для побудування захищеної ІКС. Ідентифікація дозволяє дізнатись, хто ти є у системі та відрізнити від інших. Зазвичай, цей етап є процесом передачі логіну, що може бути ідентифікаційним ім’ям, поштою, номером телефону. Процес аутентифікації – це спроба впевнитись, що ти і є, за кого себе представляєш. Це може бути передачею паролю, PIN-коду або біометрія. Після цього, система робить процес авторизації, а саме надавання прав дозволу на використання тих чи інших ресурсів.

Популярним фреймворком для авторизації та аутентифікації є протокол OAuth 2.0. Протокол дозволяє делегувати процес авторизації та аутентифікації авторизаційному серверу, що є безперечно безпечно, ніж делегувати на кожний сервер ресурсів. Також OAuth 2.0 дає змогу гнучко налаштувати безпеку, шляхом надавання різних способів та типів авторизації.

Отже, дуже важливо створювати безпечний процес ідентифікації, аутентифікації та авторизації в розподіленій системи, щоб гарантувати конфіденційність, доступність та цілісність інформації.

4 РОЗРОБКА ВЛАСНОГО РІШЕННЯ НА БАЗІ ЗАПРОПОНОВАНОГО ПІДХОДА

4.1 Постановка задачі та аналіз вимог

Щоб досягнути головну мету дипломної роботи, а саме розробити безпечний спосіб аутентифікації та авторизації користувачів у розподіленій системі, потрібно визначити вимоги, які дозволять створити захищену та надійну інформаційно-комунікаційну систему. Також для створення застосунку поставимо задачі, які вона повинна виконувати.

Припустимо, що існує замовник, який хоче створити інформаційну систему для своєї організації страхування. Задача стоїть у створенні розподіленої системи, яка має змогу зберігати інформацію про клієнта та його страхування. Існує два типи ролей: клієнти та адміністратори. Клієнти можуть подивитись інформацію про страхування (тільки своє) та укласти його, створити свій профіль та передивлятись свої дані. Адміністратори мають можливість подивитись страхування будь-якого клієнта та видалити будь-якого користувача. Звісно, що клієнт не може робити дії адміністратора та навпаки.

Розподілена система повинна працювати на основі мікросервісної архітектури. Для взаємодії з сервісами треба використовувати глобальний Інтернет через протокол HTTP через технологію REST API. Навантаженість на сервер обробки запитів не висока, тому розподіл навантаженості не потрібен. Запити до серверів та відповіді повинні бути у формі JSON.

Система повинна бути захищена протоколом Oauth 2.0. Кожен виклик повинен бути аутентифікован та авторизован. Аутентифікація користувача у системі відбувається по його логіну та паролю, після цього його авторизують згідно даним, які зберігаються у базі даних. Паролі клієнта повинні бути зашифровані.

Уся інформація про клієнта та його страхування повинна знаходить у базі даних та керуватись адміністратором за потреби. Сутності та відносини між ними повинні бути нормалізовані та зручні для використання. Сутність

користувача має таку інформацію: електронна пошта, ім’я, призвіще та по-батькові, номер телефону, податковий номер, чи це організація, або приватна особа та дата народження (за потреби). Сутність страхування має такі властивості: електронна пошта клієнта, назва відділення, тип страхування, ціна, дата початку договору та кінця. Сама база даних повинна бути захищена за допомогою логіна та пароля, а також повинна комунікувати з розподіленою системою.

Отже, основні задачі, які потрібно зробити:

- Розробити авторизаційний сервер, який приймає запиту на авторизацію та аутентифікацію
- Розробити додатки, які під’єднуються до авторизаційного серверу та перекладують відповідальність процесу безпеки на авторизаційний сервер
- Створити базу даних для зберігання, додавання, змінення даних згідно вимог. Додати можливість підключення бази до серверів ресурсу
- Використовувати криптографічні алгоритми для шифрування паролів
- Розробити процес аутентифікації та авторизації за протоколом Oauth 2.0. Використати сучасні методи безпеки протоколу.
- Протестувати систему на функціональність та захищеність даних

4.2 Розробка системи збереження даних

Спираючись на вимоги замовника, які були представлені у минулому підрозділі, розробимо базу даних для збереження інформації. Для початку потрібно обрати СКБД, яка буде задовільняти вимогам. Для нашого проєкту буде використано СКБД PostgreSQL.

PostgreSQL – потужна, відкрита об’єктно-реляційна система баз даних з понад 35 роками активного розвитку, яка завоювала собі міцну репутацію надійності, стійкості функціоналу та продуктивності.[21]

Встановлюємо СКБД на сервер, який розташований у віртуальному середовищі Docker. Налаштовуємо сервер, надаючи йому порт. Доступ до

сервера відбувається через локальну мережу (IP: 127.0.0.1). Використовуємо команду:

```
docker run –name postgres-insurance -p 5432:5432 -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password -e POSTGRES_DB=postgresdb -d postgres:latest
```

яка запускає віртуальний сервер з діапазоном портів від 5432 до 5432, та встановлює змінні середовища: ім’я користувача, пароль та назву бази даних. На рисунку 4.1 зображено панель управління Docker, де ми бачимо, що сервер вдало запустився.

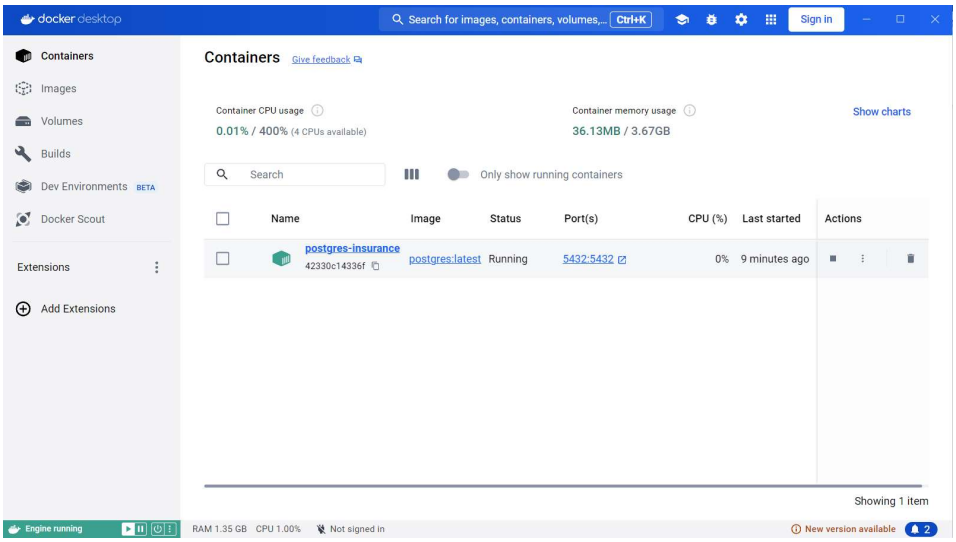

Рисунок 4.1 – панель управління Docker

Завдяки вимогам, нам відомо, що існує дві сутності: клієнт та страхування. Нехай кожний мікросервіс буде працювати зі своєю базою даних, тому, щоб не створювати багато нових баз даних, PostgreSQL пропонує створити схему (scheme) для кожного сервісу. Це допоможе кожному сервісу працювати тільки з тими даними, до яких він має доступ. Створюємо дві схеми з назвами “client” та “insurance”. Для зберігання інформації про авторизацію користувачів створимо окрему схему та назвемо її “oauth2” (див. рис. 4.2). Схема “pg_catalog” створюється системно, вона не має значення до проєкту.

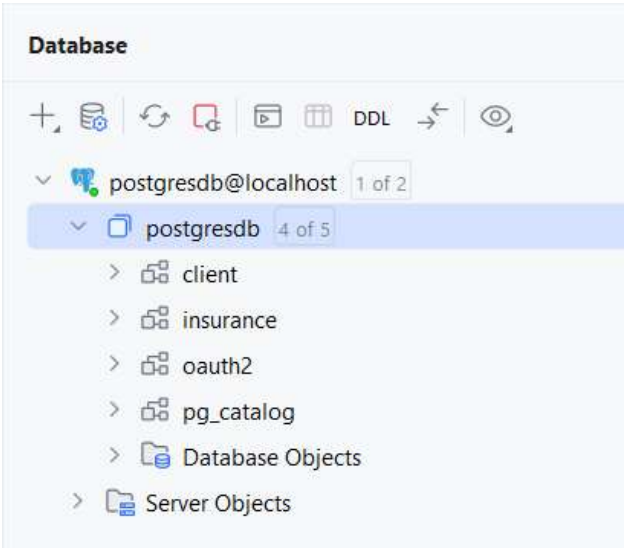

Рисунок 4.2 – створені схеми

У кожній схемі створюємо таблиці. У схемі “insurance” сутність страхування потрібно розділити на 3 таблиці та використовувати зовнішній ключ для посилання на інші таблиці. Таблиця “insurance” має два зовнішніх ключа на таблиці “type” та “department”. Первинний ключ в таблиці “insurance” є атрибут “id”, тобто ідентифікатор. На рисунку 4.3 зображено діаграма зв’язків сутностей.

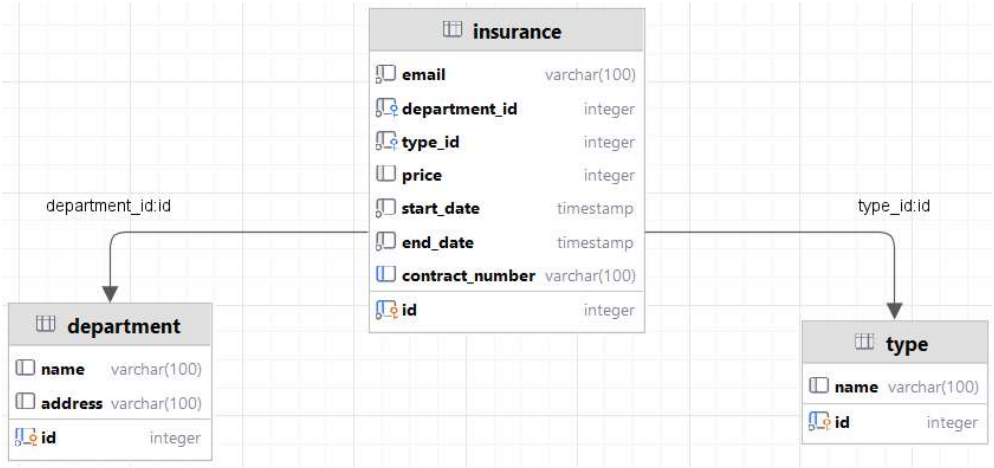

Рисунок 4.3 – діаграма зв’язків сутностей

У іншій схемі “client” створюємо лише одну таблицю “client”, яка зберігає потрібну інформацію про клієнтів системи. Первинний ключ є атрибут “id” (див. рис. 4.4).

Рисунок 4.4 – таблиця “client”

Схема для авторизації містить в собі п’ять таблиць, які виконують збереження інформації про ім’я користувачів, паролі, права доступу, зареєстровані клієнти, інформація про токен (з точки зору Oauth 2.0 ролей). Створимо п’ять таблиць: “users”, “authorities”, “oauth2_authorization_consent”, “oauth2_authorization”, “oauth2_registered_client”. Таблиця “authorities” має зовнішній ключ до таблиці “users” до атрибуту “username”. На рисунках 4.5, 4.6 зображено сутності та зв’язки між ними.

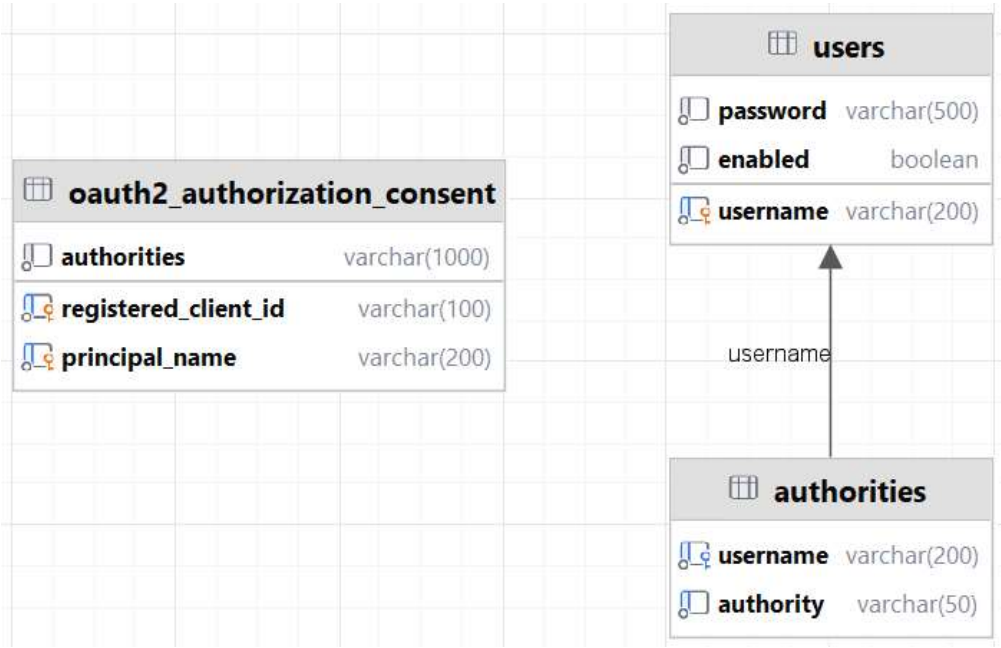

Рисунок 4.5 – зв’язки між сутностями

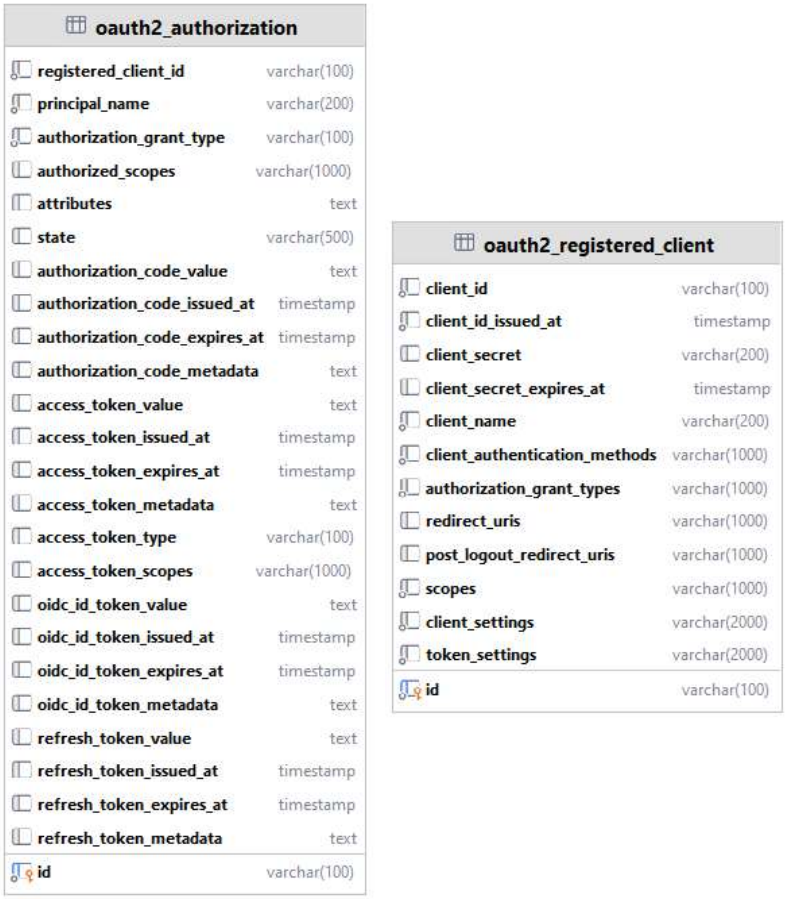

Рисунок 4.6 – сутності схеми “oauth2”

Більшість атрибутів в таблиці “oauth2_authorization” є допоміжними, що служать для різних типів авторизації та містять в собі додаткову інформацію про токен авторизації.

У додатку А знаходиться мова DDL мови SQL сутностей усіх таблиць, що були представлені для створення проєкту.

4.3 Розробка розподіленої системи

Для початку розробки розподіленої системи з протоколом Oauth 2.0, визначимось з технологіями, які будуть використовуватись у застосунку. Мова програмування, на якій буде написано додаток – Java. Мова програмування Java – це високорівнева, об'єктно-орієнтована мова програмування, розроблена компанією Sun Microsystems (пізніше придбана компанією Oracle) [22]. Мова програмування дозволяє під’єднувати бібліотеки, які допомогають використовувати вже існуючий код, що полегшує життя розробникам.

У проєкті буде використано фреймворк “Spring Framework”. “Spring Framework” забезпечує комплексну модель програмування та конфігурації для сучасних підприємницьких додатків на базі Java – на будь-якій платформі розгортання [23].

Фреймворк містить в собі багато корисних бібліотек для створення додатків, пов’язаних з розробкою веб-застосунків, роботою з базою даних, створення безпеки додатків та інше. Цей фреймворк містить в собі бібліотеку “Spring Oauth 2.0 authorization server”, який дозволяє створити авторизаційний сервер за протоколом Oauth 2.0. Кожний мікросервіс буде мати свої бібліотеки для досягнення мети самого додатку.

Проєкт буде складатись з трьох мікросервісів. Кожний сервіс має свою конфігурацію та налаштування. Відокремимо три компоненти: авторизаційний сервер, сервіс обробки даних страхувань та сервіс обробки даних клієнтів.

Створюємо новий проєкт у середовищі розробки та підгружаємо бібліотеки “Spring Framework”. Завантаження бібліотеки відбувається за допомогою інструмента Apache Maven у вигляді XML розмітки:

```
<dependencies>
    <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-data-jpa</artifactId>
       <version>3.2.3</version>
    </dependency>
    <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-oauth2-authorization-
server</artifactId>
    </dependency>
    <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-web</artifactId>
       <version>3.2.4</version>
    </dependency>
    <dependency>
       <groupId>org.postgresql</groupId>
       <artifactId>postgresql</artifactId>
       <scope>runtime</scope>
    </dependency>
</dependencies>
```

Завантажуємо основні бібліотеки для роботи з базою даних (spring-boot-starter-data-jpa, postgresql) та роботи авторизаційного серверу (spring-boot-starter-oauth2-authorization-server). Залежність “spring-boot-starter-web” дозволяє швидко налаштувати сам сервер, на якому працює додаток.

Конфігурація мікросервіса відбувається в окремому файлі “application.properties”. Там відбувається усе налаштування бібліотек та серверу за потребою. Це дозволяє розробнику гнучко створити застосунок та налаштувати його, не торкаючись самої програми. Часто у конфігураційному файлі вказуються URL, логіни, паролі до інших сервісів та інші константні значення. Основні налаштування мікросервісу представлені кодом у вигляді розмітки YAML нижче:

```
spring:
  datasource:
```

```
    password: password
    username: postgres
    url:
jdbc:postgresql://localhost:5432/postgresdb?currentSchema=oauth2
  sql:
    init:
      mode: always
  jpa:
    show-sql: true
    properties:
      hibernate:
        format_sql: true
    database: postgresql
```

У самому коді програми відокремимо 4 головних компоненти, які саме і працюють з базою даних та роблять аутентифікацію та авторизацію:

```
@Bean
JdbcOAuth2AuthorizationConsentService jdbcOAuth2AuthorizationConsentService(
        JdbcOperations jdbcOperations, RegisteredClientRepository repository) {
    return new JdbcOAuth2AuthorizationConsentService(jdbcOperations,
repository);
}

@Bean
JdbcOAuth2AuthorizationService jdbcOAuth2AuthorizationService(
        JdbcOperations jdbcOperations, RegisteredClientRepository rcr) {
    return new JdbcOAuth2AuthorizationService(jdbcOperations, rcr);
}

@Bean
RegisteredClientRepository registeredClientRepository(JdbcTemplate template) {
    return new JdbcRegisteredClientRepository(template);
}

@Bean
JdbcUserDetailsManager jdbcUserDetailsManager(DataSource dataSource) {
    return new JdbcUserDetailsManager(dataSource);
}
```

Всі чотири компоненти працюють з таблицями, які були створені у минулому підрозділі. Вони виконують роботу по під’єднанню бази даних до сервіса та надає бібліотеці необхідні дані для впровадження безпеки до захищеного ресурсу. Звісно, що всю обробку та процес авторизації виконує ця бібліотека, а розробнику залишається тільки підлаштувати під себе та вказати системі який тип процесу протоколу Oauth 2.0 йому потрібен.

Для шифрування паролів будемо використовувати алгоритм Bcrypt. Це функція хешування паролів, розроблена Нілсом Проусом та Девідом Маз’єром, на основі шифру Blowfish і представлена на конференції USENIX у 1999 році. Основна ідея BCrypt – це комбінація адаптивного хешування і «солі». «Сіль» – це додаткова випадкова стрічка даних, яка додається до пароля перед хешуванням. Це запобігає атакам типу «Rainbow table», де заздалегідь обчислені хеші можуть бути використані для злому паролів [24].

Представимо у коді новий компонент для хешування паролів:

```
@Bean
PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
}
```

Метод “createDelegatingPasswordEncoder()” повертає об’єкт Bcrypt шифрувальника. Коли користувач буде проводити аутентифікацію за допомогою пароля, програма отримає пароль та порівняє зі значенням у базі даних. Якщо пароль співпадає, далі починається процес авторизації. Користувачеві програма поверне токен авторизації, який містить в собі інформацію про нього.

У проєкті використовується JWT-токен для обміну між сервісами. JWT – рядок, що представляє набір даних у вигляді об'єкта JSON, який закодований або зашифрований, що дозволяє цим даним бути цифрово підписаними або математично визначеними та/або зашифрованими [25].

Для розробки серверів ресурсів знадобляться ті ж самі технології, тільки використовувати іншу бібліотеку та конфігурації серверу. Для того, щоб клієнт Oauth 2.0 зміг працювати з авторизаційним сервером, потрібно включити нову залежність:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-resource-server</artifactId>
    <version>6.2.2</version>
</dependency>
```

Бібліотека завантажує додатковий код для того, щоб була можливість зареєструвати клієнта та запитувати дозвіл на авторизацію. Обов'язковим етапом є конфігурація шляху до авторизаційного серверу:

```
spring:
    security:
        oauth2:
            resourceserver:
                jwt:
                    issuer-uri: http://localhost:8080
```

Для доступу до захищеного ресурсу, потрібно створити API, яка обробляє запити та робить транзакції до бази даних. "Spring Framework" дозволяє швидко створити обробник REST API запитів та відповіді на них, спираючись на HTTP протокол. Наведений код нижче демонструє шляхи до цих методів та перевірку прав доступу:

```
@GetMapping(«/insurance»)
@PreAuthorize("hasAuthority('CLIENT')»)
public List<InsuranceInfoResponse> getInsurances() {
    return insuranceService.getInsurances();
}

@PostMapping(«/insurance»)
@PreAuthorize(«hasAuthority('CLIENT')»)
public ResponseEntity<Map<String, String>> createInsurance(@RequestBody
InsuranceCreateRequest request) {
    return new ResponseEntity<>(Map.of(«contractNumber»,
insuranceService.createInsurance(request)), HttpStatusCode.valueOf(200));

}

@GetMapping(«/internal/insurance»)
@PreAuthorize(«hasAuthority('ADMIN')»)
public List<InsuranceInfoResponse> getInsurancesByEmail(@RequestParam String
email) {
```

```
    return insuranceService.getInsurancesByEmail(email);
}
```

Анотації, які стоять вище заголовків методів служать фреймворку допоміжною інформацією, як правильно інтерпретувати код. Анотація “GetMapping” означає, що спосіб доступу до API відбувається через HTTP метод GET. Анотація “PreAuthorize” говорить компілятору, що доступ до ресурсу повинен бути вказаним та перевіреним.

Налаштуємо другий сервер ресурсів так само, тільки використовуємо інший API для доступу до інформації про клієнта:

```
@PostMapping(«/client/register»)
@PreAuthorize(«hasAuthority('CLIENT')»)
public RegisterClientResponse registerClient(@Valid @RequestBody
RegisterClientRequest request) {
    return clientService.registerClient(request);
}

@GetMapping(«/client»)
@PreAuthorize(«hasAuthority('CLIENT')»)
public ClientInfoResponse getClientInfo() {
    return clientService.getClientInfo();
}

@DeleteMapping(«/client»)
@PreAuthorize(«hasAuthority('ADMIN')»)
public DeleteClientResponse deleteClient(@RequestParam String email) {
    return clientService.deleteClient(email);
}
```

Як ми бачимо, запити відбуваються до іншого сервісу, який обробляє транзакції до бази даних клієнта. Також ресурси захищені за допомогою ролевої моделі. Коли користувач намагається отримати доступ до ресурсу, зареєстрований клієнт перенаправляє користувача системи до форми логіну та паролю. Якщо введені дані коректні, то авторизаційний сервер повертає JWT-токен, який сервер ресурсів декодує та перетворює на об’єкт авторизації. Якщо токен містить потрібну роль, то сервер пропускає користувача за необхідними даними, або не впускає в систему з кодом помилки HTTP – 403 (Forbidden).

У додатку Б розташовано посилання на кодову базу програми для детального ознайомлення, а також для зручності код було добавлено у репозиторій GitHub [26].

4.4 Тестування системи

Для тестування системи нам знадобиться застосунок Postman, який має змогу робити HTTP запити та отримувати токен протоколу Oauth 2.0 через авторизаційний сервер.

Запустимо сервери на локальній мережі (127.0.0.1) та спробуємо зробити запит на авторизаціний сервер, щоб дізнатись метадані про сервіс (див. рис. 4.7).

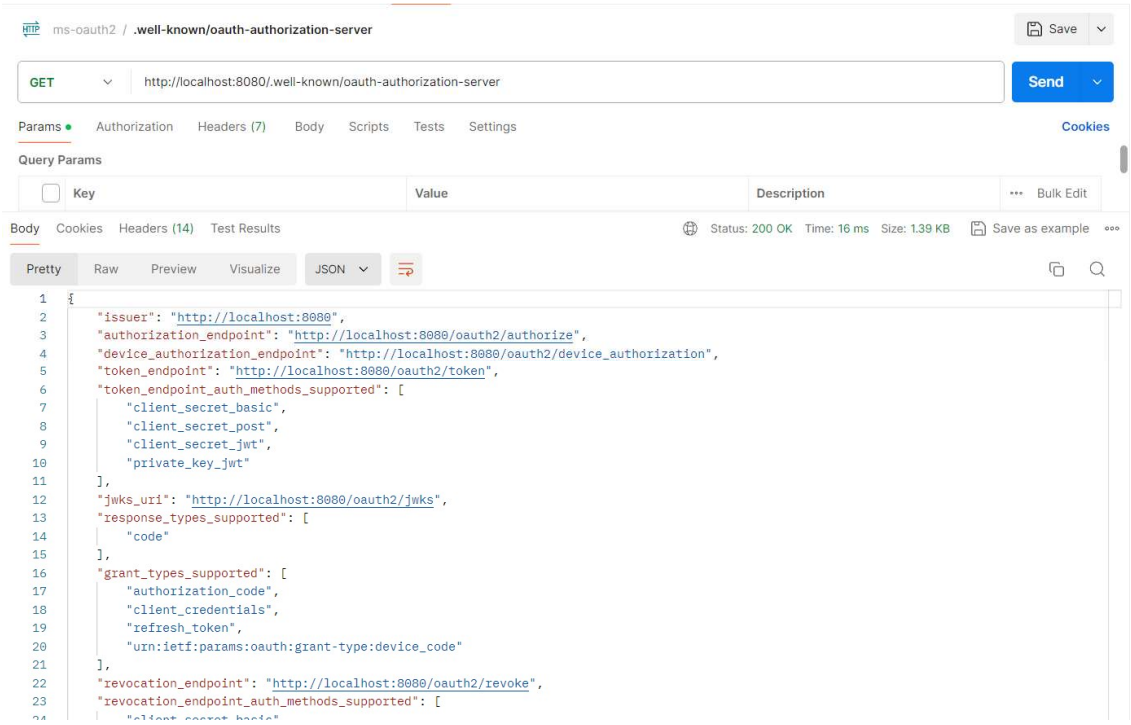

Рисунок 4.7 – запит на авторизаційний сервер

Запит на URL “/.well-known/oauth-authorization-server” дозволяє переглянути інформацію про сервіс, а саме адресу мережі видавця токена, URL для отримання токену авторизації, способи аутентифікації клієнтів Oauth 2.0, способи авторизації користувачів та інша корисна інформація.

Для тестування необхідно заповнити базу даних інформацією, щоб перевірити способність серверів обробляти інформацію та передавати її. На рисунку 4.8 бачимо, що таблиця “oauth2_registered_client” в схемі “oauth2”

заповнена даними. Зареєстрований клієнт Oauth 2.0 має ім’я, зашифрований пароль, доступні способи авторизації користувачів та інше.

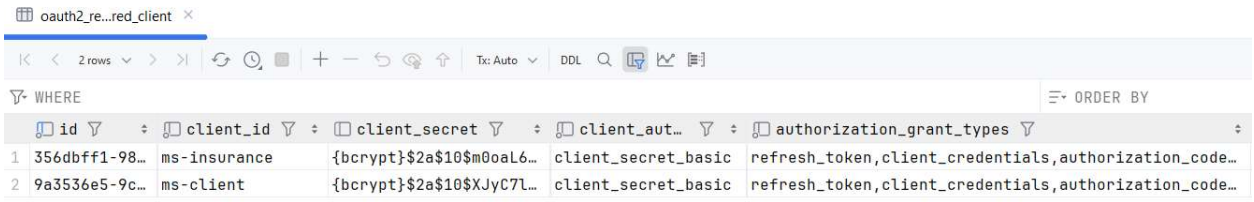

Рисунок 4.8 – заповнена таблиця

У застосунку Postman можливо генерувати токен доступу за допомогою вбудованих функцій (див. рис. 4.9).

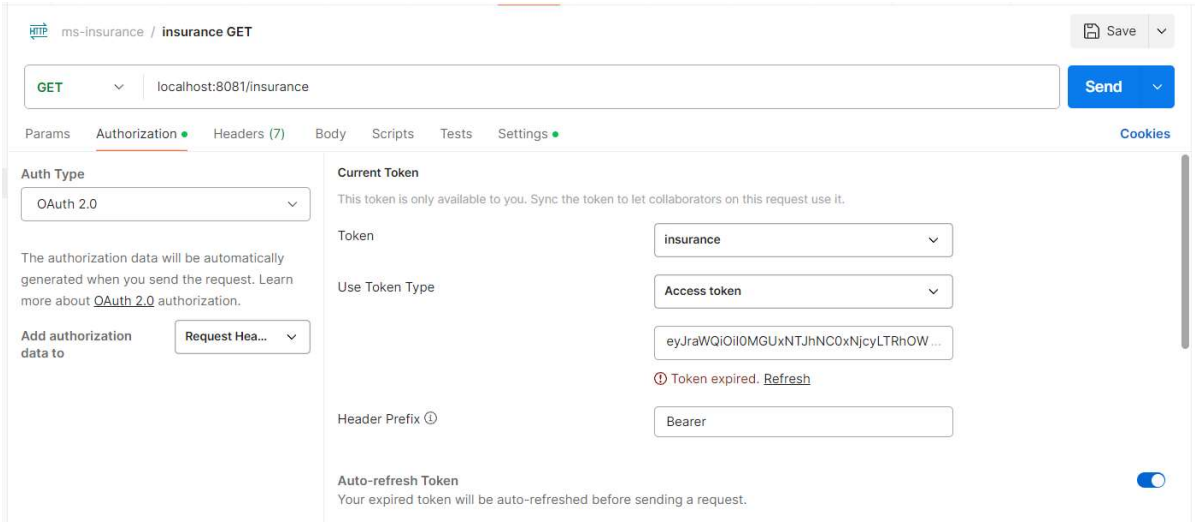

Рисунок 4.9 – вбудована функція отримання токену доступу

Для отримання авторизації, потрібно заповнити необхідні дані. На рисунку 4.10 зображено налаштування отримання доступу. Обираємо “Grant Type” як “Auhtorization Code”. Поле “Callback URL” – це шлях куди буде перенаправлен користувач після повної авторизації. Фреймворк дозволяє перенаправляти саме туди, звідки був перший запит до захищених ресурсів. Поле “Auth URL” – це шлях, де повинен отримуватись авторизаційний код, а поле “Access Token URL” – це шлях до отримання токену доступа. Вказуємо облікові дані клієнта Oauth 2.0 та спробуємо отримати токен.

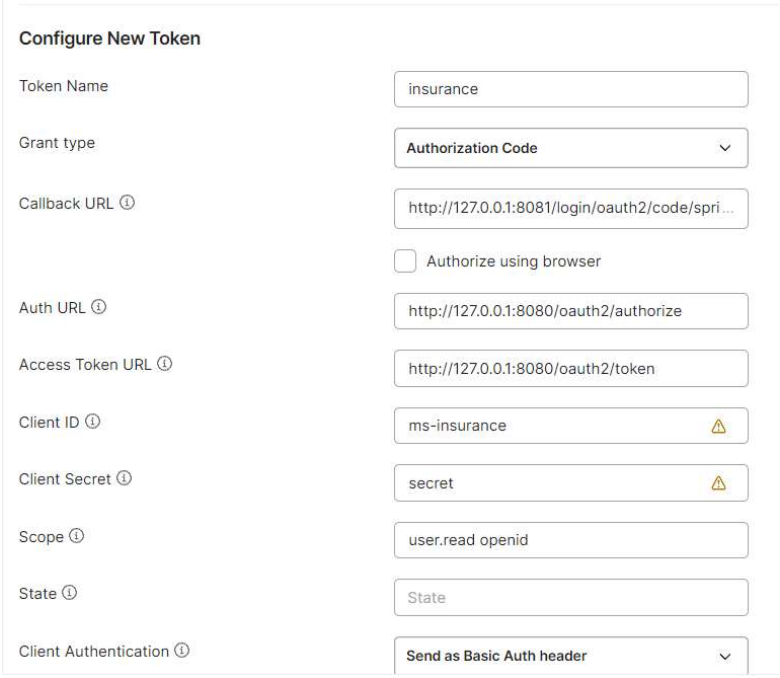

Рисунок 4.10 – налаштування отримання токену

Після цього відкривається вікно веб-браузеру, де потрібно ввести логін та пароль (див. рис. 4.11).

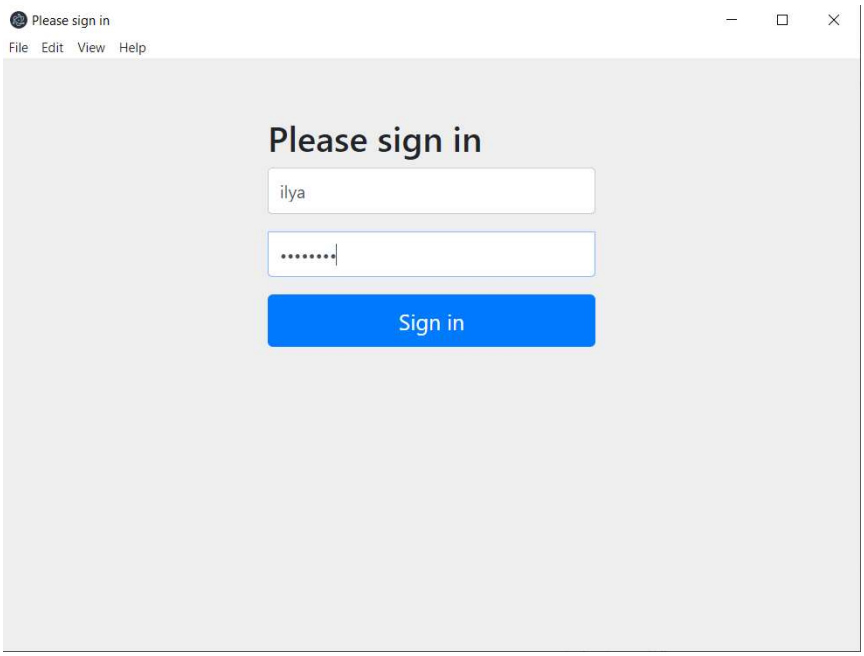

Рисунок 4.11 – вікно веб-браузеру

На рисунку 4.12 зображено отриманий токен, з яким ми маємо доступ до захищених ресурсів.

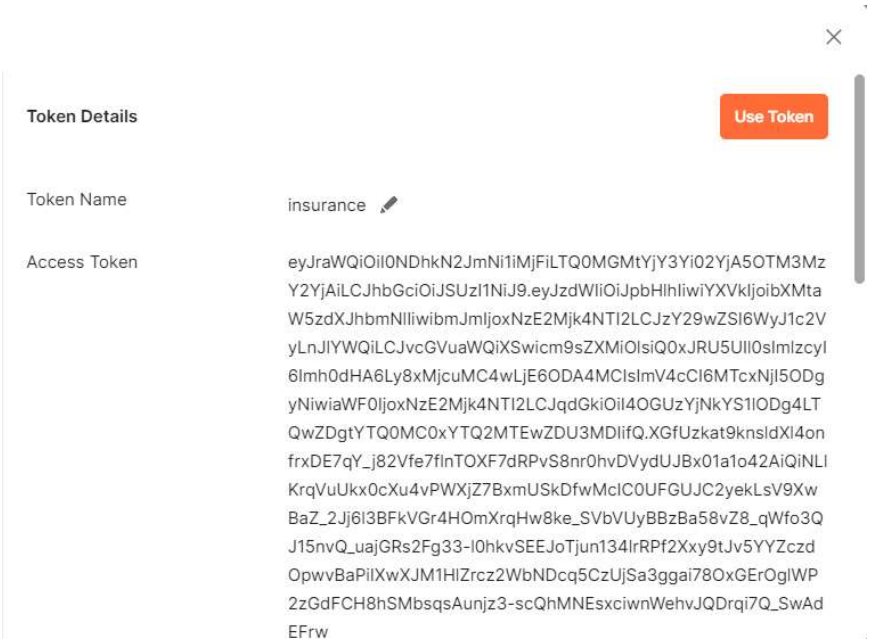

Рисунок 4.12 – отриманий токен доступу

Якщо розшифрувати JWT токен, то отримаємо інформацію про користувача та токен, а саме: ім’я користувача, назва клієнту Oauth 2.0, час отримання токену, час закінчення дії токену, роль користувача та URL адресу хто видав токен. Розшифровані дані у вигляді JSON об’єкта:

```
{
  «sub»: «ilya»,
  «aud»: «ms-insurance»,
  «nbf»: 1716298526,
  «scope»: [
    «user.read»,
    «openid»
  ],
  «roles»: [
    «CLIENT»
  ],
  «iss»: «http://127.0.0.1:8080»,
  «exp»: 1716298826,
  «iat»: 1716298526,
  «jti»: «88e3b3da-e888-40d8-a440-1a46110d5702»
}
```

Після цього, токен буде записаний у заголовки HTTP запиту, та сервер ресурсів буде отримувати цей токен та перевіряти його у програмі на наявність

ролей. На рисунку 4.13 зображено HTTP запит та відповідь на отримання захищенного ресурсу.

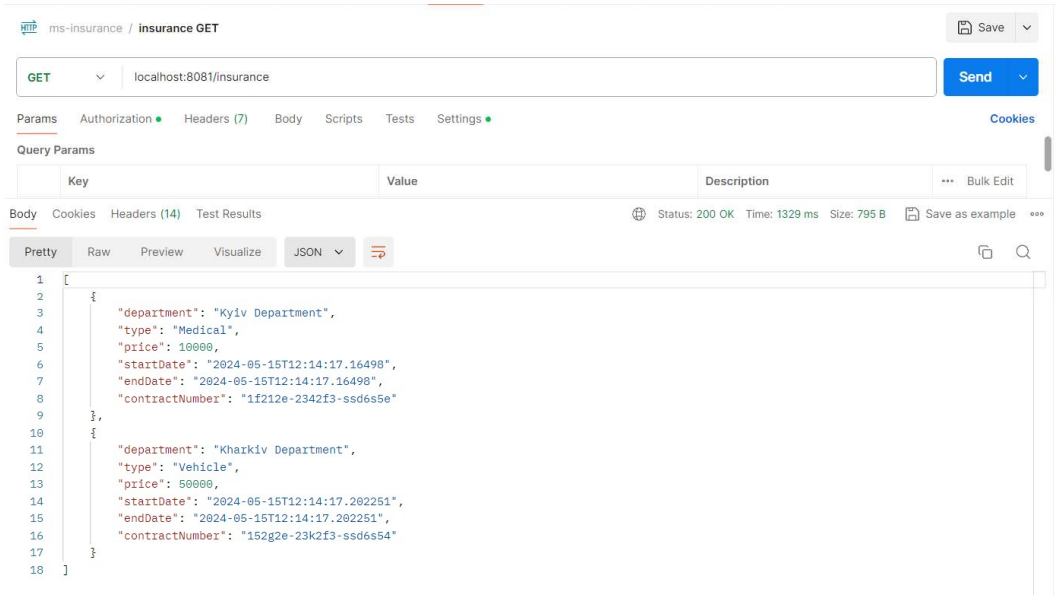

Рисунок 4.13 – HTTP запит та відповідь

Якщо ввести неправильний пароль, то звісно, що авторизаційний сервер поверне помилку (див. рис. 4.14).

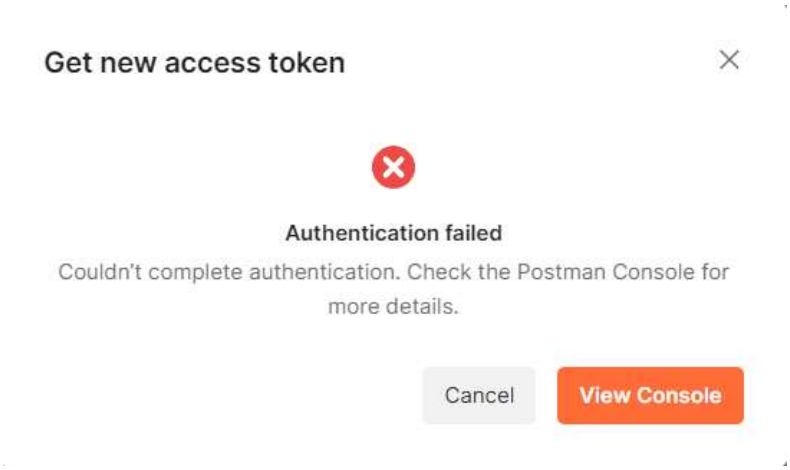

Рисунок 4.14 – спроба невдалої аутентифікації

Спробуємо зробити запит, який потребує прав адміністратора, коли у користувача є права тільки клієнта. На рисунку 4.15 зображено відповідь сервера на запит. Як ми бачимо, сервер повертає HTTP код 403, що означає заборону на отримання захищеного ресурсу.

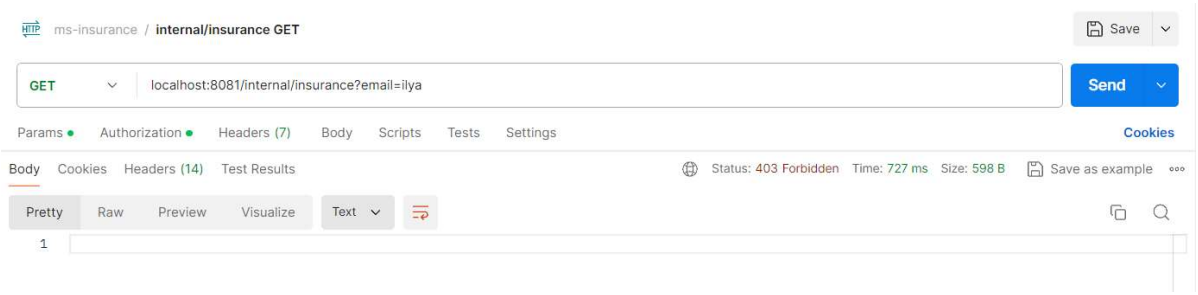

Рисунок 4.15 – спроба отримати захищений ресурс

Щоб впевнетись, що процес авторизаційного коду працює, як треба, подивимось на логування викликів у застосунку Postman при спробі отримати захищений ресурс. На рисунку 4.16 зображено послідовність викликів.

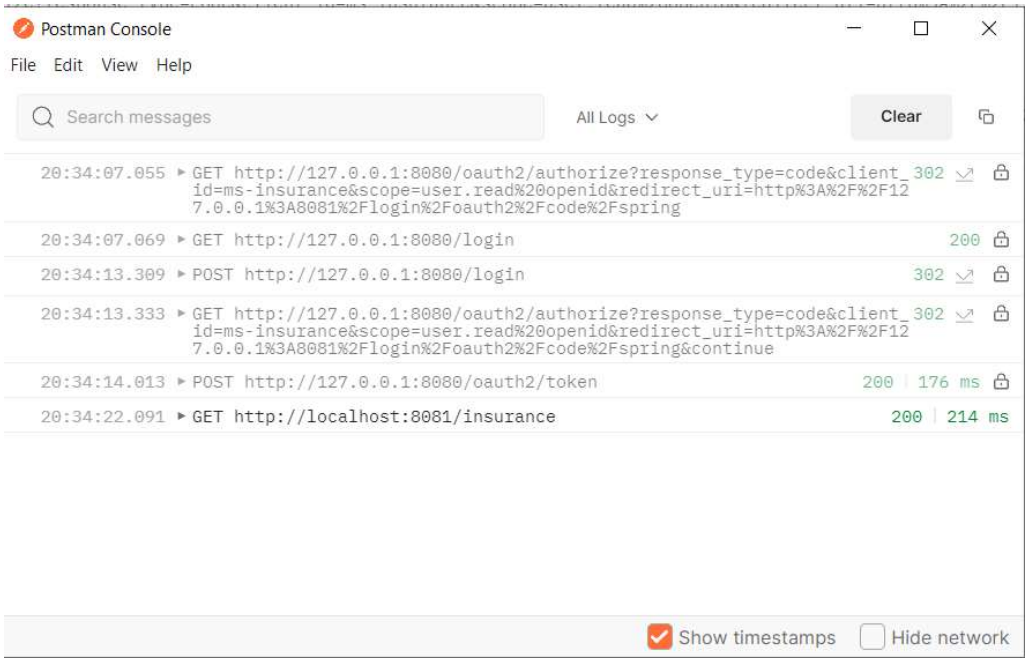

Рисунок 4.16 – логування викликів Postman

Перший виклик намагається авторизувати користувача у системі, та бачить, що він не аутентифікован. Далі йде перенаправлення на сторінку аутентифікації з вводом логіну та пароля. Після успішної аутентифікації, система посилає запит на отримання авторизаційного коду, тобто продовження першого виклику. Далі після завершення авторизації, авторизаційний сервер надсилає токен доступу (див. рис. 4.17). А останній виклик вже надходить до серверу ресурсів вже з токеном доступу у заголовках запиту.

Рисунок 4.17 – отримання токену доступу

4.5 Висновки по розділу

Отже, на початку розробки засобу безпечної аутентифікації та авторизації були висунуті вимоги щодо реалізації розподіленої системи. Також були поставлені задачи, які допомогають повністю та надійно виконати розробку.

У ході розробки була створена база даних PostgreSQL, яка складається зі схем, таблиць та зв’язків. За допомогою серверної віртуалізації Docker база даних має змогу працювати у реальному часі, незалежно від мікросервісів. СКБД захищена за допомогою логіна та пароля, що дозволяє мікросервісам безпечно під’єднуватись до неї.

Основна частина розробки проєкту – розробка мікросервісів. Було створено три мікросервіса: авторизаційний сервер, сервер обробки страхувань та клієнтів. Усі три мікросервіса працюють зі своєю схемою бази даних та зберігають там інформацію. Вдалось реалізувати протокол безпеки OAuth 2.0, який авторизує запити до мікросервісів. Самі мікросервіси складаються з бібліотек фреймворку “Spring Framework”, коду та конфіграції. Облікові дані клієнта, а саме пароль користувача зашифрований алгоритмом BCrypt.

Після розробки дипломного проєкту, було проведено належне тестування щодо роботоздатності та дотримання вимог. За допомогою додатку Postman,

вдалось протестувати працюючу систему HTTP запитами до авторизаційного серверу та серверів ресурсу. Тестування показало, що процес протоколу безпеки OAuth 2.0 повністю працює та має змогу безпечно зберігати та обробляти дані.

Отже, були виконані всі вимоги та задачи замовника, які були представлені на початку розділа.

ВИСНОВКИ

Отже, розподілені системи є популярною технологією, що дозволяє створювати величезні додатки, шляхом розбиття на компоненти. Основна ідея розподілених систем полягає у тому, що вони спільно працюють над вирішенням задачі, розділяючи навантаження та ресурси між собою. Для користувачів важливо забезпечити враження єдиної системи, незважаючи на те, що вона може бути розгорнута на різних пристроях чи комп'ютерах, що працюють у різних мережах.

Однією з ключових інновацій у сфері розподілених систем є мікросервісна архітектура. У порівнянні з традиційними монолітними додатками, мікросервісна архітектура дозволяє розбити додаток на невеликі, незалежні сервіси, які можуть функціонувати та масштабуватися незалежно один від одного. Це створює більш гнучкі та розширювані системи, оскільки кожен мікросервіс може бути розроблений, впроваджений та масштабований окремо.

Мікросервісна архітектура – це архітектурний стиль, вид розподіленої системи, де великий додаток розбивається на набір окремих компонентів, які незалежні один від одного. Мікросервіси надають більшу гнучкість, масштабованість та незалежність, в порівнянні з монолітом, що робить їх популярним вибором для розробки сучасних розподілених систем. Мікросервіси складаються з окремих компонентів, які допомогають виконувати різні задачі, починаючи від комунікації між сервісами, закінчуючи безпекою додатків.

Треба зазначити, що мікросервісна архітектура потребує в ретельному захисті. Згідно з згаданими особливостями, захист цієї системи набагато складніший, тому при виборі архітектури безпеки, потрібно знати та аналізувати сучасні та перевірені методи забезпечення захисту розподіленої системи.

Безпека в мікросервісній архітектурі є важливим аспектом для створення безпечної інформаційно-комунікаційної системи. Головними принципами безпеки системі, як і в будь-якій іншій, є конфіденційність, доступність та цілісність.

Ідентифікація, аутентифікація та авторизація є важливими компонентами для побудування захищеної ІКС. Ідентифікація дозволяє дізнатись, хто ти є у системі та відрізнити від інших. Зазвичай, цей етап є процесом передачі логіну, що може бути ідентифікаційним ім'ям, поштою, номером телефону. Процес аутентифікації – це спроба впевнитись, що ти і є, за кого себе представляєш. Це може бути передачею паролю, PIN-коду або біометрія. Після цього, система робить процес авторизації, а саме надавання прав дозволу на використання тих чи інших ресурсів.

Популярним фреймворком для авторизації та аутентифікації є протокол OAuth 2.0. Протокол дозволяє делегувати процес авторизації та аутентифікації авторизаційному серверу, що є безперечно безпечно, ніж делегувати на кожний сервер ресурсів. Також OAuth 2.0 дає змогу гнучко налаштувати безпеку, шляхом надавання різних способів та типів авторизації.

Під час розробки практичної частини дипломної роботи, були поставлені вимоги та задачи, які потрібно реалізувати. Спочатку було розроблено базу даних PostgreSQL, яка містила в собі схеми, таблиці та зв'язки. За допомогою серверної віртуалізації Docker база даних має змогу працювати у реальному часі, незалежно від мікросервісів. СКБД захищена за допомогою логіна та пароля, що дозволяє мікросервісам безпечно під'єднуватись до неї.

Основною частиною проєкту було сворення мікросервісів та безпеки на основі протоколу OAuth 2.0. Було створено три мікросервіса: авторизаційний сервер, сервер обробки страхувань та клієнтів. Усі три мікросервіса працюють зі своєю схемою бази даних та зберігають там інформацію. Вдалось реалізувати протокол безпеки OAuth 2.0, який авторизує запити до мікросервісів. Самі мікросервіси складаються з бібліотек фреймворку "Spring Framework", коду та

конфіграції. Облікові дані клієнта, а саме пароль користувача зашифрований алгоритмом BCrypt.

Після створення проєкту, було проведено тестування функціональності та дотримання вимог. За допомогою додатку Postman, вдалось протестувати працюючу систему HTTP запитами до авторизаційного серверу та серверів ресурсу. Тестування показало, що процес протоколу безпеки OAuth 2.0 повністю працює та має змогу безпечно зберігати та обробляти дані.

Для досягнення мети дипломної роботи було досліджено актуальність та розвиток розподілених систем у сучасному світі. Проаналізовано теоретичні дані про безпеку конфіденційної інформації та засоби її захисту у розподілених системах, включаючи інформацію про основі принципи безпеки даних у інформаційно-комунікаційних системах та сучасні методи аутентифікації та авторизації. Розроблено проєкт, спираючись на поставлені вимоги у 4 розділі та виконані основні завдання, які були поставлені.

Таким чином, дипломна робота зосереджена на важливій і актуальній темі кібербезпеки, пропонуючи практичне рішення для забезпечення надійної аутентифікації в розподілених системах, що є важливим внеском у сферу захисту особистих даних та інформаційної безпеки.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ДСТУ 3396.2-97. Технічний захист інформації. Терміни та визначення. [01.01.1998 р.]. Київ, 1998. 2 с. (Інформація та документація).
2. ДСТУ 3396.0-96. Технічний захист інформації. Основні положення. [01.01.1997 р.]. Київ, 1997. 2 с. (Інформація та документація).
3. Andrew S. Tanenbaum, Maarten Van Steen. Distributed systems: principles and paradigms : підручник. 2006. 1-2 с. Режим доступу: https://vowi.fsinf.at/images/b/bc/TU_Wien-Verteilte_Systeme_VO_%28G%C3%B6schka%29_-_Tannenbaum-distributed_systems_principles_and_paradigms_2nd_edition.pdf (дата звернення: 05.04. 2024).
4. Uzunov, A., E. Fernandez, and Katrina Falkner. Engineering security into distributed systems: A survey of methodologies : стаття. 2012. 1-2 с. Режим доступу: https://hekyll.services.adelaide.edu.au/dspace/bitstream/2440/77496/1/hdl_77496.pdf (дата звернення: 05.04. 2024).
5. Coulouris, George F., Jean Dollimore, and Tim Kindberg. Distributed systems: concepts and design. pearson education : підручник. 2005. 266 с. Режим доступу: https://books.google.com.ua/books?hl=uk&lr=&id=d63sQPvBezgC&oi=fnd&pg=PR11&dq=3.+Distributed+systems.+Concepts+and+Design.+Fifth+Edition+/+George+Coulouris,+Jean+Dollimore,+Tim+Kindberg,+Gordon+Blair&ots=qzoy1tCl_F&sig=66cQ5OaMHAuMNyut5Er9NY2bEyk&redir_esc=y#v=onepage&q&f=false (дата звернення: 06.04. 2024).
6. Microservices : веб-сайт. Режим доступу: https://microservices.io (дата звернення 07.04.2024)

7. Microservices vs. monolithic architecture: веб-сайт. Режим доступу: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith (дата звернення 07.03.2024)
8. Newman, Sam. Building microservices. " O'Reilly Media, Inc." : підручник. 2021. 19-26 с. Режим доступу: https://books.google.com.ua/books?hl=uk&lr=&id=d63sQPvBezgC&oi=fnd&pg=PR11&dq=3.+Distributed+systems.+Concepts+and+Design.+Fifth+Edition+/+George+Coulouris,+Jean+Dollimore,+Tim+Kindberg,+Gordon+Blair&ots=qzoy1tCl_F&sig=66cQ5OaMHAuMNyut5Er9NY2bEyk&redir_esc=y#v=onepage&q&f=false (дата звернення: 06.04. 2024).
9. Core Components of Microservices Architecture : веб-сайт. Режим доступу: https://www.linkedin.com/pulse/core-components-microservices-architecture-abid-anjum (дата звернення 07.04.2024)
10. Logging (computing) : веб-сайт. Режим доступу: https://en.wikipedia.org/wiki/Logging_(computing) (дата звернення 07.04.2024)
11. Chris Richardson. Microservices patterns: with examples in Java. Simon and Schuster : підручник. 2018. 66-68 с. Режим доступу: https://books.google.com.ua/books?hl=uk&lr=&id=QTgzEAAAQBAJ&oi=fnd&pg=PT21&dq=Chris+Richardson.+(2018).+Microservices+patterns:+with+examples+in+Java&ots=95b72GSHB8&sig=gV4fwFmcAIiavq0CUnmrjT9s9RQ&redir_esc=y#v=onepage&q=Chris%20Richardson.%20(2018).%20Microservices%20patterns%3A%20with%20examples%20in%20Java&f=false (дата звернення: 08.04. 2024).
12. What is the best way to implement event-driven and asynchronous communication for microservices? : веб-сайт. Режим доступу: https://www.linkedin.com/advice/1/what-best-way-implement-event-driven-asynchronous (дата звернення 10.04.2024)
13. ДСТУ ISO/IEC 27001:2015. Методи захисту системи управління інформаційною безпекою. [Чинний від 2015-12-18]. Київ, 2015. 5 с. (Інформація та документація).

14. Security information and event management : веб-сайт. Режим доступу: https://en.wikipedia.org/wiki/Security_information_and_event_management (дата звернення 11.04.2024)
15. What is Multi-Factor Authentication (MFA)? : веб-сайт. Режим доступу: https://aws.amazon.com/what-is/mfa/?nc1=h_ls (дата звернення 12.04.2024)
16. Defining identification & authentication : веб-сайт. Режим доступу: https://www.okta.com/identity-101/identification-vs-authentication/ (дата звернення 11.04.2024)
17. Role-based access control: веб-сайт. Режим доступу: https://en.wikipedia.org/wiki/Role-based_access_control (дата звернення 12.04.2024)
18. Authentication vs. Authorization: веб-сайт. Режим доступу: https://www.okta.com/identity-101/authentication-vs-authorization/ (дата звернення 12.04.2024)
19. Richer, Justin, and Antonio Sanso. OAuth 2 in action. Simon and Schuster : підручник. 2017. 3-4 с. Режим доступу: https://books.google.com.ua/books?hl=uk&lr=&id=jTozEAAAQBAJ&oi=fnd&pg=PT10&dq=OAuth+2.0+in+action&ots=pJ0qWFOE-r&sig=HIuyBBEwEg7h8U8CpTkPvWvgntw&redir_esc=y#v=onepage&q=OAuth%202.0%20in%20action&f=false (дата звернення: 12.04. 2024)
20. Dick Hardt. RFC 6749: The OAuth 2.0 authorization framework : стаття. 2012. 4-10 с. Режим доступу: https://datatracker.ietf.org/doc/html/rfc6749#section-1 (дата звернення: 12.04. 2024)
21. PostgreSQL: веб-сайт. Режим доступу: https://www.postgresql.org/ (дата звернення 15.04.2024)
22. Java: веб-сайт. Режим доступу: https://uk.wikipedia.org/wiki/Java (дата звернення 17.04.2024)
23. Spring Framework: веб-сайт. Режим доступу: https://spring.io/projects/spring-framework (дата звернення 17.04.2024)

24. BCrypt: веб-сайт. Режим доступу: https://en.wikipedia.org/wiki/Bcrypt (дата звернення 17.04.2024)
25. Jones Michael, John Bradley, Nat Sakimura. RFC 7519: JSON Web Token (JWT) : стаття. 2015. 5 с. Режим доступу: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 18.04. 2024)
26. GitHub repository. ms-oauth2 : веб-сайт. Режим доступу: https://github.com/IlyaKosarenko/ms-oauth2 (дата звернення 18.04.2024)

ДОДАТОК А

Лістинг DDL мови SQL для створення таблиць та зв’язків:

```
CREATE TABLE if not exists oauth2_registered_client
(
    id                        varchar(100)                              NOT NULL,
    client_id                 varchar(100)                              NOT NULL,
    client_id_issued_at       timestamp    DEFAULT CURRENT_TIMESTAMP NOT NULL,
    client_secret             varchar(200) DEFAULT NULL,
    client_secret_expires_at timestamp   DEFAULT NULL,
    client_name               varchar(200)                              NOT NULL,
    client_authentication_methods varchar(1000)                         NOT NULL,
    authorization_grant_types  varchar(1000)                            NOT NULL,
    redirect_uris              varchar(1000) DEFAULT NULL,
    post_logout_redirect_uris  varchar(1000) DEFAULT NULL,
    scopes                     varchar(1000)                            NOT NULL,
    client_settings            varchar(2000)                            NOT NULL,
    token_settings             varchar(2000)                            NOT NULL,
    PRIMARY KEY (id)
);


create table if not exists users
(
    username   varchar(200) not null primary key,
    password   varchar(500) not null,
    enabled    boolean      not null
);


create table if not exists authorities
(
    username  varchar(200) not null,
    authority varchar(50)  not null,
    constraint fk_authorities_users foreign key (username) references users
(username),
    constraint username_authority UNIQUE (username, authority)
);


create table if not exists oauth2_authorization_consent
(
    registered_client_id varchar(100)  NOT NULL,
    principal_name       varchar(200)  NOT NULL,
    authorities          varchar(1000) NOT NULL,
    PRIMARY KEY (registered_client_id, principal_name)
);

create table if not exists oauth2_authorization
(
    id                            varchar(100) NOT NULL,
    registered_client_id          varchar(100) NOT NULL,
    principal_name                varchar(200) NOT NULL,
    authorization_grant_type      varchar(100) NOT NULL,
    authorized_scopes             varchar(1000) DEFAULT NULL,
    attributes                    text          DEFAULT NULL,
    state                         varchar(500)  DEFAULT NULL,
    authorization_code_value      text          DEFAULT NULL,
    authorization_code_issued_at  timestamp     DEFAULT NULL,
    authorization_code_expires_at timestamp     DEFAULT NULL,
    authorization_code_metadata   text          DEFAULT NULL,
```

```
    access_token_value            text          DEFAULT NULL,
    access_token_issued_at        timestamp     DEFAULT NULL,
    access_token_expires_at       timestamp     DEFAULT NULL,
    access_token_metadata         text          DEFAULT NULL,
    access_token_type             varchar(100)  DEFAULT NULL,
    access_token_scopes           varchar(1000) DEFAULT NULL,
    oidc_id_token_value           text          DEFAULT NULL,
    oidc_id_token_issued_at       timestamp     DEFAULT NULL,
    oidc_id_token_expires_at      timestamp     DEFAULT NULL,
    oidc_id_token_metadata        text          DEFAULT NULL,
    refresh_token_value           text          DEFAULT NULL,
    refresh_token_issued_at       timestamp     DEFAULT NULL,
    refresh_token_expires_at      timestamp     DEFAULT NULL,
    refresh_token_metadata        text          DEFAULT NULL,
    user_code_value               text          DEFAULT NULL,
    user_code_issued_at           timestamp     DEFAULT NULL,
    user_code_expires_at          timestamp     DEFAULT NULL,
    user_code_metadata            text          DEFAULT NULL,
    device_code_value             text          DEFAULT NULL,
    device_code_issued_at         timestamp     DEFAULT NULL,
    device_code_expires_at        timestamp     DEFAULT NULL,
    device_code_metadata          text          DEFAULT NULL,
    PRIMARY KEY (id)
);


create schema insurance;

alter schema insurance owner to postgres;


create table if not exists insurance (
    id SERIAL PRIMARY KEY,
    email VARCHAR(100) NOT NULL,
    department_id INTEGER NOT NULL,
    type_id INTEGER NOT NULL,
    price INTEGER,
    start_date TIMESTAMP DEFAULT NOW() NOT NULL,
    end_date TIMESTAMP DEFAULT NOW() NOT NULL,
    contract_number VARCHAR(100) UNIQUE
);

CREATE TABLE IF NOT EXISTS department (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    address VARCHAR(100)
);

CREATE TABLE IF NOT EXISTS type (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100)
);

ALTER TABLE insurance
    ADD CONSTRAINT fk_department_id FOREIGN KEY (department_id) REFERENCES
department (id);

ALTER TABLE insurance
    ADD CONSTRAINT fk_type_id FOREIGN KEY (type_id) REFERENCES type (id);
```

ДОДАТОК Б

Лістинг додатку ms-oauth2 (авторизаційний сервер):

```
@EnableJpaRepositories
@SpringBootApplication
public class MsOauth2Application {

    public static void main(String[] args) {
       SpringApplication.run(MsOauth2Application.class, args);
    }

}
@Configuration
public class AuthorizationServerConfiguration {

    private static RSAKey generateRsa() {
        KeyPair keyPair = generateRsaKey();
        RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
        RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
        return new
RSAKey.Builder(publicKey).privateKey(privateKey).keyID(UUID.randomUUID().toStrin
g()).build();
    }

    private static KeyPair generateRsaKey() {
        KeyPair keyPair;
        try {
            KeyPairGenerator keyPairGenerator =
KeyPairGenerator.getInstance("RSA");
            keyPairGenerator.initialize(2048);
            keyPair = keyPairGenerator.generateKeyPair();
        } catch (Exception ex) {
            throw new IllegalStateException(ex);
        }
        return keyPair;
    }

    @Bean
    JdbcOAuth2AuthorizationConsentService jdbcOAuth2AuthorizationConsentService(
            JdbcOperations jdbcOperations, RegisteredClientRepository
repository) {
        return new JdbcOAuth2AuthorizationConsentService(jdbcOperations,
repository);
    }

    @Bean
    JdbcOAuth2AuthorizationService jdbcOAuth2AuthorizationService(
            JdbcOperations jdbcOperations, RegisteredClientRepository rcr) {
        return new JdbcOAuth2AuthorizationService(jdbcOperations, rcr);
    }

    @Bean
    public OAuth2TokenCustomizer<JwtEncodingContext> jwtTokenCustomizer() {
        return (context) -> {
            if (OAuth2TokenType.ACCESS_TOKEN.equals(context.getTokenType())) {
                context.getClaims().claims((claims) -> {
                    Set<String> roles =
AuthorityUtils.authorityListToSet(context.getPrincipal().getAuthorities())
                            .stream()
                            .map(c -> c.replaceFirst("^ROLE_", ""))
```

```
.collect(Collectors.collectingAndThen(Collectors.toSet(),
Collections::unmodifiableSet));
                    claims.put("roles", roles);
                });
            }
        };
    }

    @Bean
    JwtDecoder jwtDecoder(JWKSource<SecurityContext> jwkSource) {
        return OAuth2AuthorizationServerConfiguration.jwtDecoder(jwkSource);
    }

    @Bean
    JWKSource<SecurityContext> jwkSource() {
        RSAKey rsaKey = generateRsa();
        JWKSet jwkSet = new JWKSet(rsaKey);
        return new ImmutableJWKSet<>(jwkSet);
    }

}
@Configuration
public class ClientsConfiguration {

    @Bean
    RegisteredClientRepository registeredClientRepository(JdbcTemplate template)
{
        return new JdbcRegisteredClientRepository(template);
    }

}


@Configuration
public class PasswordEncoderCofiguration {

    @Bean
    PasswordEncoder passwordEncoder() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

}

@Configuration
public class UsersConfiguration {

    private final PasswordEncoder passwordEncoder;

    public UsersConfiguration(PasswordEncoder passwordEncoder) {
        this.passwordEncoder = passwordEncoder;
    }

    @Bean
    JdbcUserDetailsManager jdbcUserDetailsManager(DataSource dataSource) {
        return new JdbcUserDetailsManager(dataSource);
    }
}
```

Лістинг додатку ms-insurance (сервер ресурсів):

```
@RestController
public class InsuranceEndpoint {

    private final InsuranceService insuranceService;

    public InsuranceEndpoint(InsuranceService insuranceService) {
        this.insuranceService = insuranceService;
    }


    @GetMapping("/insurance")
    @PreAuthorize("hasAuthority('CLIENT')")
    public List<InsuranceInfoResponse> getInsurances() {
        return insuranceService.getInsurances();
    }

    @PostMapping("/insurance")
    @PreAuthorize("hasAuthority('CLIENT')")
    public ResponseEntity<Map<String, String>> createInsurance(@RequestBody
InsuranceCreateRequest request) {
        return new ResponseEntity<>(Map.of("contractNumber",
insuranceService.createInsurance(request)), HttpStatusCode.valueOf(200));

    }

    @GetMapping("/internal/insurance")
    @PreAuthorize("hasAuthority('ADMIN')")
    public List<InsuranceInfoResponse> getInsurancesByEmail(@RequestParam String
email) {
        return insuranceService.getInsurancesByEmail(email);
    }

}
@Service
public class InsuranceService {

    private final InsuranceRepository insuranceRepository;

    private final DepartmentRepository departmentRepository;

    private final TypeRepository typeRepository;

    private final ClientTokenInfoService clientTokenInfoService;


    public InsuranceService(InsuranceRepository insuranceRepository,
                            DepartmentRepository departmentRepository,
                            TypeRepository typeRepository,
                            ClientTokenInfoService clientTokenInfoService) {
        this.insuranceRepository = insuranceRepository;
        this.departmentRepository = departmentRepository;
        this.typeRepository = typeRepository;
        this.clientTokenInfoService = clientTokenInfoService;
    }


    public List<InsuranceInfoResponse> getInsurances() {
        String email = clientTokenInfoService.getEmailFromToken();
        return
insuranceRepository.getInsurancesByEmail(email).stream().map(this::map).toList()
;
```

```
    }

    public String createInsurance(InsuranceCreateRequest request) {
        Department department =
departmentRepository.findById(request.getDepartmentId())
                .orElseThrow(() -> new DepartmentNotFoundException("Department
not found"));

        Type type = typeRepository.findByName(request.getType())
                .orElseThrow(() -> new TypeNotFoundException("Type not found"));

        String contractNumber = UUID.randomUUID().toString();

        return insuranceRepository.save(toEntity(request, department, type,
contractNumber)).getContractNumber();
    }

    public List<InsuranceInfoResponse> getInsurancesByEmail(String email) {
        return
insuranceRepository.getInsurancesByEmail(email).stream().map(this::map).toList()
;
    }

    private InsuranceInfoResponse map(Insurance insurance) {
        InsuranceInfoResponse response = new InsuranceInfoResponse();
        response.setContractNumber(insurance.getContractNumber());
        response.setDepartment(insurance.getDepartment().getName());
        response.setPrice(insurance.getPrice());
        response.setType(insurance.getType().getName());
        response.setEndDate(insurance.getEndDate());
        response.setStartDate(insurance.getStartDate());
        return response;
    }

    private Insurance toEntity(InsuranceCreateRequest request, Department
department, Type type, String contractNumber) {
        Insurance insurance = new Insurance();
        insurance.setDepartment(department);
        insurance.setType(type);
        insurance.setPrice(request.getPrice());
        insurance.setEmail(clientTokenInfoService.getEmailFromToken());
        insurance.setStartDate(request.getStartDate());
        insurance.setEndDate(request.getEndDate());
        insurance.setContractNumber(contractNumber);
        return insurance;
    }

}

@Service
public class ClientTokenInfoService {

    public String getEmailFromToken() {
        Authentication auth =
SecurityContextHolder.getContext().getAuthentication();
        return Optional.ofNullable(auth.getName()).orElseThrow(() -> new
EmailNotFoundException("Email not found from authentication token"));
    }

}

@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig {
```

```
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests()
                .anyRequest()
                .authenticated()
                .and()
                .oauth2ResourceServer()
                .jwt()
                .decoder(new CustomJwtDecoder())
                .jwtAuthenticationConverter(jwtAuthenticationConverter());
        return http.build();
    }

    @Bean
    public JwtAuthenticationConverter jwtAuthenticationConverter() {
        JwtAuthenticationConverter jwtAuthenticationConverter = new
JwtAuthenticationConverter();
        jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(jwt -> {
            List<String> authorities = jwt.getClaim("roles");
            return
authorities.stream().map(SimpleGrantedAuthority::new).collect(toList());
        });

        return jwtAuthenticationConverter;
    }

}
```

Лістинг додатку ms-client (сервер ресурсів):

```
@RestController
public class InternalClientController {

    private final ClientService clientService;

    public InternalClientController(ClientService clientService) {
        this.clientService = clientService;
    }


    @PostMapping("/client/register")
    @PreAuthorize("hasAuthority('CLIENT')")
    public RegisterClientResponse registerClient(@Valid @RequestBody
RegisterClientRequest request) {
        return clientService.registerClient(request);
    }

    @GetMapping("/client")
    @PreAuthorize("hasAuthority('CLIENT')")
    public ClientInfoResponse getClientInfo() {
        return clientService.getClientInfo();
    }

    @DeleteMapping("/client")
    @PreAuthorize("hasAuthority('ADMIN')")
    public DeleteClientResponse deleteClient(@RequestParam String email) {
        return clientService.deleteClient(email);
    }

}
```

```
@Service
@Transactional
public class ClientService {

    private final ClientRepository clientRepository;

    private final ClientTokenInfoService clientTokenInfoService;

    public ClientService(ClientRepository clientRepository,
                         ClientTokenInfoService clientTokenInfoService) {
        this.clientRepository = clientRepository;
        this.clientTokenInfoService = clientTokenInfoService;
    }

    public RegisterClientResponse registerClient(RegisterClientRequest request)
{
        String email = clientTokenInfoService.getEmailFromToken();
        clientRepository.save(toEntity(request, email));
        return new RegisterClientResponse("OK");
    }

    public ClientInfoResponse getClientInfo() {
        String email = clientTokenInfoService.getEmailFromToken();
        return map(clientRepository.findByEmail(email).orElse(new Client()));
    }

    public DeleteClientResponse deleteClient(String email) {
        var client = clientRepository.deleteByEmail(email);
        if (client.isEmpty()) {
            return new DeleteClientResponse("Client with email " + email + " not
found");
        }
        return new DeleteClientResponse("OK");
    }

    private Client toEntity(RegisterClientRequest request, String email) {
        Client client = new Client();
        client.setEmail(email);
        client.setFirstName(request.getFirstName());
        client.setSurname(request.getSurname());
        client.setThirdName(request.getThirdName());
        client.setPhoneNumber(request.getPhoneNumber());
        client.setTaxId(request.getTaxId());
        client.setIsCompany(request.getCompany());
        client.setBirthDate(request.getBirthDate());
        return client;
    }

    private ClientInfoResponse map(Client client) {
        ClientInfoResponse response = new ClientInfoResponse();
        response.setBirthDate(client.getBirthDate());
        response.setEmail(client.getEmail());
        response.setCompany(client.getIsCompany());
        response.setFirstName(client.getFirstName());
        response.setSurname(client.getSurname());
        response.setThirdName(client.getThirdName());
        response.setTaxId(client.getTaxId());
        response.setPhoneNumber(client.getPhoneNumber());
        return response;
    }

}

@Repository
public interface ClientRepository extends JpaRepository<Client, Long> {
```

```
    Optional<Client> findByEmail(String email);

    Optional<Client> deleteByEmail(String email);

}
```