Полтавський університет економіки і торгівлі
Навчально-науковий інститут денної освіти
Форма навчання денна
Кафедра комп'ютерних наук та інформаційних технологій

Допускається до захисту
Завідувач кафедри
_______Олена ОЛЬХОВСЬКА
(підпис)

«_____»_________202_ р.

КВАЛІФІКАЦІЙНА РОБОТА
на тему

«СТВОРЕННЯ СЕРВЕРНОЇ АРХІТЕКТУРИ HTTP-СЕРВІСУ КОРИСТУВАЧІВ НА ОСНОВІ МОВИ ПРОГРАМУВАННЯ GO»

зі спеціальності 122 Комп'ютерні науки
освітня програма «Комп'ютерні науки»
ступеня бакалавр

Виконавець роботи Сизько Ростислав Сергійович
______«____» ________202_ р.
(підпис)

Науковий керівник доцент, к.ф.-м.н. Черненко О. О.
_______«_____»_________202_ р.
(підпис)

Рецензент

ПОЛТАВА 2026

РЕФЕРАТ

Записка: 74 с., 13 рис., 2 таблиці, 1 додаток, 14 джерел.

СЕРВЕРНА АРХІТЕКТУРА, HTTP-СЕРВІС, GO, УПРАВЛІННЯ КОРИСТУВАЧАМИ, АВТЕНТИФІКАЦІЯ, JWT, ДВОФАКТОРНА АВТЕНТИФІКАЦІЯ, RBAC, REST API, POSTGRESQL, CLEAN ARCHITECTURE, МІКРОСЕРВІС

Об'єктом розробки є серверна архітектура HTTP-сервісу управління користувачами для веб- та мікросервісних додатків.

Предметом розробки є програмна реалізація модулів автентифікації, авторизації, контролю доступу, аудиту та керування ідентичністю на основі мови програмування Go.

Метою роботи є створення серверної архітектури HTTP-сервісу користувачів, що забезпечує повний цикл управління ідентифікацією та доступом (Identity and Access Management) і може бути використана як автономний бекенд або як основа для інтеграції з зовнішніми клієнтськими додатками.

Результатом роботи стало розроблення серверного застосунку «User Service» на базі мови програмування Go з використанням маршрутизатора chi та реляційної СУБД PostgreSQL. Реалізовано ключові модулі:

- модуль автентифікації — реєстрація, вхід, JWT-токени з ротацією refresh-токенів, верифікація електронної пошти, скидання пароля;
- модуль двофакторної автентифікації — підтримка одноразових паролів TOTP за стандартом RFC 6238 (сумісність з Google Authenticator);
- модуль управління користувачами — операції CRUD, керування профілем, м'яке видалення, обов'язкові дії при вході;
- модуль контролю доступу RBAC — ролі, індивідуальні дозволи, групи з успадкуванням прав;
- модуль безпеки — обмеження частоти запитів, блокування облікових записів після невдалих спроб, хешування паролів алгоритмом bcrypt;
- модуль аудиту — журналювання критичних дій у системі та історія входів;

- модуль API-ключів та керування активними сесіями користувача;
- адміністративна панель — односторінковий веб-інтерфейс на основі Bootstrap 5.

Особливості: тришарова Clean Architecture, контейнеризація на базі Docker, автоматичне застосування міграцій бази даних, інтерактивна документація API на основі специфікації OpenAPI 3.0 (Swagger UI).

Проведено модульне тестування сервісного шару (модуль user_service_test.go), а також ручне функціональне тестування усіх HTTP-ендпоінтів через інтерфейс Swagger UI та через адміністративну панель.

«User Service» може використовуватися як автентифікаційний бекенд для веб-додатків і мобільних клієнтів, як ядро ідентичності для мікросервісних систем або як легковагова альтернатива промисловим IAM-платформам типу Keycloak і Auth0 для невеликих та середніх проєктів.

ЗМІСТ

ВСТУП 7
ПОСТАНОВКА ЗАДАЧІ 10
1. ІНФОРМАЦІЙНИЙ ОГЛЯД 13
1.1. Аналіз предметної області управління ідентифікацією та доступом 13
1.2. Огляд існуючих програмних рішень у сфері IAM 15
1.3. Обґрунтування доцільності власної розробки 18
2. ТЕОРЕТИЧНА ЧАСТИНА 21
2.1. Архітектурні принципи побудови HTTP-сервісів на мові Go 21
2.2. Механізми автентифікації на основі JWT та ротація refresh-токенів 24
2.3. Модель контролю доступу RBAC і двофакторна автентифікація TOTP 25
2.4. PostgreSQL як сховище даних для IAM-сервісу 27
3. ПРАКТИЧНА ЧАСТИНА 29
3.1. Загальна архітектура системи та структура проєкту 29
3.2. Проєктування схеми бази даних і система міграцій 31
3.3. Реалізація модуля автентифікації: реєстрація, вхід, JWT та refresh-токени 33
3.4. Модуль контролю доступу RBAC: ролі, дозволи та групи 36
3.5. Двофакторна автентифікація на основі TOTP 38
3.6. Управління сесіями, API-ключами, аудит дій та обмеження запитів 40
3.7. Адміністративна панель та документація API 42
3.8. Інструкція для користувача та системного адміністратора 44
ВИСНОВКИ 48
СПИСОК ІНФОРМАЦІЙНИХ ДЖЕРЕЛ 51
ДОДАТОК А 53

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ, ТЕРМІНІВ

Умовні позначення, символи, скорочення, терміни	Пояснення умовних позначень, скорочень, символів
API	Application Programming Interface — програмний інтерфейс взаємодії
CRUD	Create, Read, Update, Delete — базові операції над даними
CSS	Cascading Style Sheets — каскадні таблиці стилів
HTML	HyperText Markup Language — мова розмітки гіпертексту
HTTP	HyperText Transfer Protocol — протокол передачі гіпертексту
HTTPS	HyperText Transfer Protocol Secure — захищена версія HTTP
IAM	Identity and Access Management — управління ідентифікацією та доступом
JSON	JavaScript Object Notation — текстовий формат обміну даними
JWT	JSON Web Token — стандарт токенів для автентифікації (RFC 7519)
ORM	Object-Relational Mapping — об'єктно-реляційне відображення
RBAC	Role-Based Access Control — рольова модель управління доступом
REST	Representational State Transfer — архітектурний стиль взаємодії
RFC	Request for Comments — серія документів з

	технічними стандартами
SMTP	Simple Mail Transfer Protocol — протокол передачі електронної пошти
SPA	Single Page Application — односторінковий веб-додаток
SQL	Structured Query Language — мова структурованих запитів
SSL	Secure Sockets Layer — протокол захищеного передавання даних
TLS	Transport Layer Security — протокол захищеного передавання даних
TOTP	Time-based One-Time Password — одноразовий пароль на основі часу (RFC 6238)
UI	User Interface — інтерфейс користувача
URL	Uniform Resource Locator — уніфікований локатор ресурсу
UUID	Universally Unique Identifier — універсальний унікальний ідентифікатор
БД	База даних
СУБД	Система управління базами даних

ВСТУП

Сучасний ринок програмного забезпечення характеризується стрімким зростанням кількості веб-додатків, мобільних клієнтів і мікросервісних систем, кожен з яких потребує надійного та безпечного механізму управління ідентифікацією користувачів. Питання автентифікації, авторизації та захисту персональних даних набувають критичного значення в умовах постійного зростання кількості кібератак і посилення вимог регуляторів до забезпечення приватності, зокрема положень Закону України «Про захист персональних даних» та Загального регламенту захисту даних (GDPR) Європейського Союзу.

За даними щорічних галузевих звітів про витоки даних (Verizon Data Breach Investigations Report 2023), близько 80 % інцидентів інформаційної безпеки пов'язані саме з компрометацією облікових записів користувачів. Це підкреслює важливість побудови якісного, добре спроєктованого та захищеного серверного компонента, що відповідає за управління ідентичністю та доступом (Identity and Access Management, IAM) і є фундаментом будь-якої сучасної інформаційної системи.

Промислові платформи IAM, такі як Keycloak, Auth0, Okta та Authentik, надають широкий спектр можливостей, проте їх упровадження у малих і середніх проєктах часто є економічно та технічно недоцільним. Великі платформи мають значні системні вимоги, складну архітектуру з десятків компонентів, потребують окремої команди супроводу, мають крутий поріг входження та накладають істотні обмеження на гнучкість бізнес-логіки. Альтернативою є реалізація власного спеціалізованого IAM-сервісу, який містить лише необхідні функції, легко інтегрується з існуючою інфраструктурою та може бути адаптований під конкретні потреби проєкту.

Мова програмування Go (Golang), розроблена компанією Google, упевнено зайняла позицію стандарту de facto для побудови високопродуктивних серверних застосунків та мікросервісів. Серед її ключових переваг — статична типізація,

вбудована підтримка конкурентного виконання через горутини та канали, мінімалістичний синтаксис, стандартна бібліотека з повноцінним HTTP-сервером, а також можливість компіляції в єдиний бінарний файл, що значно спрощує розгортання у контейнерному середовищі. За опитуванням розробників Stack Overflow Developer Survey за 2023 рік, Go входить до десятки мов з найвищим рівнем задоволеності розробників.

Метою роботи є створення серверної архітектури HTTP-сервісу користувачів на основі мови програмування Go, яка забезпечує повний цикл управління ідентифікацією та доступом і може застосовуватися як автономне рішення або як ядро для побудови мікросервісних систем.

Для досягнення поставленої мети у роботі визначено такі основні завдання:

- проаналізувати предметну область управління ідентифікацією та доступом, дослідити основні поняття IAM, типові процеси автентифікації й авторизації, вимоги до сучасних рішень;
- провести огляд існуючих програмних рішень — як комерційних (Auth0, Okta), так і відкритих (Keycloak, Authentik, Ory Kratos) — і обґрунтувати доцільність розробки власного компактного IAM-сервісу;
- дослідити теоретичні засади побудови REST API на мові Go, принципи Clean Architecture, механізми автентифікації на основі JWT та реалізації RBAC і двофакторної автентифікації за стандартом TOTP;
- спроєктувати архітектуру системи, схему реляційної бази даних PostgreSQL, систему версіонування міграцій та структуру програмного коду;
- реалізувати ключові модулі сервісу: автентифікацію, RBAC з підтримкою ролей, дозволів та груп, двофакторну автентифікацію TOTP, керування сесіями та API-ключами, аудит дій і обмеження частоти запитів;
- розробити адміністративну панель у вигляді односторінкового веб-додатку та інтерактивну документацію API на базі специфікації OpenAPI 3.0;
- провести модульне та функціональне тестування реалізованих компонентів, оформити інструкцію для користувача та системного адміністратора.

Об'єктом дослідження є процеси управління ідентифікацією та доступом у веб-додатках і мікросервісних системах.

Предметом дослідження є програмна реалізація серверного компонента IAM на основі мови Go, що включає модулі автентифікації, авторизації, аудиту, двофакторної автентифікації та керування активними сесіями користувача.

Методи дослідження. У роботі застосовано методи системного аналізу для дослідження предметної області, методи об'єктно-орієнтованого та функціонального проєктування для побудови архітектури, методи реляційної алгебри та нормалізації для проєктування схеми даних, а також практичні методи розробки програмного забезпечення з використанням систем контролю версій та контейнеризації.

Практичне значення одержаних результатів полягає у створенні готового до використання модульного серверного застосунку, який може бути інтегрований у реальні комерційні проєкти, а також у систематизації знань про сучасні підходи до побудови IAM-сервісів на мові Go.

Кваліфікаційна робота складається зі вступу, постановки задачі, трьох розділів, висновків, списку інформаційних джерел та одного додатку. Загальний обсяг записки становить 55 сторінок, робота містить 12 рисунків, 2 таблиці та 1 додаток. Список використаних джерел налічує 14 найменувань.

ПОСТАНОВКА ЗАДАЧІ

Основною задачею кваліфікаційної роботи є проєктування та реалізація серверної архітектури HTTP-сервісу управління користувачами на мові програмування Go, що забезпечує безпечне виконання повного циклу операцій з ідентичностями користувачів і відповідає сучасним вимогам до інформаційної безпеки веб-додатків.

Розроблюваний сервіс повинен надавати уніфікований REST-інтерфейс для зовнішніх клієнтських додатків, забезпечувати персистентне зберігання даних користувачів у реляційній СУБД PostgreSQL, підтримувати стандартні механізми автентифікації за стандартом JWT, а також містити вбудовану адміністративну панель для управління користувачами, ролями, групами та правами доступу.

Функціональні вимоги до сервісу передбачають наявність модулів реєстрації, автентифікації, контролю доступу, двофакторної автентифікації, аудиту дій і керування активними сесіями. Нефункціональні вимоги охоплюють безпеку (зокрема, хешування паролів алгоритмом bcrypt, обмеження частоти запитів, тимчасове блокування облікових записів після невдалих спроб входу), продуктивність (низькі затримки відповіді, ефективне використання пам'яті) та простоту розгортання (контейнеризація на базі Docker, автоматичні міграції бази даних).

Для досягнення поставленої задачі необхідно виконати такі підзадачі:

1. проаналізувати сучасні підходи до побудови серверної архітектури HTTP-сервісів управління користувачами, виявити типові архітектурні шаблони та вимоги до безпеки;
2. провести огляд існуючих рішень у сфері IAM, виконати порівняльний аналіз їх функціональних можливостей, продуктивності та складності розгортання, скласти таблицю порівняння аналогів;
3. дослідити теоретичні засади побудови REST API на основі мови Go: маршрутизацію HTTP-запитів, шари абстракції та принципи Clean

Architecture, механізми автентифікації за стандартом JWT, моделі контролю доступу RBAC, двофакторну автентифікацію TOTP;

4. обґрунтувати вибір технологічного стеку: версії мови Go, маршрутизатора, драйвера PostgreSQL, бібліотек для роботи з JWT, TOTP, шифрування паролів та документування API;
5. спроєктувати багатошарову архітектуру системи з чітким розподілом обов'язків між шарами доменних моделей, репозиторіїв, сервісів та обробників HTTP-запитів;
6. спроєктувати реляційну схему бази даних із підтримкою користувачів, ролей, груп, дозволів, токенів, історії входів, журналу аудиту, секретів TOTP та API-ключів, реалізувати систему версіонування міграцій;
7. реалізувати модуль автентифікації з підтримкою реєстрації, входу, ротації refresh-токенів, верифікації електронної пошти та скидання пароля;
8. реалізувати модуль контролю доступу RBAC, що підтримує індивідуальні дозволи користувача, групи користувачів та успадкування дозволів від груп;
9. реалізувати модуль двофакторної автентифікації за стандартом TOTP з можливістю генерації QR-кодів для сумісних застосунків (Google Authenticator, Authy);
10. реалізувати модуль керування активними сесіями користувача, видачі та відкликання довгоживучих API-ключів, модуль аудиту дій та механізм обмеження частоти запитів;
11. розробити адміністративну панель у вигляді односторінкового веб-додатку, що дозволяє керувати користувачами, групами, дозволами та переглядати журнал аудиту;
12. згенерувати інтерактивну документацію API на базі специфікації OpenAPI 3.0 за допомогою інструмента swag;
13. підготувати конфігурацію Docker та docker-compose для розгортання сервісу разом із базою даних, написати модульні тести сервісного шару;
14. провести функціональне тестування усіх реалізованих ендпоінтів API через Swagger UI та адміністративну панель, оформити інструкцію для користувача

та системного адміністратора.

1. ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1. Аналіз предметної області управління ідентифікацією та доступом

Управління ідентифікацією та доступом (Identity and Access Management, IAM) є фундаментальним напрямом інформаційної безпеки, який об'єднує політики, процеси та технічні засоби для забезпечення того, що відповідні особи у відповідний час отримують відповідний доступ до відповідних ресурсів. У контексті веб-додатків та мікросервісних систем IAM-сервіс відповідає за виконання трьох ключових процесів: ідентифікації (визначення, ким користувач себе оголошує), автентифікації (підтвердження достовірності оголошеної ідентичності) і авторизації (визначення дозволених для користувача дій з ресурсами). [1]

Ідентифікація користувача в типових веб-системах виконується за допомогою унікального ідентифікатора — електронної адреси, імені користувача або зовнішнього профілю (Google, Microsoft, GitHub). Автентифікація може ґрунтуватися на одному, двох або кількох факторах, які традиційно поділяють на три категорії: те, що користувач знає (пароль, PIN-код), те, що користувач має (одноразовий пароль з апаратного або програмного токена, SMS-повідомлення, апаратний ключ), і те, чим користувач є (біометричні характеристики). Двофакторна автентифікація (2FA) поєднує елементи з двох різних категорій і значно знижує ризик компрометації облікового запису у разі викрадення пароля.

Авторизація після успішної автентифікації встановлює, які саме дії користувач може виконувати в системі. Найпоширенішими моделями авторизації у веб-додатках є рольовий контроль доступу (Role-Based Access Control, RBAC), у якому права визначаються через ролі користувача, та атрибутивний контроль (Attribute-Based Access Control, ABAC), що враховує контекст і атрибути запиту. RBAC є простішим у реалізації і повністю покриває потреби більшості бізнес-

додатків, тоді як ABAC застосовується у більш складних сценаріях з тонкою деталізацією прав. [2]

Основні процеси, які повинен забезпечувати IAM-сервіс, наведено на схемі (див. рис. 1.1). До них належать реєстрація користувача, верифікація електронної пошти, вхід з підтримкою 2FA, видача та оновлення короткоживучих access-токенів, відкликання сесій, скидання пароля, а також адміністративні операції: створення груп, призначення дозволів, перегляд журналу аудиту.

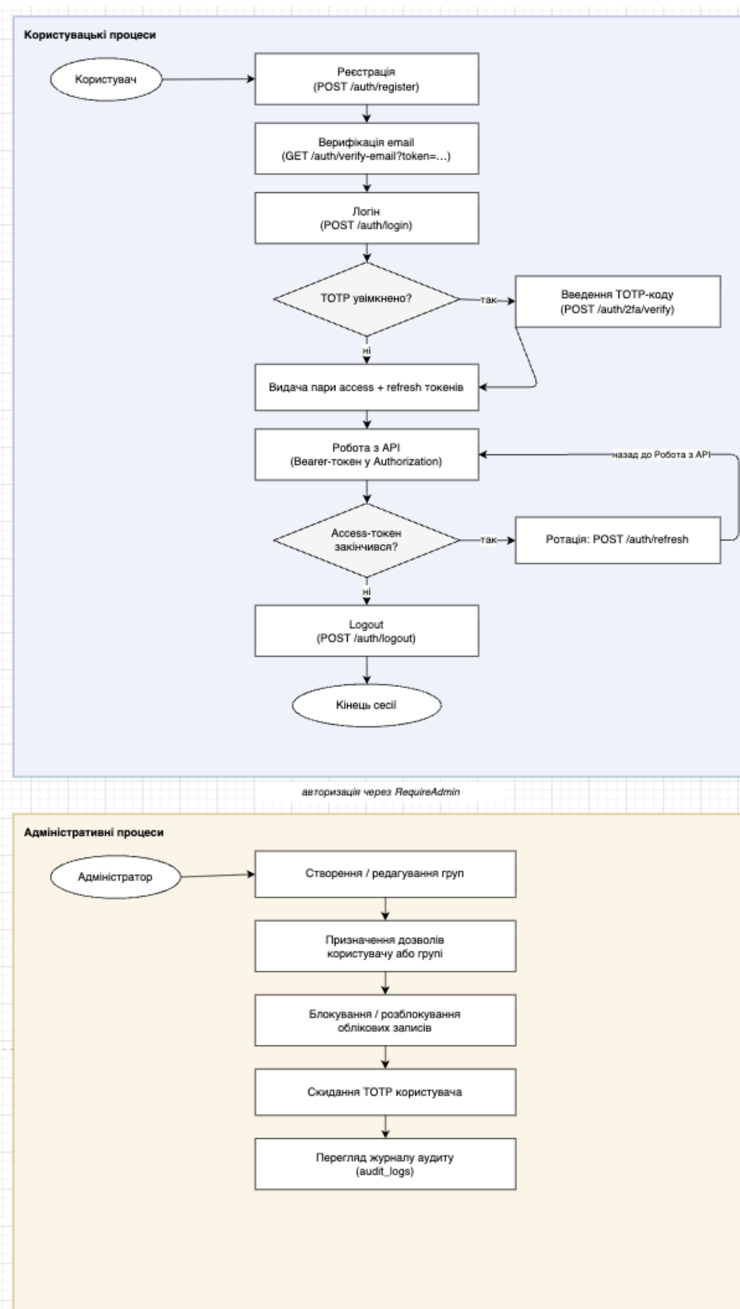

Рисунок 1.1 – Узагальнена схема процесів управління ідентичністю в IAM-сервісі

Сучасні стандарти та протоколи у сфері IAM забезпечують сумісність між системами різних виробників. До найважливіших належать OAuth 2.0 для делегованої авторизації, OpenID Connect як надбудова OAuth 2.0 для автентифікації, SAML 2.0 для корпоративного Single Sign-On (SSO), а також WebAuthn для безпарольної автентифікації за допомогою апаратних ключів. Для передачі автентифікаційних даних між сервісами де-факто стандартом стали JSON Web Tokens (JWT, RFC 7519), які являють собою компактні самодостатні токени, підписані секретним ключем або асиметричним алгоритмом.

Аналіз предметної області показує, що для побудови IAM-сервісу для веб- і мікросервісних систем малого та середнього розміру достатньо реалізувати модель RBAC з підтримкою груп користувачів, JWT-автентифікацію з ротацією refresh-токенів, двофакторну автентифікацію за стандартом TOTP (RFC 6238), та повний журнал аудиту дій. Додатково необхідно забезпечити захист від типових атак: брутфорсу пароля (через тимчасове блокування), масової реєстрації (rate limiting), компрометації токенів (через ротацію та короткий час життя access-токенів).

1.2. Огляд існуючих програмних рішень у сфері IAM

На ринку програмного забезпечення представлений широкий спектр готових рішень для управління ідентифікацією та доступом. Їх можна умовно розділити на три категорії: комерційні хмарні платформи Identity-as-a-Service (Auth0, Okta, AWS Cognito), відкриті повнофункціональні платформи (Keycloak, Authentik) та легковагові бібліотеки/компоненти (Ory Kratos, SuperTokens, NextAuth). Розглянемо найбільш відомі представники кожної категорії.

Keycloak — це провідне рішення з відкритим кодом для IAM, що розробляється компанією Red Hat. Платформа реалізована мовою Java з використанням сервера WildFly та підтримує OAuth 2.0, OpenID Connect, SAML, інтеграцію з LDAP/Active Directory, соціальні мережі, а також багатотенантність. Перевагами Keycloak є зрілість, велика спільнота, наявність готових інтеграцій з

більшістю популярних мов і фреймворків. Основними недоліками є висока ресурсоємність (мінімальні рекомендовані 1 ГБ оперативної пам'яті лише для самого сервера, не враховуючи СУБД), складність початкового налаштування, специфічна модель адміністрування через консоль і необхідність глибокого розуміння Java-екосистеми для розширення функціоналу. [4]

Auth0 є комерційною хмарною платформою компанії Okta, яка надає IAM як послугу (Identity-as-a-Service). Платформа підтримує всі сучасні стандарти автентифікації, має розвинений редактор правил на JavaScript для модифікації токенів і потоків автентифікації, а також велику бібліотеку готових інтеграцій. Основними недоліками є висока вартість при масштабуванні (тарифікація за активних користувачів на місяць), залежність від зовнішнього провайдера, потенційні складнощі з відповідністю вимогам про локалізацію персональних даних в межах конкретної юрисдикції.

Authentik — відносно нова відкрита платформа на мові Python, що позиціонує себе як легша альтернатива Keycloak. Підтримує OAuth 2.0, OpenID Connect, SAML, LDAP-проксі, має зручну веб-консоль і гнучку систему політик доступу на основі візуального конструктора. Authentik менш ресурсомісткий за Keycloak, проте все одно потребує розгортання кількох компонентів: серверу, воркера, бази PostgreSQL та Redis для черги задач.

Ory Kratos — легковаговий сервіс автентифікації на мові Go, який є частиною екосистеми Ory (разом з Hydra для OAuth 2.0 та Keto для авторизації). Kratos є API-first: не має власного інтерфейсу користувача та зосереджений виключно на автентифікаційних потоках. Це робить його гнучким у вбудовуванні, але підвищує складність повної інтеграції — потрібно окремо реалізувати UI та інші компоненти екосистеми. Інтерфейс адміністратора у Keycloak показано на рисунку (див. рис. 1.2) для порівняння з підходом Ory.

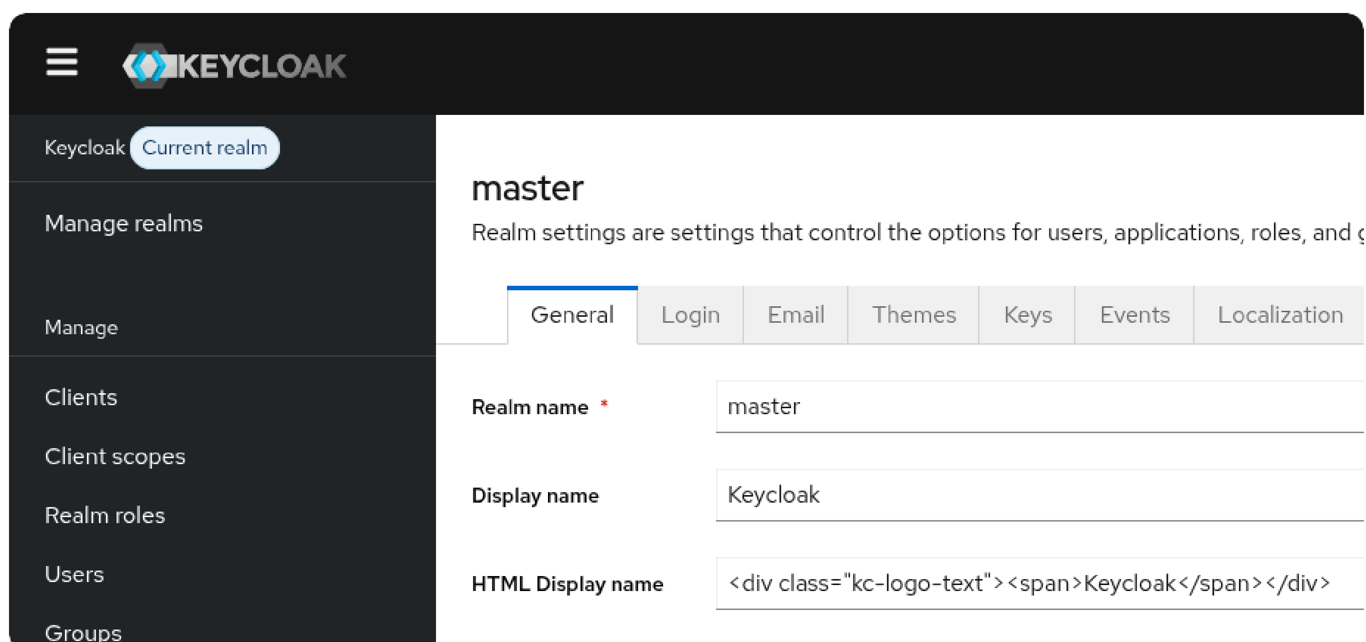

Рисунок 1.2 – Веб-інтерфейс адміністратора платформи Keycloak

Для систематизованого порівняння провідних рішень за ключовими критеріями — мовою реалізації, підтримкою стандартних протоколів, ресурсними вимогами та простотою розгортання — складено таблицю 1.1. Розроблюваний у рамках цієї кваліфікаційної роботи продукт «User Service» включено до таблиці для подальшого обґрунтування його місця серед існуючих рішень.

Таблиця 1.1 — Порівняння функціональних можливостей IAM-рішень

Критерій	**Keycloak**	**Auth0**	**Authentik**	**User Service**
Мова реалізації	Java	пропр.	Python	Go
Ліцензія	Apache 2.0	комерційна	MIT	відкрита
OAuth 2.0 / OIDC	+	+	+	частково
SAML 2.0	+	+	+	–
JWT (RFC 7519)	+	+	+	+
TOTP 2FA	+	+	+	+
RBAC	+	+	+	+

Групи користувачів	+	+	+	+
Журнал аудиту	+	+	+	+
REST API	+	+	+	+
Swagger / OpenAPI	частково	+	–	+
Адмінпанель	+	+	+	+
Мін. RAM (МБ)	1024	хмара	512	64
Простота розгортання	низька	висока	середня	висока
Кастомізація бізнес-логіки	середня	висока	середня	максимальна

Аналіз показує, що промислові платформи перевершують власне рішення за широтою функціональності (зокрема за наявністю SAML, OAuth 2.0 з усіма потоками, інтеграцією з LDAP), проте програють за ресурсними витратами та простотою розгортання. Власна розробка дає максимальну гнучкість у налаштуванні бізнес-логіки і не потребує тривалого вивчення консолі адміністрування.

1.3. Обґрунтування доцільності власної розробки

Аналіз існуючих рішень у попередньому підрозділі дозволяє зробити висновок, що жодне з них не є оптимальним для усіх сценаріїв використання. Промислові платформи на кшталт Keycloak забезпечують максимальну функціональну повноту, проте мають значні системні вимоги і потребують спеціалізованої експертизи для адміністрування. Хмарні рішення Auth0 та Okta усувають експлуатаційні складнощі, але формують додаткову залежність від зовнішнього провайдера і викликають питання щодо суверенітету персональних

даних. Легковагові компоненти на зразок Ory Kratos потребують додаткового написання коду для повноцінного використання. [5]

У рамках кваліфікаційної роботи поставлена задача розробки спеціалізованого, цілеспрямованого IAM-сервісу, який реалізує лише необхідний для більшості бізнес-проєктів набір функцій: реєстрацію та логін з паролем, ротацію JWT-токенів, RBAC з групами, двофакторну автентифікацію TOTP, аудит дій і керування сесіями. Такий підхід має кілька важливих переваг.

По-перше, мінімальний об'єм коду й залежностей значно спрощує розуміння системи розробниками, які вперше з нею працюють. Загальний обсяг проєкту становить близько 4500 рядків коду на мові Go без врахування адміністративної панелі. Для порівняння, Keycloak містить понад 600 тисяч рядків коду на Java, що ускладнює його розуміння та модифікацію.

По-друге, ресурсна ефективність мови Go дозволяє запускати сервіс на серверах з мінімальними вимогами: бінарний файл після компіляції займає близько 25 МБ, оперативної пам'яті в стані спокою достатньо 30–60 МБ. Це дає змогу розгортати сервіс на найдешевших VPS-планах або у обмежених середовищах виконання, наприклад у edge-кластерах.

По-третє, гнучкість у кастомізації бізнес-логіки. Власна реалізація дозволяє безпосередньо змінювати правила валідації, формат токенів, склад полів користувача, схему дозволів — без потреби у складних правилах, скриптах чи плагінах. Будь-яка модифікація стає звичайною зміною в коді з можливістю тестування, перегляду через систему контролю версій і простого відкату.

По-четверте, відсутність залежності від зовнішнього хмарного провайдера повністю усуває питання локалізації даних, оскільки всі дані залишаються у власній інфраструктурі організації. Це особливо актуально для державних, медичних, фінансових організацій та проєктів, що працюють з критичною інформаційною інфраструктурою згідно з вимогами Закону України «Про основні засади забезпечення кібербезпеки України».

Таким чином, розробка власного, компактного, але функціонально повного IAM-сервісу на мові Go є обґрунтованим вибором для широкого класу проєктів.

Подальші розділи кваліфікаційної роботи присвячені теоретичним основам реалізації такого сервісу та практичним аспектам його розробки.

2. ТЕОРЕТИЧНА ЧАСТИНА

2.1. Архітектурні принципи побудови HTTP-сервісів на мові Go

Сучасні HTTP-сервіси будуються переважно за архітектурним стилем REST (Representational State Transfer), запропонованим Роєм Філдінгом у його дисертації 2000 року. REST-сервіс розглядає функціональність системи як набір ресурсів, що ідентифікуються унікальними URI, а взаємодія з ресурсами здійснюється стандартними методами HTTP (GET, POST, PUT, PATCH, DELETE). Стиль REST передбачає відсутність стану на стороні сервера між запитами (stateless), уніфіковану структуру повідомлень, кешуваність відповідей та можливість шарування інфраструктури проксі-серверами.

Мова програмування Go має надзвичайно зручну стандартну бібліотеку для побудови HTTP-сервісів. Пакет net/http стандартної бібліотеки надає повний набір примітивів для роботи з HTTP: типи http.Request і http.Response, інтерфейс http.Handler з методом ServeHTTP, маршрутизатор http.ServeMux, готовий сервер http.Server з налаштуваннями таймаутів. Цього набору достатньо для реалізації базового сервісу, проте на практиці більшість проєктів використовують зовнішні маршрутизатори, які додають зручне групування маршрутів, шарові middleware та параметризовані шляхи. [6]

Серед популярних маршрутизаторів для Go виділяються chi, gin, echo та fiber. У межах цієї роботи обрано маршрутизатор chi (github.com/go-chi/chi версії 5.x), оскільки він є idiomatic Go-рішенням, повністю сумісним зі стандартним інтерфейсом http.Handler, не нав'язує власних типів і зберігає максимальну прозорість. Chi надає механізм груп маршрутів через метод Route, ланцюжки middleware через Use, параметризовані шляхи у форматі /users/{id}, а також підтримку класичного шаблону mounting для модульної організації API.

Принцип Clean Architecture, сформульований Робертом Мартіном у книзі однойменної назви, є одним із найпопулярніших підходів до структурування коду

серверних застосунків. Згідно з ним, код поділяється на концентричні шари, де внутрішні шари не знають про існування зовнішніх. Типова чотиришарова структура для HTTP-сервісу на Go включає шар сутностей (domain entities), шар прикладної логіки (use cases або service), шар адаптерів (presenters, controllers, gateways) та шар інфраструктури (frameworks та drivers).

Узагальнена схема шарування на основі Clean Architecture, адаптована для HTTP-сервісу на Go, наведена на рисунку (див. рис. 2.1). Стрілки на схемі позначають напрямок залежностей: внутрішні шари оголошують інтерфейси, які реалізують зовнішні, тому домен залишається повністю незалежним від конкретного фреймворку чи СУБД.

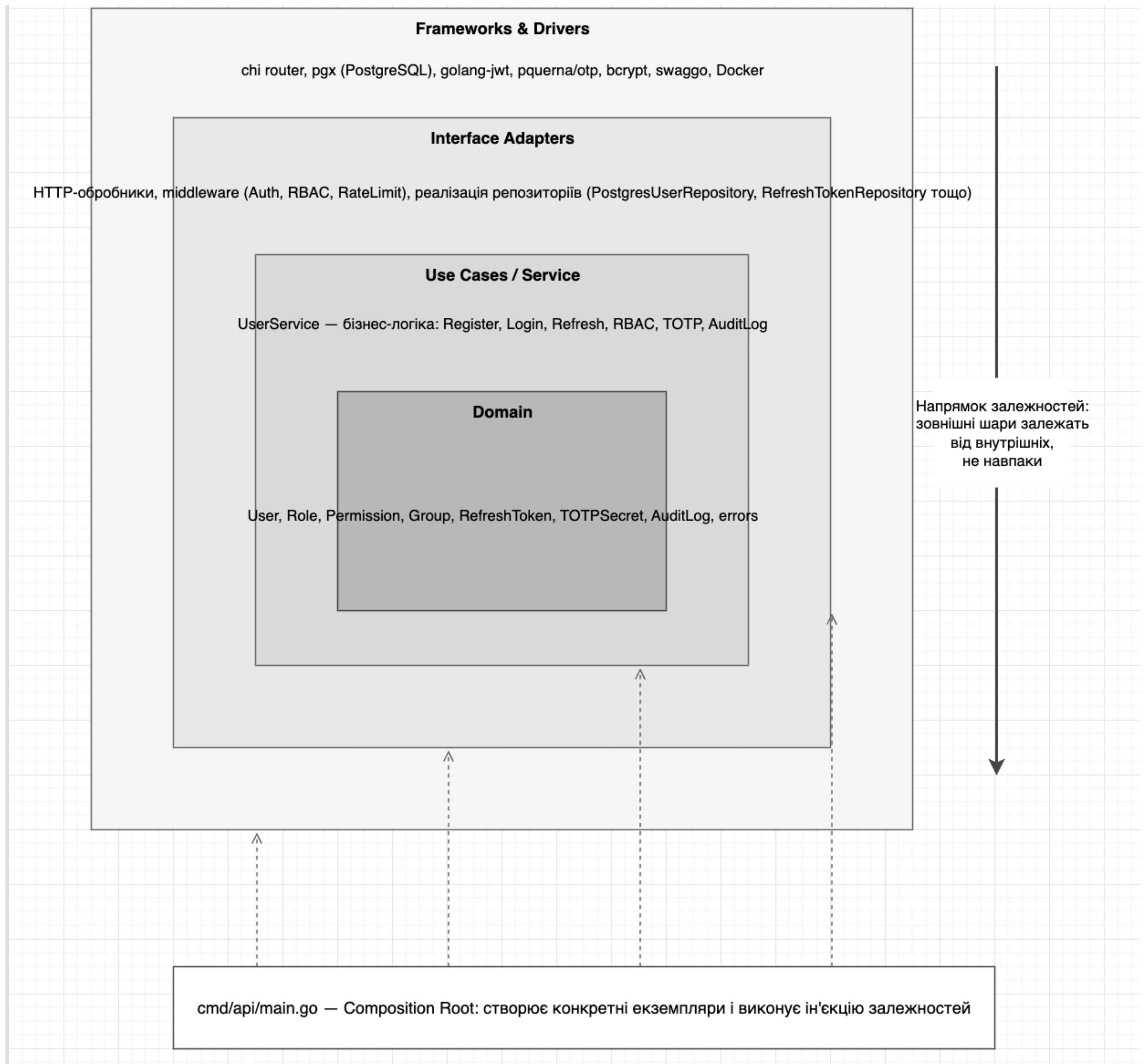

Рисунок 2.1 – Шари Clean Architecture у застосунку User Service

У реалізованій системі шарування виглядає таким чином: пакет internal/domain містить лише структури сутностей та інтерфейси (без жодних зовнішніх залежностей); пакет internal/repository оголошує інтерфейси сховищ, конкретна реалізація яких розташована у internal/repository/postgres; пакет internal/service містить бізнес-логіку (use cases) та залежить лише від домену та інтерфейсів репозиторіїв; пакет internal/handler містить HTTP-обробники і залежить від інтерфейсу сервісного шару; пакет internal/middleware реалізує наскрізну функціональність (автентифікація, журналювання, обмеження запитів). Точка входу

cmd/api/main.go виконує роль композиційного кореня — створює конкретні екземпляри репозиторіїв, ін'єктує їх у сервісний шар, ін'єктує сервіс у HTTP-обробники й запускає сервер. [7]

Такий підхід забезпечує тестованість бізнес-логіки (сервісний шар можна тестувати з мок-репозиторіями, не розгортаючи СУБД), легкість заміни компонентів (наприклад, перехід з PostgreSQL на MongoDB потребує написання нової реалізації репозиторію без зміни сервісного коду) і зрозумілість для нових розробників, оскільки у кожному пакеті містяться файли з чітко визначеною роллю.

2.2. Механізми автентифікації на основі JWT та ротація refresh-токенів

JSON Web Token (JWT, RFC 7519) є відкритим стандартом для безпечної передачі стверджень у формі JSON-об'єкта, підписаного криптографічним алгоритмом. JWT складається з трьох частин, розділених крапкою: заголовка (header), корисного навантаження (payload) і підпису (signature). Кожна частина закодована у форматі Base64URL. У заголовку зазначається алгоритм підпису (HS256, RS256, ES256 тощо), у корисному навантаженні — стверджування про користувача (так звані claims), а підпис формується шляхом застосування заданого алгоритму до конкатенації header.payload з використанням секретного ключа.

Сила JWT полягає у його самодостатності: сервер може перевірити автентичність токена і витягнути дані про користувача, не звертаючись до бази даних. Це робить JWT ідеальним для горизонтально масштабованих stateless-систем, у яких будь-який екземпляр сервісу може обробити будь-який запит без спільного стану. Стандарт визначає кілька зарезервованих claims, найважливіші з яких: iss (емітент токена), sub (суб'єкт, зазвичай ID користувача), exp (час закінчення дії у форматі Unix timestamp), iat (час видачі), aud (отримувач). Користувацькі claims (наприклад, role, permissions) можна додавати довільно. [8]

Основним недоліком JWT є складність відкликання вже виданих токенів, оскільки сервер за визначенням не зберігає стан про активні токени. Для розв'язання цієї проблеми застосовується схема двох токенів: коротkoживучого access-токена

(типово 15 хвилин — 1 година), який передається з кожним запитом і не може бути відкликаний достроково, і довгоживучого refresh-токена (типово 7–30 днів), який зберігається у базі даних і використовується лише для отримання нової пари токенів. У разі компрометації access-токена він має обмежений час дії, а у разі компрометації refresh-токена — він може бути миттєво відкликаний у БД.

У проєкті User Service застосовано додаткову техніку — ротацію refresh-токенів. Кожне використання refresh-токена для оновлення пари негайно інвалідує цей токен у базі даних і видає новий. Якщо зловмисник перехопив refresh-токен і використав його раніше, ніж легітимний користувач, тоді легітимна спроба зазнає невдачі (токен уже видалений) і користувач негайно дізнається про витік. Як додаткова міра, всі активні refresh-токени конкретного користувача можна примусово відкликати при підозрі на компрометацію.

У бібліотеці golang-jwt/jwt/v5, що використовується у проєкті, генерація access-токена виконується методами NewWithClaims та SignedString, а перевірка — функцією Parse, що повертає об'єкт *jwt.Token та помилку. Для верифікації підпису використовується замикання, яке надає секретний ключ і перевіряє, що алгоритм підпису належить до очікуваного класу (для HS256 — HMAC-SHA256). Це є важливим запобіжником від атаки «alg=none» та інших класичних атак на JWT. [9]

2.3. Модель контролю доступу RBAC і двофакторна автентифікація TOTP

Модель контролю доступу на основі ролей (Role-Based Access Control, RBAC) формалізована стандартом NIST RBAC у документах INCITS 359-2004. Ключова ідея полягає у тому, що права доступу пов'язуються не безпосередньо з користувачами, а з ролями, які можуть бути присвоєні одному чи кільком користувачам. Користувач отримує всі права, що належать призначеним йому ролям. Така модель спрощує адміністрування у системах з великою кількістю користувачів, оскільки зміна прав здійснюється у одній ролі і автоматично застосовується до всіх її носіїв.

Стандарт RBAC визначає чотири рівні зрілості моделі. Базовий рівень (Core RBAC) включає поняття користувача, ролі, дозволу та сесії. Ієрархічний (Hierarchical RBAC) додає можливість успадкування дозволів між ролями. Обмежувальний (Constrained RBAC) додає обмеження для розв'язання задачі розділення обов'язків (Separation of Duties). Симетричний (Symmetric RBAC) додає методи для огляду присвоєння дозволів. [10]

У реалізації User Service застосовано спрощений варіант між Core RBAC і Hierarchical RBAC: користувач має одну роль (user або admin), може мати індивідуальні дозволи (через таблицю user_permissions) і може бути включений у одну або кілька груп, від яких також успадковує дозволи. Загальний набір дозволів користувача формується об'єднанням індивідуальних дозволів і дозволів груп. Така модель забезпечує гнучкість, достатню для типових бізнес-сценаріїв, і водночас зберігає просту реалізацію.

Двофакторна автентифікація на основі одноразових паролів (Time-based One-Time Password, TOTP) описана у RFC 6238 і є розширенням протоколу HOTP (RFC 4226). Алгоритм генерує шестизначний код на підставі спільного секрету (shared secret) і поточного часу, заокругленого до 30-секундних інтервалів. Формула обчислення: HOTP(K, T), де K — секрет, T = floor(currentUnixTime / 30) — лічильник часу. Сама функція HOTP базується на HMAC-SHA1 і виокремленні чотирьох байтів результату за алгоритмом dynamic truncation.

Перевага TOTP полягає у відсутності необхідності зв'язку між сервером і клієнтом для генерації кодів — користувач формує коди в офлайн-застосунку (Google Authenticator, Microsoft Authenticator, Authy, FreeOTP), а сервер незалежно генерує очікуваний код за тим самим секретом і часом. Початкове встановлення секрету виконується через QR-код у форматі otpauth://, який сканується мобільним застосунком. Стандарт також передбачає допустимі відхилення часу: сервер традиційно приймає коди для попереднього і поточного 30-секундних вікон, що компенсує дрейф годинника та затримки введення. [11]

У проєкті використано бібліотеку pquerna/otp, яка повністю реалізує RFC 6238: метод totp.Generate створює секрет і повертає об'єкт Key, з якого можна

отримати otpauth-URL для генерації QR-коду; метод totp.Validate перевіряє наданий користувачем код проти секрету з налаштуванням допустимого вікна часу. Для генерації самого QR-коду застосовано бібліотеку boombuler/barcode, яка створює PNG-зображення з закодованим otpauth-URL.

2.4. PostgreSQL як сховище даних для IAM-сервісу

У ролі сховища даних для розроблюваного сервісу обрано реляційну СУБД PostgreSQL версії 16. PostgreSQL є найпотужнішою з відкритих реляційних СУБД, повністю відповідає стандарту SQL:2016, підтримує транзакції з повним ACID, має розширений набір вбудованих типів даних (включаючи UUID, JSON/JSONB, масиви, мережеві типи) і найкращу серед безкоштовних СУБД підтримку складних запитів з аналітичними функціями. Для IAM-сервісу важливими перевагами є вбудоване розширення pgcrypto з функцією gen_random_uuid() для генерації UUID на стороні бази, а також типи TIMESTAMPTZ з підтримкою часових поясів для коректного зберігання часових міток.

Для взаємодії з PostgreSQL з мови Go використано бібліотеку jackc/pgx версії 5.x — нативний драйвер з повноцінною підтримкою специфічних типів PostgreSQL. На відміну від драйверів через стандартний інтерфейс database/sql, pgx надає типобезпечні методи QueryRow та Exec з підтримкою параметризації запитів і автоматичного перетворення типів. Для управління пулом з'єднань застосовано тип pgxpool.Pool, який автоматично керує життєвим циклом з'єднань, обмежує їх кількість і перевикористовує при високому навантаженні. [12]

Для забезпечення керованого розвитку схеми бази даних впроваджено систему версіонування міграцій. Міграція являє собою пару SQL-файлів: up-міграція (накат) і down-міграція (відкат), що дає змогу будь-коли повернути базу до попереднього стану. У проєкті використано бібліотеку golang-migrate/migrate версії 4.x, яка зберігає поточну версію бази у спеціальній таблиці schema_migrations і автоматично застосовує усі непокачені файли при старті застосунку. Усього в проєкті розроблено десять міграцій від 000001_create_users до

000010_required_actions, що поетапно вводять структуру таблиць (див. розділ 3.2 для детального опису).

Безпека з'єднання з БД у промисловому використанні реалізується через TLS (параметр sslmode у DSN-рядку). У середовищі розробки застосовується режим disable, у промисловому — verify-full з перевіркою повного ланцюжка сертифікатів. Конфіденційні дані (паролі, секретні ключі, токени) зберігаються в БД у незворотно перетвореному вигляді: паролі — як bcrypt-хеш з адаптивним числом ітерацій, токени скидання пароля та верифікації — як випадкові 32-байтові послідовності, що генеруються джерелом криптографічно стійких випадкових чисел crypto/rand стандартної бібліотеки Go.

3. ПРАКТИЧНА ЧАСТИНА

3.1. Загальна архітектура системи та структура проєкту

Розроблений сервіс User Service являє собою монолітний серверний застосунок мовою Go, упакований у Docker-образ і запущений як окремий мікросервіс. Базою даних є PostgreSQL, розгорнута поряд із сервісом у тому самому файлі docker-compose.yml. Зовнішня взаємодія здійснюється через REST API на порту 8080 (всередині контейнера) і опційну адміністративну панель за шляхом /admin/. Інтерактивна документація API доступна за шляхом /swagger/. Загальна архітектура системи представлена на рисунку (див. рис. 3.1).

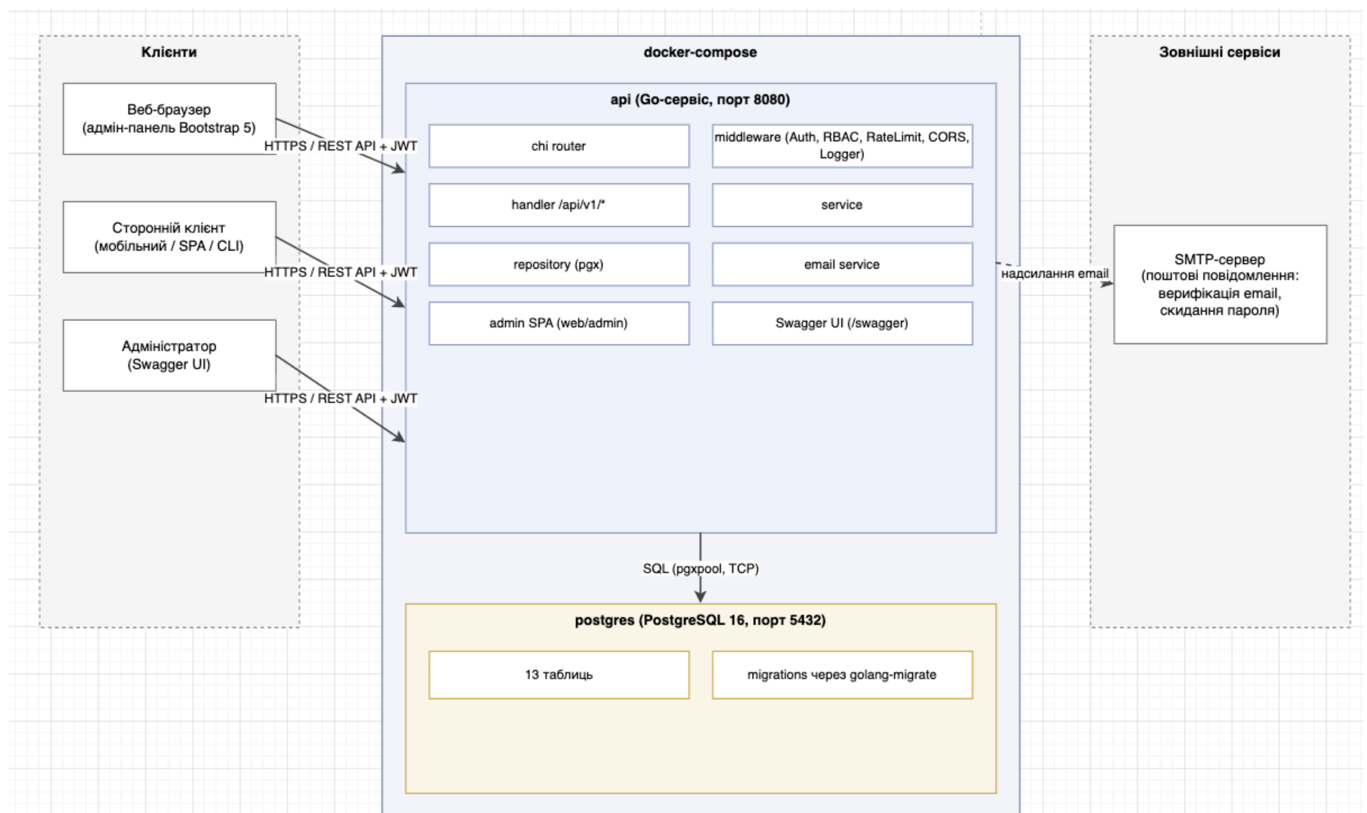

Рисунок 3.1 – Загальна архітектура системи User Service

Структура проєкту повністю відповідає рекомендаціям Standard Go Project Layout і принципам Clean Architecture, описаним у розділі 2.1. Папка cmd/api

містить точку входу main.go, що відповідає за ініціалізацію залежностей і запуск сервера. Папка internal приховує усю внутрішню реалізацію, недоступну зовнішнім споживачам пакета (за конвенцією Go). Усередині internal виокремлено такі пакети: domain (структури сутностей і помилки), repository та repository/postgres (інтерфейси сховищ і їх реалізація), service (бізнес-логіка), handler (HTTP-обробники), middleware (наскрізна функціональність), email (сервіс поштових повідомлень), config (завантаження конфігурації з ENV).

Папка migrations містить SQL-файли міграцій, які автоматично застосовуються при старті сервісу. Папка docs зберігає згенерований інструментом swag JSON-опис специфікації OpenAPI 3.0. Папка web/admin містить статичний HTML-файл адміністративної панелі. Зовнішніми залежностями проєкту є chi (маршрутизація), pgx (драйвер PostgreSQL), golang-migrate (міграції), golang-jwt (JWT), pquerna/otp (TOTP), boombuler/barcode (QR), bcrypt (хешування), godotenv (завантаження .env), swaggo/http-swagger (Swagger UI).

Конфігурація сервісу будується на змінних середовища: SERVER_PORT, DB_HOST, DB_PORT, DB_USER, DB_PASSWORD, DB_NAME, DB_SSLMODE, JWT_SECRET, JWT_EXPIRATION_HOURS, ADMIN_EMAIL, ADMIN_PASSWORD. Завантаження виконує функція config.Load з пакета internal/config, яка спершу робить спробу прочитати файл .env, а потім читає змінні з середовища, застосовуючи значення за замовчуванням для відсутніх параметрів. Такий підхід відповідає принципу 12-Factor App, що забезпечує незалежність застосунку від конкретного середовища виконання.

Послідовність ініціалізації застосунку у файлі cmd/api/main.go має чітко виражений каскад залежностей. Спочатку створюється логер на основі стандартного пакета log/slog із структурованим виведенням у форматі JSON. Далі завантажується конфігурація. Створюється пул з'єднань з БД (pgxpool.New), виконується перевірка зв'язку через Ping. Запускаються міграції БД через golang-migrate (м.Up). За потреби (якщо задано ADMIN_EMAIL та ADMIN_PASSWORD) виконується сидинг адміністраторського облікового запису. Створюються конкретні екземпляри репозиторіїв (NewUserRepository, NewRefreshTokenRepository

тощо) і сервісного шару (NewUserService). Створюється HTTP-обробник (handler.New) і роутер з ланцюжком middleware. Запускається HTTP-сервер з налаштованими таймаутами і обробкою сигналів SIGINT/SIGTERM для коректного завершення (graceful shutdown).

3.2. Проєктування схеми бази даних і система міграцій

Реляційна схема бази даних спроєктована як набір з тринадцяти таблиць, нормалізованих до третьої нормальної форми. Усі первинні ключі мають тип UUID і автоматично генеруються функцією gen_random_uuid() з розширення pgcrypto. Це усуває залежність від послідовностей (sequences), спрощує об'єднання даних із кількох систем та виключає ризик передбачення ID. Часові поля використовують тип TIMESTAMPTZ для коректної роботи з часовими поясами. ER-діаграма основних таблиць наведена на рисунку (див. рис. 3.2).

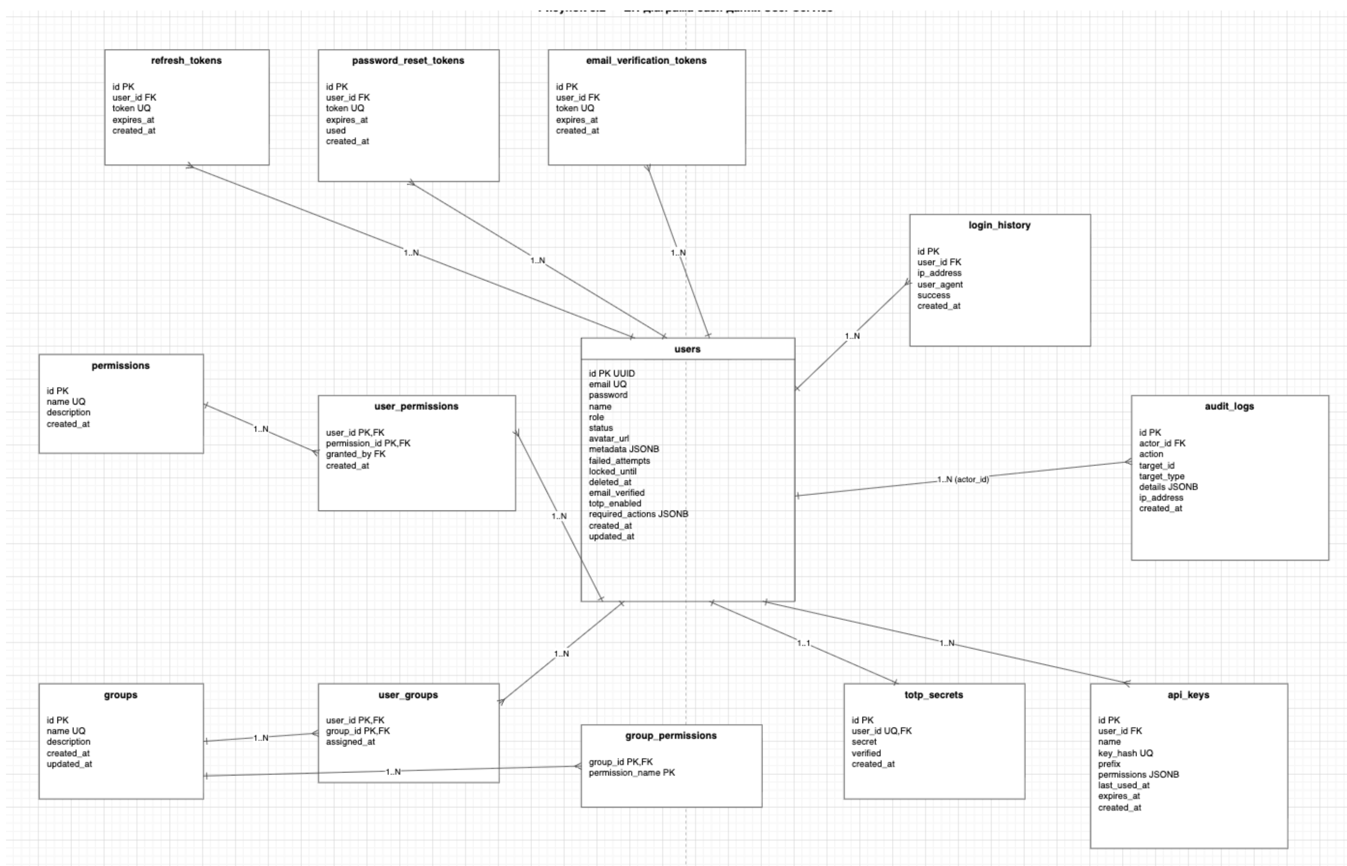

Рисунок 3.2 – ER-діаграма бази даних User Service

Центральною є таблиця users, яка зберігає основну інформацію про користувача: email, хеш пароля, ім'я, роль (user/admin), статус (active/blocked/deleted), URL аватара, метадані (JSONB-поле для довільних розширень), лічильник невдалих спроб входу, час блокування (locked_until), позначку м'якого видалення (deleted_at), прапор підтвердження email, прапор увімкнення TOTP, перелік обов'язкових дій (JSONB-масив). Усього таблиця має 16 колонок, що дозволяє повноцінно описувати стан користувача без необхідності в додаткових таблицях для часто запитуваних атрибутів.

Авторизаційні таблиці permissions, user_permissions, groups, user_groups, group_permissions реалізують модель RBAC, описану в підрозділі 2.3. Таблиця permissions містить заздалегідь визначений набір атомарних дозволів (наприклад, users:read, users:write, users:delete, audit:read), які можна призначити користувачу безпосередньо (через user_permissions) або через групу (через group_permissions). Композитний первинний ключ у user_permissions і user_groups гарантує унікальність зв'язку.

Окрему групу таблиць утворюють tokens-таблиці: refresh_tokens (довгоживучі токени для оновлення access-токенів), password_reset_tokens (одноразові токени для скидання пароля), email_verification_tokens (одноразові токени для підтвердження email). Усі вони мають індекс по полю token для швидкого пошуку, поле expires_at для позначення часу закінчення дії, поле user_id з зовнішнім ключем до users і каскадним видаленням при видаленні користувача.

Допоміжні таблиці включають login_history (історія входів з IP-адресою, User-Agent та прапором успіху), audit_logs (журнал критичних адміністративних дій з полями actor_id, action, target_id, target_type і JSONB-полем details для довільної додаткової інформації), totp_secrets (секрети двофакторної автентифікації, по одному на користувача через UNIQUE-обмеження на user_id) та api_keys (довгоживучі API-ключі з збереженням тільки SHA-256-хеша та публічного префіксу для пошуку).

Усі міграції розташовані у папці migrations і мають іменовий шаблон NNNNNN_назва.up.sql / NNNNNN_назва.down.sql. Бібліотека golang-migrate автоматично визначає поточну версію бази через таблицю schema_migrations і застосовує усі непокачені файли при старті. У проєкті розроблено десять кроків міграції: 000001 створює базову таблицю users, 000002 — refresh_tokens, 000003 розширює users допоміжними полями, 000004 додає auth-таблиці (login_history, токени скидання, токени верифікації, audit_logs), 000005 — RBAC-таблиці permissions та user_permissions, 000006 додає поле email_verified, 000007 створює таблиці груп, 000008 — TOTP, 000009 — API-ключі, 000010 — поле required_actions для зберігання обов'язкових дій (наприклад, обов'язкова зміна пароля при першому вході). Такий поетапний підхід дозволяє у будь-який момент відкотити структуру до конкретної версії, а у промисловому розгортанні — застосовувати міграції зі скриптів CI/CD.

3.3. Реалізація модуля автентифікації: реєстрація, вхід, JWT та refresh-токени

Модуль автентифікації реалізований у файлах internal/handler/auth.go, internal/service/user_service.go та internal/repository/postgres/refresh_token_repo.go. На рівні HTTP-маршрутів модуль обробляє ендпоінти /api/v1/auth/register, /auth/login, /auth/refresh, /auth/logout, /auth/forgot-password, /auth/reset-password, /auth/verify-email, /auth/resend-verification та /auth/2fa/verify. Усі маршрути об'єднані у chi-групу із застосуванням middleware AuthRateLimit, що дозволяє максимум 10 запитів на хвилину з однієї IP-адреси і захищає від брутфорсу пароля.

Процес реєстрації починається з валідації вхідних даних: формат email перевіряється регулярним виразом, довжина пароля повинна бути не менш ніж 8 символів, ім'я — не менш ніж 2 символи. У разі коректності даних метод service.Register генерує bcrypt-хеш пароля з фактором роботи за замовчуванням (10) і викликає user_repo.Create. Якщо у відповідь повертається помилка унікальності email, репозиторій повертає типізовану помилку domain.ErrEmailAlreadyTaken, яку обробник перетворює у HTTP-статус 409 Conflict. Після успішного створення

користувача в окремій горутині запускається асинхронна задача надсилання верифікаційного листа: генерується випадковий 32-байтовий токен, зберігається у таблиці email_verification_tokens з TTL 24 години, і викликається email-сервіс для надсилання повідомлення.

Послідовність обробки запиту входу графічно ілюструє діаграма послідовності (див. рис. 3.3). Метод service.Login виконує таку послідовність дій: спочатку шукає користувача за email, потім перевіряє його статус (повертає помилку, якщо користувач заблокований або у тимчасовій ізоляції після невдалих спроб), далі звіряє хеш пароля методом bcrypt.CompareHashAndPassword. У разі помилки збільшується лічильник невдалих спроб, і при досягненні порогу (5 спроб) обліковий запис блокується на 15 хвилин. У разі успіху лічильник скидається, і виконується перевірка, чи увімкнена двофакторна автентифікація. Якщо так, видається тимчасовий токен (temp_token) з обмеженим часом життя (5 хвилин), який клієнт повинен обміняти на повну пару токенів через ендпоінт /auth/2fa/verify. Якщо TOTP не увімкнений, негайно генерується пара access+refresh.

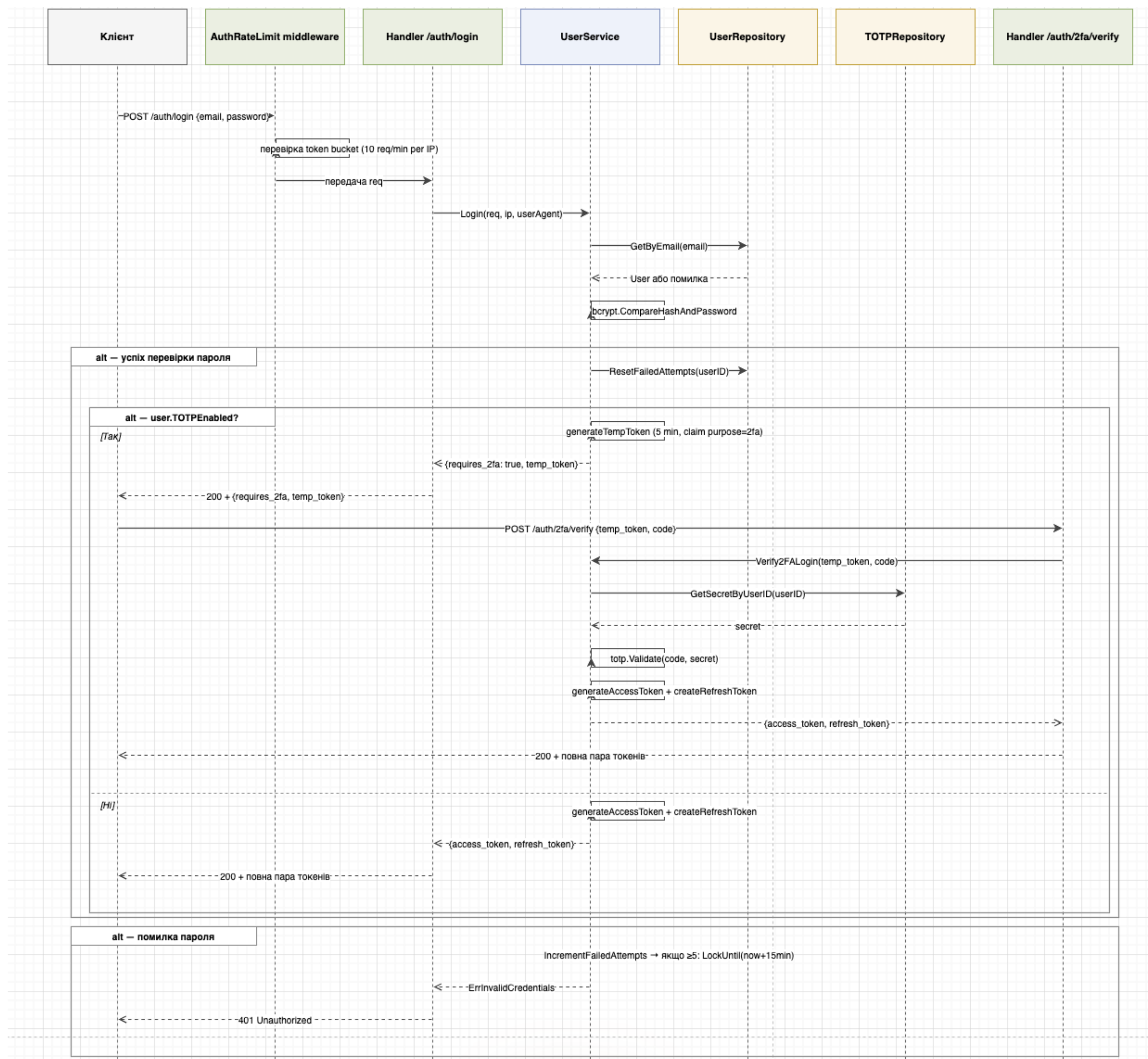

Рисунок 3.3 – Діаграма послідовності процесу входу з підтримкою TOTP

Генерація access-токена відбувається у методі generateAccessToken: створюються claims з полями user_id (UUID), role (user або admin), exp (поточний час плюс налаштована кількість годин), iat (час видачі). Токен підписується алгоритмом HMAC-SHA256 з використанням секретного ключа, заданого через ENV-змінну JWT_SECRET. Refresh-токен формується як випадкова 32-байтова послідовність, закодована у форматі hex, і зберігається в таблиці refresh_tokens з TTL 30 днів. У відповідь клієнту повертається об'єкт LoginResponse з полями access_token, refresh_token та опційними requires_2fa, temp_token.

Ротація refresh-токенів реалізована у методі service.Refresh. При надходженні запиту з refresh_token виконується пошук його в БД, перевірка терміну дії, отримання користувача, перевірка його статусу. У разі успіху старий токен негайно видаляється з БД, і генерується нова пара access+refresh. Це гарантує, що один refresh-токен може бути використаний рівно один раз, а будь-яка повторна спроба буде відкинута. Якщо зловмисник встиг скористатися токеном раніше за легітимного користувача, легітимна спроба провалиться, що сигналізує про компрометацію.

Скидання пароля реалізує безпечну процедуру з використанням одноразових токенів. На запит /auth/forgot-password сервер завжди повертає HTTP-статус 202 Accepted (незалежно від наявності користувача з таким email), що запобігає атаці перебору наявних email-адрес (user enumeration attack). Якщо користувач існує, генерується токен, зберігається у password_reset_tokens з TTL 1 година, і надсилається лист зі спеціальним посиланням. Метод service.ResetPassword перевіряє валідність токена, його статус Used, термін дії, оновлює пароль (з новим bcrypt-хешем) і помічає токен як використаний (used = true), що унеможливлює його повторне застосування.

3.4. Модуль контролю доступу RBAC: ролі, дозволи та групи

Модуль контролю доступу побудований навколо двох механізмів: грубої перевірки за роллю та тонкої перевірки за дозволами. Перший рівень — middleware RequireAdmin — застосовується до маршрутів адміністративної частини /api/v1/admin/* і відсіює запити від користувачів з роллю user одразу на рівні маршрутизатора. Другий рівень — підсистема дозволів — використовується усередині бізнес-логіки для тонкого контролю окремих операцій.

У базі даних дозволи зберігаються у таблиці permissions як унікальні рядкові ідентифікатори у форматі ресурс:дія, наприклад users:read, users:write, users:delete, audit:read. У початковій міграції створюється чотири базових дозволи, але адміністратор може додавати власні через адміністративний інтерфейс. Дозволи

призначаються користувачу одним з двох способів: безпосередньо (запис у таблиці user_permissions з посиланнями на user_id та permission_id) або через групу (користувач включений у групу, групі надано дозвіл через group_permissions). Загальна структура схеми RBAC показана на рисунку (див. рис. 3.4).

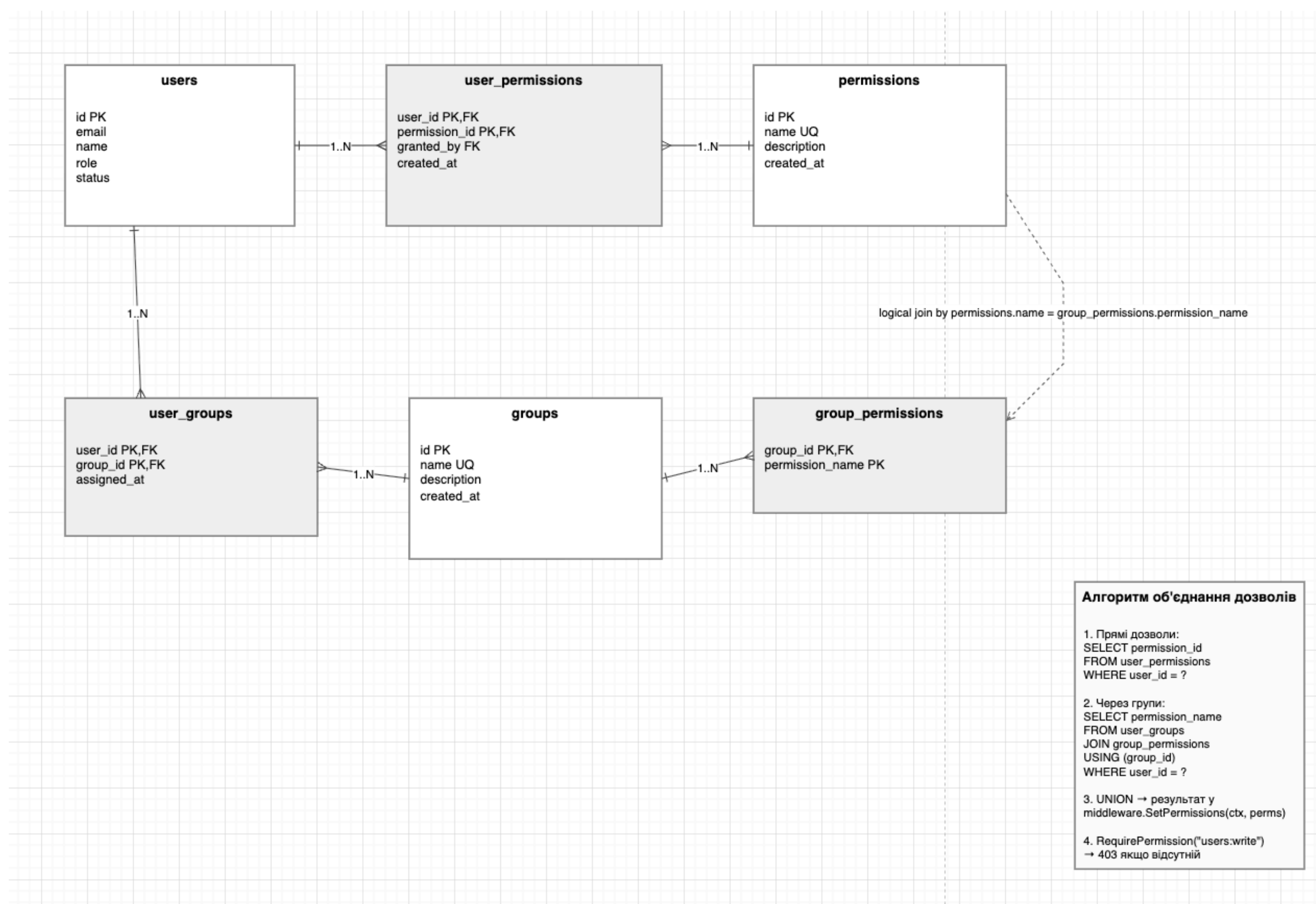

Рисунок 3.4 – Схема таблиць RBAC: користувачі, групи, дозволи

У момент успішної автентифікації, у середовищі middleware Auth, виконується завантаження повного списку дозволів користувача через метод service.GetUserStatus. Метод об'єднує (UNION у SQL) індивідуальні дозволи (через JOIN таблиць user_permissions і permissions) та дозволи всіх груп, до яких належить користувач (через JOIN user_groups і group_permissions). Отриманий список зберігається у контексті запиту через middleware.SetPermissions і доступний у будь-якому обробнику через виклики middleware.HasPermission або middleware.RequirePermission.

Допоміжна функція middleware.RequirePermission(perm) приймає рядок дозволу і повертає middleware-функцію, яка перевіряє його наявність у контексті запиту. Якщо дозвіл відсутній, повертається HTTP-статус 403 Forbidden з повідомленням про конкретний відсутній дозвіл. Це дає змогу декларативно захищати маршрути, наприклад: r.With(middleware.RequirePermission("users:write")).Patch("/users/{id}", h.UpdateUser).

Адміністративні операції з RBAC доступні через ендпоінти /admin/groups (CRUD над групами), /admin/groups/{id}/users/{userID} (додавання/видалення користувача з групи), /admin/groups/{id}/permissions/{perm} (надання/відкликання дозволу групі), /admin/users/{id}/permissions (перегляд та керування індивідуальними дозволами). У відповідь повертаються типізовані об'єкти Group, GroupResponse, PermissionResponse, що мають поля id, name, description, permissions та updated_at для зручної серіалізації.

3.5. Двофакторна автентифікація на основі TOTP

Двофакторна автентифікація у проєкті реалізована як опціональний шар захисту, який користувач увімкнує самостійно у власному профілі. Послідовність налаштування передбачає три кроки. На першому користувач викликає ендпоінт POST /api/v1/users/me/2fa/setup, у відповідь сервер генерує новий TOTP-секрет, зберігає його у таблиці totp_secrets з прапором verified=false, і повертає об'єкт TOTPSetupResponse з полями secret (текстове представлення), otpauth_url (повний URL у форматі otpauth://) та qr_code_image (PNG-зображення QR-коду, закодоване у Base64).

На другому кроці користувач сканує QR-код за допомогою сумісного мобільного застосунку (Google Authenticator, Microsoft Authenticator, Authy, FreeOTP). Застосунок реєструє акаунт і починає генерувати шестизначні коди з періодом 30 секунд. Користувач вводить поточний код у форму і викликає ендпоінт POST /users/me/2fa/confirm з кодом у тілі запиту. Сервер перевіряє код через метод

totp.Validate(code, secret), і у разі успіху встановлює прапор verified=true та totp_enabled=true для користувача. Інтерфейс налаштування TOTP в адмінпанелі показано на рисунку (див. рис. 3.5).

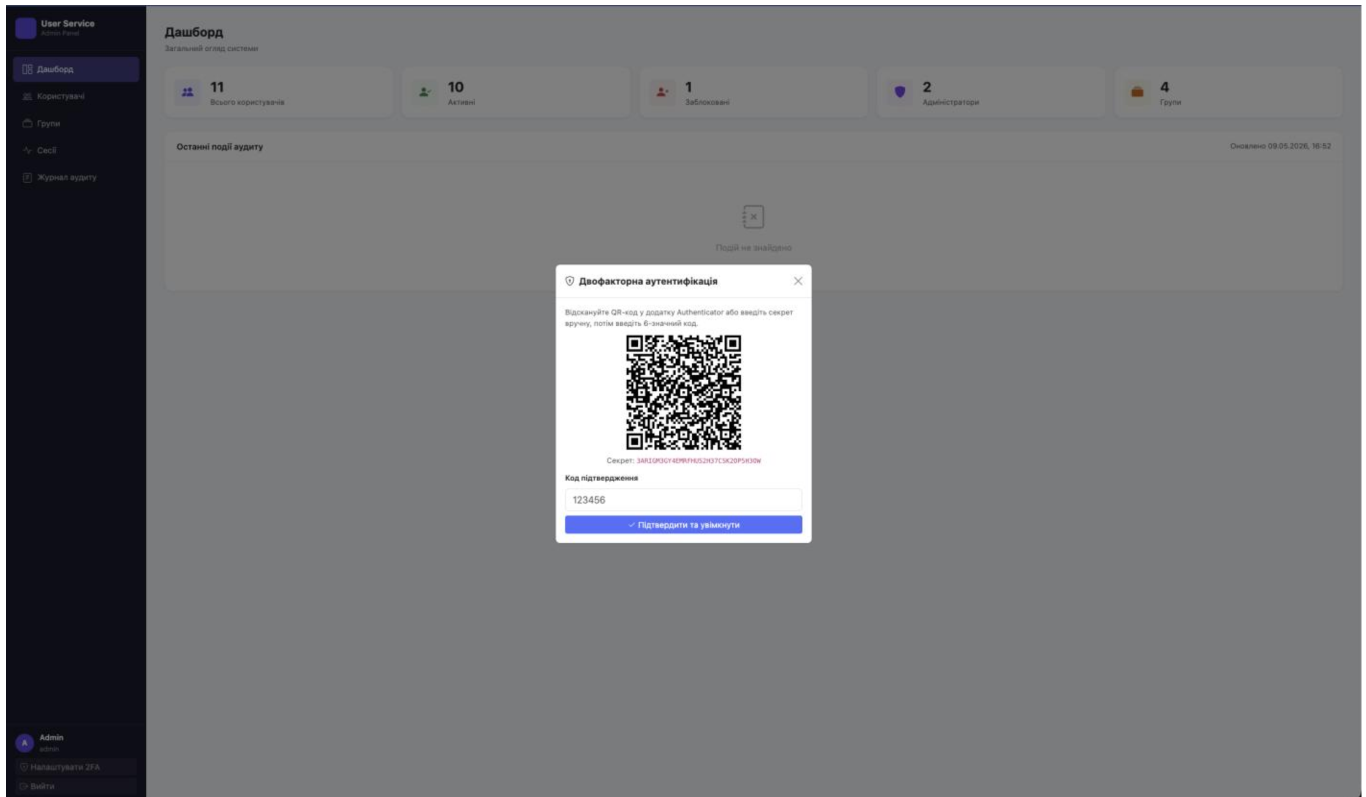

Рисунок 3.5 – Налаштування двофакторної автентифікації TOTP

На третьому кроці, при наступному вході, після успішної перевірки пароля, сервіс виявляє увімкнений TOTP і повертає лише тимчасовий токен (temp_token) разом з прапором requires_2fa=true. Клієнт має звернутися до ендпоінта POST /auth/2fa/verify з temp_token і кодом TOTP, щоб отримати повну пару access+refresh. Тимчасовий токен підписаний тим самим секретом JWT_SECRET, що й access-токен, але має дуже короткий час життя (5 хвилин) та містить спеціальне поле claim "purpose":"2fa", яке унеможливлює його використання для звичайних API-запитів.

Адміністратор має можливість примусово скинути TOTP для користувача (наприклад, у разі втрати телефону) через ендпоінт PATCH /admin/users/{id}/2fa/reset. Дія записується у журнал аудиту з action="2fa_reset", target_id=user_id та actor_id=admin_id, що забезпечує повний прозорий аудит критичних безпекових операцій.

3.6. Управління сесіями, API-ключами, аудит дій та обмеження запитів

Цей підрозділ об'єднує опис кількох тісно пов'язаних безпекових механізмів: керування активними сесіями користувача, видачу довгоживучих API-ключів для машинної інтеграції, журнал аудиту критичних дій та обмеження частоти запитів (rate limiting).

Активна сесія користувача у системі ототожнюється з активним refresh-токеном у таблиці refresh_tokens. Користувач може переглянути список своїх активних сесій через ендпоінт GET /users/me/active-sessions, що повертає масив з полями id, created_at, expires_at та частковою інформацією про пристрій. Окрему сесію можна відкликати через DELETE /users/me/active-sessions/{id}, або всі одразу через DELETE /users/me/active-sessions. Адміністратор має додаткові ендпоінти для перегляду та відкликання сесій будь-якого користувача: GET /admin/sessions, DELETE /admin/sessions/{id}, GET /admin/users/{id}/sessions.

API-ключі реалізовані як альтернативний механізм автентифікації для серверних інтеграцій, де неможливо використовувати ротацію JWT (наприклад, у CI/CD-скриптах). Користувач створює ключ через POST /users/me/api-keys, вказуючи ім'я та опційно термін дії й список дозволів. Сервер генерує випадкову 32-байтову послідовність, формує строку у форматі prefix_hex, де prefix — короткий публічний префікс (8 символів), а hex — основна частина. У БД зберігається лише SHA-256 хеш повного ключа і публічний префікс. Сирий ключ повертається користувачу один раз і більше ніколи не може бути отриманий — це запобігає його витоку через дамп БД. Перевірка ключа на вході через middleware APIKeyAuth (заголовок X-API-Key) виконується пошуком за префіксом і верифікацією повного хешу.

Журнал аудиту фіксує усі критичні адміністративні дії: блокування/розблокування користувача, зміну ролі, видалення, надання/відкликання дозволу, скидання TOTP, створення груп, видалення сесій. Кожен запис у таблиці audit_logs містить actor_id (хто виконав дію), action (тип дії),

target_id (над ким), target_type (тип цільового об'єкта), details (JSONB з додатковою інформацією — наприклад, попереднім та новим значенням), ip_address (IP-адреса виконавця) та created_at. Адміністратор переглядає журнал через ендпоінт GET /admin/audit-logs з фільтрами за actor_id, action, target_id та діапазоном дат, та з пагінацією. Приклад запису з журналу аудиту наведений на рисунку (див. рис. 3.6).

Рисунок 3.6 – Перегляд журналу аудиту в адмінпанелі

Обмеження частоти запитів реалізоване у пакеті middleware/ratelimit.go за алгоритмом Token Bucket. Для кожної IP-адреси підтримується відомість про кількість токенів і час останнього оновлення. При надходженні запиту обчислюється кількість токенів, що поповнилися з моменту останнього звернення (зі швидкістю rate токенів на секунду), і якщо у відомості є хоча б один токен, він списується і запит пропускається. У сервісі діють два обмежувачі: глобальний (60 запитів на хвилину з однієї IP) і авторизаційний (10 запитів на хвилину для маршрутів /auth/*). Останнє суттєво ускладнює атаки брутфорсу пароля. Для очищення від давно неактивних відомостей запускається фонова горутина, яка раз на 5 хвилин видаляє записи старші за 10 хвилин.

3.7. Адміністративна панель та документація API

Адміністративна панель реалізована як односторінковий веб-додаток (Single Page Application) на чистій мові JavaScript без використання важких фреймворків. Файл web/admin/index.html містить близько 1580 рядків розмітки, стилів та логіки і розгортається статично з тієї самої HTTP-сервісу через FileServer-обробник за шляхом /admin/. Стилізація виконана з використанням CSS-фреймворку Bootstrap 5 та набору іконок Bootstrap Icons. Загальний вигляд адміністративної панелі представлений на рисунку (див. рис. 3.7).

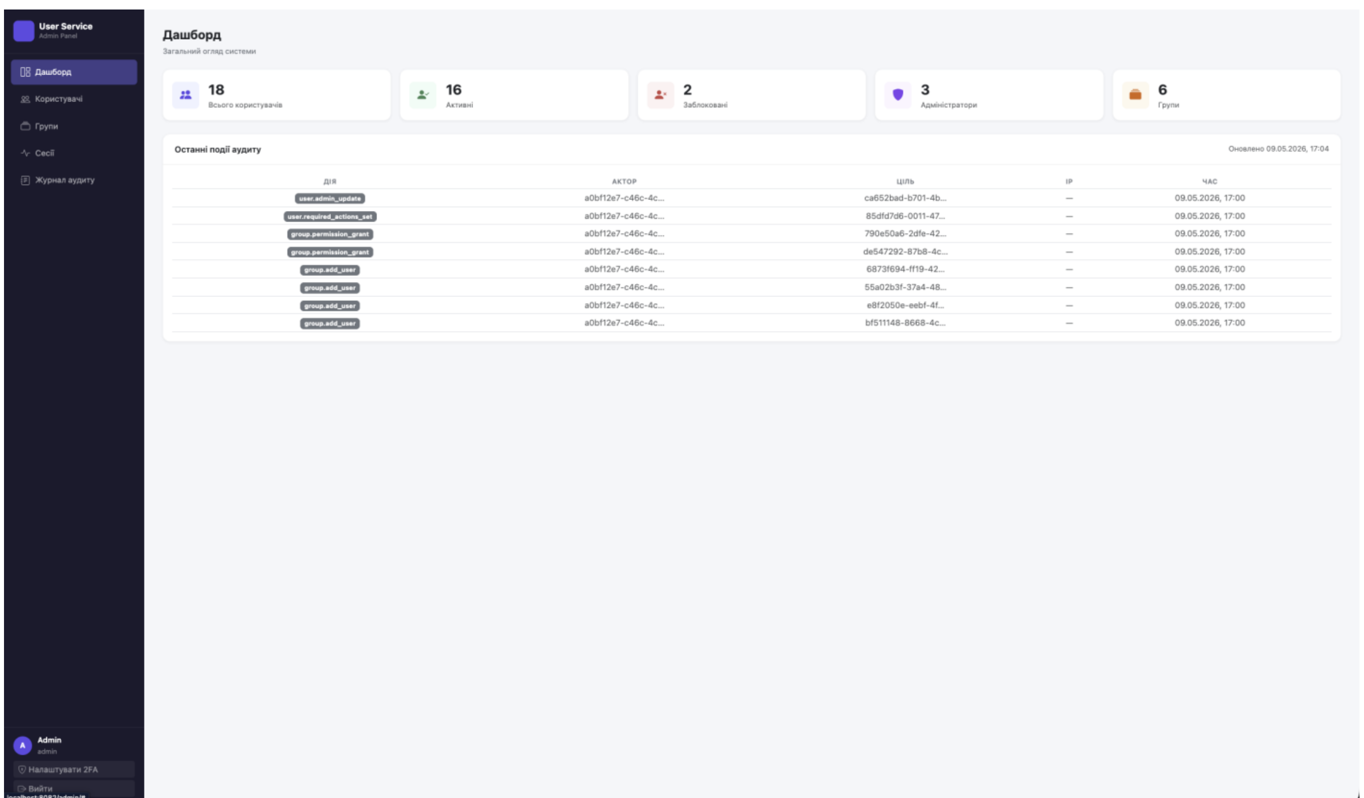

Рисунок 3.7 – Загальний вигляд адміністративної панелі User Service

Архітектурно адмінпанель складається зі статичного навігаційного меню зліва (sidebar) і динамічного контенту справа (main-content). Перемикання між розділами «Користувачі», «Групи», «Дозволи», «Сесії», «Аудит», «Профіль» виконується через JavaScript-роутер, який слухає зміни хешу адресного рядка (window.location.hash) і завантажує відповідний шаблон. Усі взаємодії з API

виконуються через єдину функцію api(method, path, body), що автоматично додає заголовок Authorization з токеном з localStorage та обробляє відповіді з кодами 401 (виконує перенаправлення на форму входу) і 403 (показує повідомлення про відсутність прав).

Розділ керування користувачами підтримує перегляд, пошук, фільтрацію за роллю та статусом, посторінкову навігацію (таблична компонента з 20 записів на сторінку), деталізований перегляд профілю, редагування полів, блокування і розблокування, зміну ролі, видалення з підтвердженням, перегляд історії входів та індивідуальних дозволів конкретного користувача. Розділ груп дозволяє створювати, редагувати, видаляти групи, додавати до них користувачів і призначати дозволи через візуальний компонент із чек-боксами. Журнал аудиту відображається у вигляді таблиці з можливістю фільтрації за датою, типом дії та виконавцем.

Документація API згенерована автоматично з коментарів у коді за допомогою інструмента swag (github.com/swaggo/swag). Кожен HTTP-обробник містить блок коментарів з директивами @Summary, @Description, @Tags, @Accept, @Produce, @Param, @Success, @Failure, @Router, @Security. Команда swag init -g cmd/api/main.go --output docs обходить весь проєкт, парсить коментарі, генерує JSON-файл специфікації OpenAPI 3.0 у docs/swagger.json і go-файл docs/docs.go, який вбудовується у бінарний файл застосунку. Сама інтерактивна сторінка Swagger UI монтується через httpSwagger.Handler за шляхом /swagger/. Інтерфейс Swagger UI показано на рисунку (див. рис. 3.8).

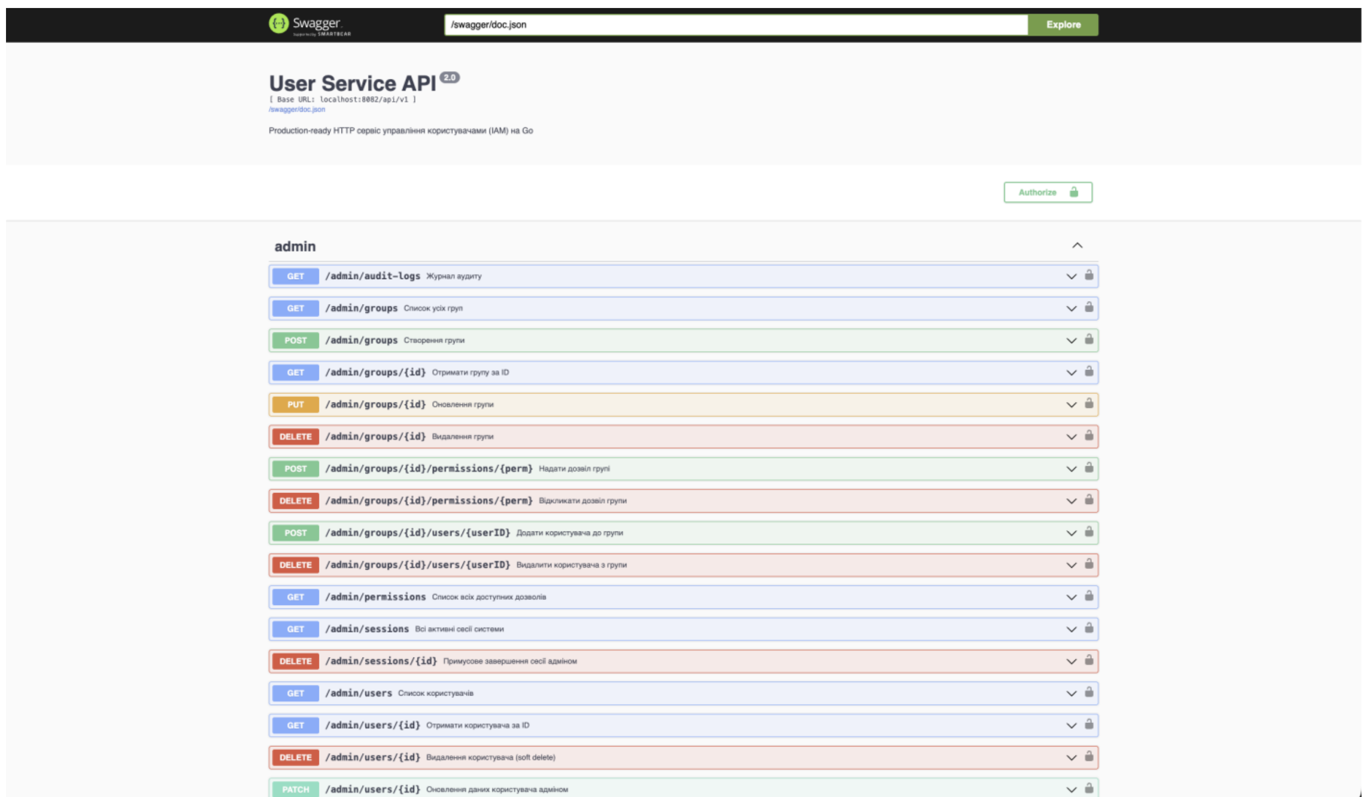

Рисунок 3.8 – Інтерактивна документація API на основі Swagger UI

У документації згруповано усі ендпоінти за тегами (auth, users, admin), для кожного вказано параметри тіла запиту з прикладами, можливі коди відповіді, схему даних відповіді. Розділ Authorize дозволяє ввести Bearer-токен один раз і виконувати усі захищені запити безпосередньо з браузера. Це значно прискорює тестування і дозволяє розробникам клієнтських застосунків випробовувати взаємодію без необхідності користуватися окремими інструментами на кшталт Postman.

3.8. Інструкція для користувача та системного адміністратора

Розгортання сервісу виконується за допомогою файлу docker-compose.yml у корені проєкту. У файлі описано два сервіси: postgres (СУБД на образі postgres:16-alpine з прив'язкою порту 5433 до 5432 контейнера для уникнення конфлікту з локальним PostgreSQL) і api (зборка з Dockerfile проєкту на порту 8082). Усі необхідні змінні середовища встановлюються секцією environment, включаючи дані

для підключення до БД, секретний ключ JWT і параметри першого адміністраторського облікового запису.

Для запуску сервісу системний адміністратор виконує команду docker compose up -d у корені проєкту. Контейнер бази даних запускається першим (із healthcheck pg_isready), а сервіс api стартує після успішного healthcheck бази. При першому запуску виконуються усі міграції БД, і за наявності змінних ADMIN_EMAIL та ADMIN_PASSWORD створюється акаунт адміністратора. Перевірити стан можна командою docker compose logs -f api.

Користувач взаємодіє з системою через адміністративну панель за адресою http://host:8082/admin/ або через REST API. Для звичайного користувача типовий сценарій початку роботи такий: реєстрація через форму на /admin/#register, отримання верифікаційного листа на email, перехід за посиланням з листа для активації акаунта, вхід через /admin/#login. Після успішного входу відкривається дашборд з можливістю перегляду власного профілю, керування активними сесіями, налаштування двофакторної автентифікації та керування API-ключами. Інтерфейс реєстрації показано на рисунку (див. рис. 3.9).

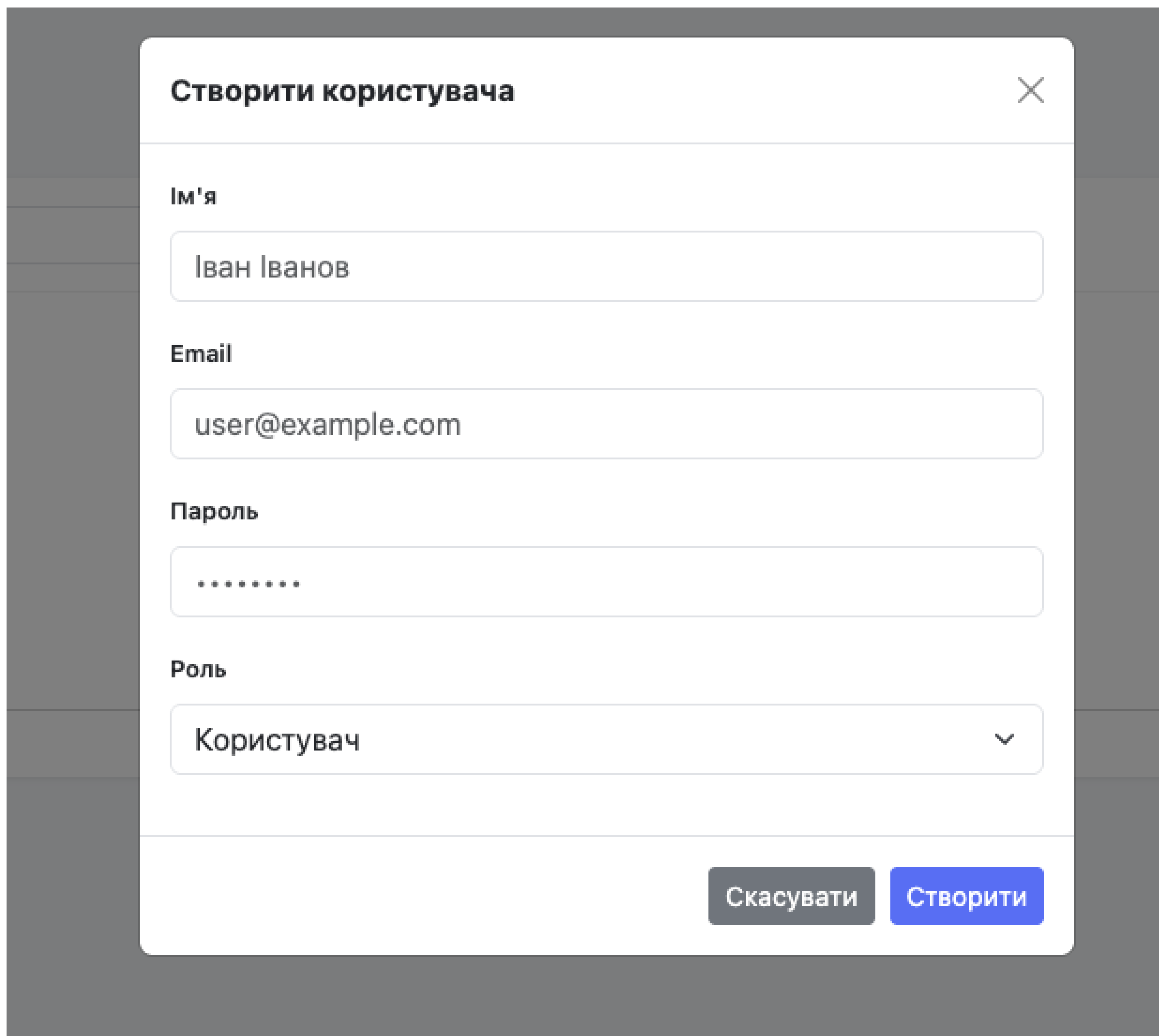

Рисунок 3.9 – Створення нового користувача в адміністративній панелі

Адміністратор додатково має доступ до розділів управління користувачами, групами та журналу аудиту. Типові робочі задачі адміністратора включають: створення нової групи з певним набором дозволів, додавання користувачів до групи, призначення індивідуальних дозволів окремим користувачам, перегляд історії входів і журналу аудиту, скидання двофакторної автентифікації для користувача, який втратив телефон, тимчасове блокування підозрілих облікових записів. Адмінська панель з активним користувачем у статусі заблокованого показано на рисунку (див. рис. 3.10).

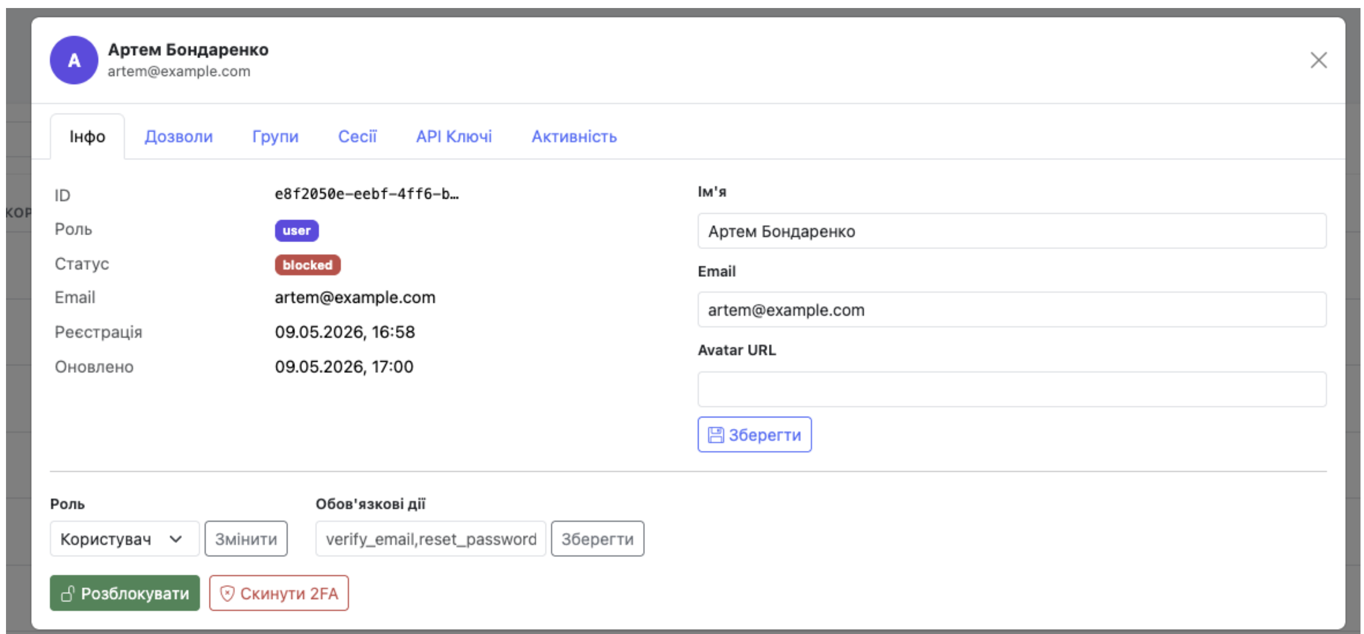

Рисунок 3.10 – Робота з заблокованим користувачем в адмінпанелі

Резервне копіювання бази даних виконується стандартними засобами PostgreSQL: pg_dump для створення дампа і pg_restore для відновлення. У типовому промисловому розгортанні дампи створюються щодня скриптом за розкладом cron і вивантажуються до зовнішнього сховища (S3-сумісного об'єктного сховища). Оновлення сервісу до нової версії виконується послідовністю docker compose pull, docker compose up -d, при цьому міграції застосовуються автоматично при першому старті нової версії.

Тестування коректності розгортання після оновлення передбачає перевірку трьох ключових ендпоінтів: GET /health (повертає {"status":"ok"} і HTTP 200), GET /swagger/ (відкриває інтерактивну документацію), POST /api/v1/auth/login з даними адміністратора (повертає валідну пару токенів). Завдяки модульному тестуванню сервісного шару (файл internal/service/user_service_test.go з 14 тест-кейсами) переважна більшість регресій виявляється ще на етапі CI до розгортання у промисловому середовищі.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи поставлена мета — створення серверної архітектури HTTP-сервісу користувачів на основі мови програмування Go, що забезпечує повний цикл управління ідентифікацією та доступом, — повністю досягнута. Розроблений програмний продукт User Service являє собою функціонально повний, безпечний і готовий до промислового використання серверний компонент для управління ідентичністю.

У результаті виконання кваліфікаційної роботи отримано такі основні результати:

- проведено системний аналіз предметної області управління ідентифікацією та доступом, досліджено типові процеси автентифікації й авторизації, основні стандарти у сфері IAM (OAuth 2.0, OpenID Connect, JWT, TOTP, SAML, RBAC) та їх практичне застосування у сучасних веб-сервісах;
- виконано порівняльний аналіз існуючих рішень у сфері IAM, складено таблицю порівняння за п'ятнадцятьма критеріями для платформ Keycloak, Auth0, Authentik та власної розробки, обґрунтовано доцільність побудови компактного спеціалізованого IAM-сервісу для класу малих і середніх проєктів;
- досліджено теоретичні засади побудови REST API на мові Go, принципи Clean Architecture, механізми JWT-автентифікації з ротацією refresh-токенів, моделі контролю доступу RBAC та алгоритм TOTP за стандартом RFC 6238;
- спроєктовано тришарову архітектуру системи з чітким розділенням обов'язків між шарами доменних моделей, репозиторіїв, сервісів та HTTP-обробників;
- спроєктовано і реалізовано реляційну схему бази даних PostgreSQL з тринадцяти таблиць у третій нормальній формі, побудовано систему версіонування міграцій з десяти кроків;
- реалізовано модуль автентифікації з підтримкою реєстрації, входу, ротації refresh-токенів, верифікації електронної пошти та скидання пароля з

безпечним захистом від атаки перебору email-адрес;

- реалізовано модуль контролю доступу RBAC з підтримкою ролей, індивідуальних дозволів та груп з успадкуванням прав, що забезпечує гнучкість, достатню для типових бізнес-сценаріїв;
- реалізовано двофакторну автентифікацію за стандартом TOTP з повною сумісністю з мобільними застосунками Google Authenticator та Microsoft Authenticator;
- реалізовано модулі керування активними сесіями користувача, видачі довгоживучих API-ключів, журнал аудиту критичних дій та обмеження частоти запитів за алгоритмом token bucket;
- розроблено адміністративну панель у вигляді односторінкового веб-додатку на основі Bootstrap 5, що дозволяє візуально керувати користувачами, групами, дозволами та переглядати журнал аудиту;
- згенеровано інтерактивну документацію API на основі специфікації OpenAPI 3.0 за допомогою інструмента swag з автоматичним парсингом коментарів у Go-коді;
- підготовлено конфігурацію Docker і docker-compose для розгортання сервісу разом із базою даних, написано модульні тести сервісного шару;
- проведено функціональне тестування усіх реалізованих ендпоінтів API через інтерактивну документацію Swagger UI та через адміністративну панель.

Розроблений сервіс має ряд практичних переваг: компактний розмір кодової бази (близько 4500 рядків коду на мові Go), низькі ресурсні вимоги (близько 30–60 МБ оперативної пам'яті при типовому навантаженні), повна підтримка контейнеризації, автоматичне застосування міграцій бази даних при старті, графічна адміністративна панель та інтерактивна документація API. Усе це робить його зручним рішенням для широкого класу проєктів — від внутрішніх корпоративних додатків до мікросервісів стартап-компаній.

Перспективи подальшого розвитку проєкту включають додавання підтримки протоколу OAuth 2.0 для делегованої автентифікації через зовнішніх провайдерів (Google, Microsoft, GitHub), реалізацію Single Sign-On за стандартом SAML 2.0,

інтеграцію з системами SIEM для централізованого моніторингу подій безпеки, додавання підтримки апаратних ключів за стандартом WebAuthn, розробку клієнтських SDK для популярних мов (TypeScript, Python, Go, Java) та інтеграцію з системою резервного копіювання у хмарне об'єктне сховище.

Кваліфікаційна робота продемонструвала практичні навички системного аналізу, проєктування багатошарової архітектури, програмування мовою Go, роботи з реляційними базами даних та сучасними практиками інформаційної безпеки. Розроблений сервіс відповідає поставленим вимогам, успішно проходить функціональне тестування і готовий до використання у реальних проєктах.

СПИСОК ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

1. The Go Programming Language Documentation. Інтернет-доступ: https://go.dev/doc/
2. Chi: lightweight, idiomatic and composable router for building Go HTTP services. Інтернет-доступ: https://github.com/go-chi/chi
3. PostgreSQL 16 Documentation. The PostgreSQL Global Development Group, 2023. Інтернет-доступ: https://www.postgresql.org/docs/16/index.html
4. pgx: PostgreSQL driver and toolkit for Go. Інтернет-доступ: https://github.com/jackc/pgx
5. golang-migrate: Database migrations written in Go. Інтернет-доступ: https://github.com/golang-migrate/migrate
6. golang-jwt/jwt: A Go implementation of JSON Web Tokens. Інтернет-доступ: https://github.com/golang-jwt/jwt
7. M'Raihi D., Machani S., Pei M., Rydell J. RFC 6238: TOTP — Time-Based One-Time Password Algorithm. Інтернет-доступ: https://datatracker.ietf.org/doc/html/rfc6238
8. Jones M., Bradley J., Sakimura N. RFC 7519: JSON Web Token (JWT). Інтернет-доступ: https://datatracker.ietf.org/doc/html/rfc7519
9. OWASP Top Ten 2021. The OWASP Foundation, 2021. Інтернет-доступ: https://owasp.org/Top10/
10. Sanderson M. Practical Clean Architecture in Go. Sebastopol : O'Reilly Media, 2023. — 312 p.
11. Sandhu R., Coyne E., Feinstein H., Youman C. Role-Based Access Control: Standards and Implementations. ACM Computing Surveys, 2022. Vol. 54, № 8. P. 1–34.
12. Docker Compose specification. Docker Inc., 2024. Інтернет-доступ: https://docs.docker.com/compose/compose-file/

13. swaggo/swag: Automatically generate RESTful API documentation with Swagger 2.0 for Go. Інтернет-доступ: https://github.com/swaggo/swag

14. Ольховська О. В. Методичні рекомендації щодо виконання кваліфікаційної роботи здобувачами освітнього ступеня бакалавр спеціальності 122 «Комп'ютерні науки» / Олена Володимирівна Ольховська. – Полтава : ПУЕТ, 2024. – 67 с.

ДОДАТОК А

Файл cmd/api/main.go — точка входу до застосунку.

```
// @title           User Service API
// @version         2.0
// @description     Production-ready HTTP сервіс управління
користувачами (IAM) на Go

// @host      localhost:8082
// @BasePath  /api/v1

// @securityDefinitions.apikey BearerAuth
// @in header
// @name Authorization
// @description Формат: Bearer {access_token}
package main

import (
"context"
"fmt"
"log/slog"
"net/http"
"os"
"os/signal"
"syscall"
"time"

"github.com/golang-migrate/migrate/v4"
_ "github.com/golang-migrate/migrate/v4/database/postgres"
_ "github.com/golang-migrate/migrate/v4/source/file"
"github.com/jackc/pgx/v5/pgxpool"
"golang.org/x/crypto/bcrypt"

"user-service/internal/config"
```

```
"user-service/internal/email"
"user-service/internal/handler"
"user-service/internal/middleware"
pgrepo "user-service/internal/repository/postgres"
"user-service/internal/service"
)

func main() {
logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))

cfg, err := config.Load()
if err != nil {
logger.Error("load config", "error", err)
os.Exit(1)
}

pool, err := pgxpool.New(context.Background(), cfg.Database.DSN())
if err != nil {
logger.Error("connect to database", "error", err)
os.Exit(1)
}
defer pool.Close()

if err := pool.Ping(context.Background()); err != nil {
logger.Error("ping database", "error", err)
os.Exit(1)
}
logger.Info("connected to database")

migrateURL := fmt.Sprintf(
"postgres://%s:%s@%s:%s/%s?sslmode=%s",
cfg.Database.User, cfg.Database.Password,
cfg.Database.Host, cfg.Database.Port,
cfg.Database.Name, cfg.Database.SSLMode,
)
m, err := migrate.New("file://migrations", migrateURL)
```

```
if err != nil {
logger.Error("create migrator", "error", err)
os.Exit(1)
}
if err := m.Up(); err != nil && err != migrate.ErrNoChange {
logger.Error("run migrations", "error", err)
os.Exit(1)
}
logger.Info("migrations applied")

// seed admin if env vars are set
adminEmail := os.Getenv("ADMIN_EMAIL")
adminPassword := os.Getenv("ADMIN_PASSWORD")
if adminEmail != "" && adminPassword != "" {
if err := seedAdmin(context.Background(), pool, logger, adminEmail,
adminPassword); err != nil {
logger.Warn("seed admin failed", "error", err)
}
}

// репозиторії
userRepo    := pgrepo.NewUserRepository(pool)
rtRepo      := pgrepo.NewRefreshTokenRepository(pool)
tokenRepo   := pgrepo.NewTokenRepository(pool)
historyRepo := pgrepo.NewLoginHistoryRepository(pool)
auditRepo   := pgrepo.NewAuditLogRepository(pool)
permRepo    := pgrepo.NewPermissionRepository(pool)
groupRepo   := pgrepo.NewGroupRepository(pool)
totpRepo    := pgrepo.NewTOTPRepository(pool)
apiKeyRepo  := pgrepo.NewAPIKeyRepository(pool)

// сервіси
emailSvc := email.NewService(logger)
userSvc  := service.NewUserService(
userRepo, rtRepo, tokenRepo,
historyRepo, auditRepo, permRepo,
```

```
groupRepo, totpRepo, apiKeyRepo,
emailSvc, cfg.JWT,
)

h := handler.New(userSvc)
loggedRouter := middleware.Logger(logger)(h.Router(cfg.JWT.Secret))

srv := &http.Server{
Addr:         ":" + cfg.Server.Port,
Handler:      loggedRouter,
ReadTimeout:  15 * time.Second,
WriteTimeout: 15 * time.Second,
IdleTimeout:  60 * time.Second,
}

go func() {
logger.Info("server started", "port", cfg.Server.Port)
if err := srv.ListenAndServe(); err != nil && err !=
http.ErrServerClosed {
logger.Error("server error", "error", err)
os.Exit(1)
}
}()

quit := make(chan os.Signal, 1)
signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
<-quit

logger.Info("shutting down...")
ctx, cancel := context.WithTimeout(context.Background(),
10*time.Second)
// ... (продовження файлу не наводиться)
```

Файл internal/handler/auth.go — HTTP-обробники маршрутів автентифікації (фрагмент).

```
package handler

import (
"encoding/json"
"errors"
"net/http"

"user-service/internal/domain"
"user-service/internal/middleware"
)

// Register godoc
// @Summary      Реєстрація користувача
// @Description  Створює акаунт. Надсилає email верифікацію
// @Tags         auth
// @Accept       json
// @Produce      json
// @Param        body body domain.RegisterRequest true "Дані
реєстрації"
// @Success      201 {object} domain.UserResponse
// @Failure      400 {object} ErrorResponse
// @Failure      409 {object} ErrorResponse
// @Router       /auth/register [post]
func (h *Handler) Register(w http.ResponseWriter, r *http.Request) {
var req domain.RegisterRequest
if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
writeError(w, http.StatusBadRequest, "invalid request body")
return
}
if err := req.Validate(); err != nil {
writeError(w, http.StatusBadRequest, err.Error())
return
}
resp, err := h.svc.Register(r.Context(), req)
if err != nil {
if errors.Is(err, domain.ErrEmailAlreadyTaken) {
```

```
writeError(w, http.StatusConflict, err.Error())
return
}
writeError(w, http.StatusInternalServerError, "internal server error")
return
}
writeJSON(w, http.StatusCreated, resp)
}

// Login godoc
// @Summary      Вхід користувача
// @Description  Повертає access_token та refresh_token. Блокується
після 5 невдалих спроб
// @Tags         auth
// @Accept       json
// @Produce      json
// @Param        body body domain.LoginRequest true "Credentials"
// @Success      200 {object} domain.LoginResponse
// @Failure      400 {object} ErrorResponse
// @Failure      401 {object} ErrorResponse
// @Failure      403 {object} ErrorResponse
// @Router       /auth/login [post]
func (h *Handler) Login(w http.ResponseWriter, r *http.Request) {
var req domain.LoginRequest
if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
writeError(w, http.StatusBadRequest, "invalid request body")
return
}
if err := req.Validate(); err != nil {
writeError(w, http.StatusBadRequest, err.Error())
return
}
ip := middleware.GetClientIP(r)
ua := r.Header.Get("User-Agent")
resp, err := h.svc.Login(r.Context(), req, ip, ua)
if err != nil {
```

```
switch {
case errors.Is(err, domain.ErrInvalidCredentials):
writeError(w, http.StatusUnauthorized, err.Error())
case errors.Is(err, domain.ErrAccountBlocked):
writeError(w, http.StatusForbidden, err.Error())
case errors.Is(err, domain.ErrAccountLocked):
writeError(w, http.StatusTooManyRequests, err.Error())
default:
writeError(w, http.StatusInternalServerError, "internal server error")
}
return
}
writeJSON(w, http.StatusOK, resp)
}

// Refresh godoc
// @Summary      Оновлення access token
// @Description  Приймає refresh token, повертає нову пару (ротація)
// @Tags         auth
// @Accept       json
// @Produce      json
// @Param        body body domain.RefreshRequest true "Refresh token"
// @Success      200 {object} domain.LoginResponse
// @Failure      401 {object} ErrorResponse
// @Router       /auth/refresh [post]
func (h *Handler) Refresh(w http.ResponseWriter, r *http.Request) {
var req domain.RefreshRequest
if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
writeError(w, http.StatusBadRequest, "invalid request body")
return
}
if req.RefreshToken == "" {
writeError(w, http.StatusBadRequest, "refresh_token is required")
return
}
resp, err := h.svc.Refresh(r.Context(), req.RefreshToken)
```

```
if err != nil {
writeError(w, http.StatusUnauthorized, err.Error())
return
}
writeJSON(w, http.StatusOK, resp)
}

// Logout godoc
// @Summary      Вихід з системи
// @Tags         auth
// @Accept       json
// @Security     BearerAuth
// @Param        body body domain.RefreshRequest true "Refresh token"
// @Success      204
// @Router       /auth/logout [post]
func (h *Handler) Logout(w http.ResponseWriter, r *http.Request) {
var req domain.RefreshRequest
if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
writeError(w, http.StatusBadRequest, "invalid request body")
return
}
_ = h.svc.Logout(r.Context(), req.RefreshToken)
w.WriteHeader(http.StatusNoContent)
}

// ForgotPassword godoc
// @Summary      Запит скидання пароля
// @Description  Надсилає лист з посиланням. Завжди повертає 202 (не
розкриває існування email)
// @Tags         auth
// @Accept       json
// @Produce      json
// @Param        body body domain.ForgotPasswordRequest true "Email"
// @Success      202 {object} map[string]string
// @Router       /auth/forgot-password [post]
```

```
func (h *Handler) ForgotPassword(w http.ResponseWriter, r
*http.Request) {
var req domain.ForgotPasswordRequest
if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
writeError(w, http.StatusBadRequest, "invalid request body")
return
}
_ = h.svc.ForgotPassword(r.Context(), req.Email)
writeJSON(w, http.StatusAccepted, map[string]string{
"message": "if account exists, reset email has been sent",
})
}

// ResetPassword godoc
// @Summary      Скидання пароля
// @Description  Встановлює новий пароль за токеном з листа
// ... (продовження файлу не наводиться)
```

Файл internal/middleware/auth.go — JWT-middleware: перевірка токена, ролі та дозволів.

```
package middleware

import (
"context"
"net/http"
"strings"

"github.com/golang-jwt/jwt/v5"

"user-service/internal/domain"
)

type contextKey string

const (
```

```
UserIDKey contextKey = "user_id"
RoleKey   contextKey = "role"
)

// UserChecker — інтерфейс для перевірки статусу та дозволів
користувача
type UserChecker interface {
GetUserStatus(ctx context.Context, userID string) (domain.UserStatus,
[]string, error)
}

func Auth(jwtSecret string, checker UserChecker) func(http.Handler)
http.Handler {
return func(next http.Handler) http.Handler {
return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
authHeader := r.Header.Get("Authorization")
if authHeader == "" || !strings.HasPrefix(authHeader, "Bearer ") {
http.Error(w, `{"error":"unauthorized"}`, http.StatusUnauthorized)
return
}

tokenStr := strings.TrimPrefix(authHeader, "Bearer ")
token, err := jwt.Parse(tokenStr, func(t *jwt.Token) (interface{},
error) {
if _, ok := t.Method.(*jwt.SigningMethodHMAC); !ok {
return nil, jwt.ErrSignatureInvalid
}
return []byte(jwtSecret), nil
})
if err != nil || !token.Valid {
http.Error(w, `{"error":"unauthorized"}`, http.StatusUnauthorized)
return
}

claims, ok := token.Claims.(jwt.MapClaims)
if !ok {
```

```
http.Error(w, `{"error":"unauthorized"}`, http.StatusUnauthorized)
return
}

userID, _ := claims["user_id"].(string)
role, _ := claims["role"].(string)

ctx := context.WithValue(r.Context(), UserIDKey, userID)
ctx = context.WithValue(ctx, RoleKey, domain.Role(role))

// перевірка статусу + завантаження дозволів
if checker != nil && userID != "" {
status, perms, err := checker.GetUserStatus(ctx, userID)
if err == nil {
if status == domain.StatusBlocked || status == domain.StatusDeleted {
http.Error(w, `{"error":"account is blocked or deleted"}`,
http.StatusForbidden)
return
}
ctx = SetPermissions(ctx, perms)
}
}

next.ServeHTTP(w, r.WithContext(ctx))
})
}
}

func RequireAdmin(next http.Handler) http.Handler {
return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
role, _ := r.Context().Value(RoleKey).(domain.Role)
if role != domain.RoleAdmin {
http.Error(w, `{"error":"forbidden"}`, http.StatusForbidden)
return
}
next.ServeHTTP(w, r)
```

```
})
}

func GetUserID(ctx context.Context) string {
id, _ := ctx.Value(UserIDKey).(string)
return id
}

func GetRole(ctx context.Context) domain.Role {
role, _ := ctx.Value(RoleKey).(domain.Role)
return role
}

func GetClientIP(r *http.Request) string {
if ip := r.Header.Get("X-Real-IP"); ip != "" {
return ip
}
if ip := r.Header.Get("X-Forwarded-For"); ip != "" {
return strings.Split(ip, ",")[0]
}
if idx := strings.LastIndex(r.RemoteAddr, ":"); idx != -1 {
return r.RemoteAddr[:idx]
}
return r.RemoteAddr
}

const APIKeyUserIDKey contextKey = "api_key_user_id"

// APIKeyAuth middleware authenticates requests using the X-API-Key
header.
// The lookup function receives the raw key and returns the owning
userID and its permissions.
func APIKeyAuth(lookup func(ctx context.Context, key string) (string,
[]string, error)) func(http.Handler) http.Handler {
return func(next http.Handler) http.Handler {
return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
```

```
// ... (продовження файлу не наводиться)
```

Файл internal/middleware/ratelimit.go — обмеження частоти запитів за алгоритмом Token Bucket.

```
package middleware

import (
"net/http"
"strings"
"sync"
"time"
)

type rateLimiter struct {
mu        sync.Mutex
visitors map[string]*visitor
rate      int
burst     int
}

type visitor struct {
tokens   float64
lastSeen time.Time
}

func newRateLimiter(rate, burst int) *rateLimiter {
rl := &rateLimiter{
visitors: make(map[string]*visitor),
rate:      rate,
burst:     burst,
}
go rl.cleanup()
return rl
}
```

```
func (rl *rateLimiter) allow(ip string) bool {
rl.mu.Lock()
defer rl.mu.Unlock()

v, ok := rl.visitors[ip]
if !ok {
rl.visitors[ip] = &visitor{tokens: float64(rl.burst - 1), lastSeen:
time.Now()}
return true
}

now := time.Now()
elapsed := now.Sub(v.lastSeen).Seconds()
v.tokens += elapsed * float64(rl.rate)
if v.tokens > float64(rl.burst) {
v.tokens = float64(rl.burst)
}
v.lastSeen = now

if v.tokens < 1 {
return false
}
v.tokens--
return true
}

func (rl *rateLimiter) cleanup() {
for {
time.Sleep(5 * time.Minute)
rl.mu.Lock()
for ip, v := range rl.visitors {
if time.Since(v.lastSeen) > 10*time.Minute {
delete(rl.visitors, ip)
}
}
rl.mu.Unlock()
```

```
}
}

// RateLimit — загальний rate limiter (60 req/min per IP)
var globalLimiter = newRateLimiter(1, 60)

// AuthRateLimit — суворий limiter для auth ендпоінтів (10 req/min per
IP)
var authLimiter = newRateLimiter(10, 10)

func RateLimit(next http.Handler) http.Handler {
return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
ip := getIP(r)
if !globalLimiter.allow(ip) {
http.Error(w, `{"error":"too many requests"}`,
http.StatusTooManyRequests)
return
}
next.ServeHTTP(w, r)
})
}

func AuthRateLimit(next http.Handler) http.Handler {
return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
ip := getIP(r)
if !authLimiter.allow(ip) {
http.Error(w, `{"error":"too many requests, please try again later"}`,
http.StatusTooManyRequests)
return
}
next.ServeHTTP(w, r)
})
}

func getIP(r *http.Request) string {
if ip := r.Header.Get("X-Real-IP"); ip != "" {
```

```
return ip
}
if ip := r.Header.Get("X-Forwarded-For"); ip != "" {
return strings.Split(ip, ",")[0]
}
if idx := strings.LastIndex(r.RemoteAddr, ":"); idx != -1 {
return r.RemoteAddr[:idx]
}
return r.RemoteAddr
}
// ... (продовження файлу не наводиться)
```

Файл internal/repository/postgres/user_repo.go — реалізація репозиторію користувачів для PostgreSQL (фрагмент).

```
package postgres

import (
"context"
"encoding/json"
"errors"
"fmt"
"strings"
"time"

"github.com/jackc/pgx/v5"
"github.com/jackc/pgx/v5/pgxpool"

"user-service/internal/domain"
)

type UserRepository struct {
db *pgxpool.Pool
}

func NewUserRepository(db *pgxpool.Pool) *UserRepository {
```

```
return &UserRepository{db: db}
}

const userSelectFields = `id, email, password, name, role, status,
COALESCE(avatar_url,''), metadata, failed_attempts,
locked_until, deleted_at, email_verified, totp_enabled,
required_actions,
created_at, updated_at`

func scanUser(row pgx.Row) (*domain.User, error) {
u := &domain.User{}
var metaRaw []byte
var reqActRaw []byte
var avatarURL string
err := row.Scan(
&u.ID, &u.Email, &u.Password, &u.Name,
&u.Role, &u.Status, &avatarURL, &metaRaw,
&u.FailedAttempts, &u.LockedUntil, &u.DeletedAt,
&u.EmailVerified, &u.TOTPEnabled, &reqActRaw,
&u.CreatedAt, &u.UpdatedAt,
)
if err != nil {
return nil, err
}
u.AvatarURL = avatarURL
if metaRaw != nil {
_ = json.Unmarshal(metaRaw, &u.Metadata)
}
if u.Metadata == nil {
u.Metadata = map[string]any{}
}
if reqActRaw != nil {
_ = json.Unmarshal(reqActRaw, &u.RequiredActions)
}
if u.RequiredActions == nil {
u.RequiredActions = []string{}
```

```
}
return u, nil
}

func (r *UserRepository) Create(ctx context.Context, user
*domain.User) error {
meta, _ := json.Marshal(user.Metadata)
if meta == nil {
meta = []byte("{}")
}
var reqActRaw []byte
err := r.db.QueryRow(ctx,
`INSERT INTO users (email, password, name, role, status, metadata)
VALUES ($1, $2, $3, $4, $5, $6)
RETURNING `+userSelectFields,
user.Email, user.Password, user.Name,
user.Role, domain.StatusActive, meta,
).Scan(
&user.ID, &user.Email, &user.Password, &user.Name,
&user.Role, &user.Status, &user.AvatarURL, &meta,
&user.FailedAttempts, &user.LockedUntil, &user.DeletedAt,
&user.EmailVerified, &user.TOTPEnabled, &reqActRaw,
&user.CreatedAt, &user.UpdatedAt,
)
if err != nil {
if isUniqueViolation(err) {
return domain.ErrEmailAlreadyTaken
}
return err
}
_ = json.Unmarshal(meta, &user.Metadata)
if reqActRaw != nil {
_ = json.Unmarshal(reqActRaw, &user.RequiredActions)
}
if user.RequiredActions == nil {
user.RequiredActions = []string{}
```

```
}
return nil
}

func (r *UserRepository) GetByID(ctx context.Context, id string)
(*domain.User, error) {
row := r.db.QueryRow(ctx,
`SELECT `+userSelectFields+` FROM users WHERE id = $1 AND deleted_at
IS NULL`, id)
u, err := scanUser(row)
if err != nil {
if errors.Is(err, pgx.ErrNoRows) {
return nil, domain.ErrUserNotFound
}
return nil, err
}
return u, nil
}

func (r *UserRepository) GetByEmail(ctx context.Context, email string)
(*domain.User, error) {
row := r.db.QueryRow(ctx,
`SELECT `+userSelectFields+` FROM users WHERE email = $1 AND
deleted_at IS NULL`, email)
u, err := scanUser(row)
if err != nil {
if errors.Is(err, pgx.ErrNoRows) {
return nil, domain.ErrUserNotFound
}
return nil, err
}
return u, nil
}

func (r *UserRepository) Update(ctx context.Context, user
*domain.User) error {
```

```
meta, _ := json.Marshal(user.Metadata)
if meta == nil {
meta = []byte("{}")
}
reqActJSON, _ := json.Marshal(user.RequiredActions)
if reqActJSON == nil {
reqActJSON = []byte("[]")
}
// ... (продовження файлу не наводиться)
```

Міграція 000004 — створення таблиць історії входів, токенів скидання пароля, верифікації email і журналу аудиту.

```
CREATE TABLE IF NOT EXISTS login_history (
id          UUID        PRIMARY KEY DEFAULT gen_random_uuid(),
user_id     UUID        NOT NULL REFERENCES users(id) ON DELETE
CASCADE,
ip_address VARCHAR(45) NOT NULL,
user_agent TEXT,
success     BOOLEAN     NOT NULL DEFAULT true,
created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
CREATE INDEX IF NOT EXISTS idx_login_history_user_id ON
login_history(user_id);
CREATE INDEX IF NOT EXISTS idx_login_history_created_at ON
login_history(created_at);

CREATE TABLE IF NOT EXISTS password_reset_tokens (
id          UUID        PRIMARY KEY DEFAULT gen_random_uuid(),
user_id     UUID        NOT NULL REFERENCES users(id) ON DELETE
CASCADE,
token       TEXT        UNIQUE NOT NULL,
expires_at TIMESTAMPTZ NOT NULL,
used        BOOLEAN     NOT NULL DEFAULT false,
created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

```
CREATE INDEX IF NOT EXISTS idx_prt_token ON
password_reset_tokens(token);

CREATE TABLE IF NOT EXISTS email_verification_tokens (
id          UUID        PRIMARY KEY DEFAULT gen_random_uuid(),
user_id     UUID        NOT NULL REFERENCES users(id) ON DELETE
CASCADE,
token       TEXT        UNIQUE NOT NULL,
expires_at TIMESTAMPTZ NOT NULL,
created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
CREATE INDEX IF NOT EXISTS idx_evt_token ON
email_verification_tokens(token);

CREATE TABLE IF NOT EXISTS audit_logs (
id          UUID        PRIMARY KEY DEFAULT gen_random_uuid(),
actor_id    UUID        REFERENCES users(id) ON DELETE SET NULL,
action      VARCHAR(100) NOT NULL,
target_id   UUID,
target_type VARCHAR(50),
details     JSONB       NOT NULL DEFAULT '{}',
ip_address  VARCHAR(45),
created_at  TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
CREATE INDEX IF NOT EXISTS idx_audit_logs_actor_id  ON
audit_logs(actor_id);
CREATE INDEX IF NOT EXISTS idx_audit_logs_created_at ON
audit_logs(created_at);
```