Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Іван ПАРХОМЕНКО
«__» _____ 2025 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань *12 Інформаційні технології*
(шифр і назва галузі знань)

спеціальність *125 Кібербезпека та захист інформації*
(код і назва спеціальності)

освітній ступень *магістр*

освітньо-наукова програма *Кібербезпека*
(назва освітньої програми)

на тему: «Метод впровадження багатофакторної аутентифікації в Application Programming Interface»

Виконавець: студент ІІ курсу, групи КБм-21

_____ **Андрій СТАРОСЕЛЬСЬКИЙ**
(підпис) (Ім'я, ПРІЗВИЩЕ)

	Ім'я, ПРІЗВИЩЕ	**Підпис**
Науковий керівник	Володимир НАКОНЕЧНИЙ	
Нормоконтроль	Іван БІЛОКОНЬ	

Київ 2025

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Іван ПАРХОМЕНКО
«25» жовтня 2024 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальність *125 Кібербезпека та захист інформації*
(код і назва спеціальності)

освітній ступень *магістр*

Здобувача(ки) КБм-21 (група) Старосельського Андрія Олексійовича (прізвище ім'я по-батькові)

Тема кваліфікаційної роботи Метод впровадження багатофакторної аутентифікації в Application Programming Interface

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Рішення засідання кафедри кібербезпеки та захисту інформації факультету інформаційних технологій протокол № 4 від 24.10.2024 р.

2. МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень Процес аутентифікації користувачів у API.

Предмет досліджень Механізми впровадження багатофакторної аутентифікації у API.

Мета Розробка методу впровадження багатофакторної аутентифікації у API та його практична реалізація.

Вихідні дані для проведення роботи Архітектури API, механізми аутентифікації у веб-застосунках, підходи до впровадження багатофакторної аутентифікації

3. ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна удосконалення методу аутентифікації у API за рахунок впровадження можливості використання багатофакторної аутентифікації.

Практична цінність програмна реалізація методу впровадження багатофакторної аутентифікації у API.

4. ЕТАПИ ВИКОНАННЯ РОБОТИ

Найменування етапів робіт	Строки виконання робіт (початок-кінець)
Уточнення постановки задачі	25.10.2024 – 20.01.2025
Аналіз літературних джерел	21.01.2025 – 03.02.2025
Огляд сучасних методів аутентифікації	04.02.2025 – 17.02.2025
Аналіз технологій, що використовуються у аутентифікації	18.02.2025 – 03.03.2025
Дослідження популярних архітектур API	04.03.2025 – 17.03.2025
Огляд мов програмування, що використовуються у розробці API	18.03.2025 – 31.03.2025
Дослідження підходів до впровадження багатофакторної аутентифікації у API	01.04.2025 – 14.04.2025
Розробка методу впровадження багатофакторної аутентифікації у API	15.04.2025 – 28.04.2025
Реалізація API із багатофакторною аутентифікацією на основі запропонованого методу	29.04.2025 – 07.05.2025
Оформлення пояснювальної записки згідно методичних рекомендацій	08.05.2025 – 15.05.2025
Подача пакету документів на розгляд ЕК	15.05.2025 – 19.05.2025

Завдання видав _____ Володимир НАКОНЕЧНИЙ
(підпис) (Ім'я, ПРІЗВИЩЕ)

Завдання прийняв до виконання _____ Андрій СТАРОСЕЛЬСЬКИЙ
(підпис) (Ім'я, ПРІЗВИЩЕ)

Дата видачі завдання: 25.10.2024 р.

Термін подання кваліфікаційної роботи до ЕК 19.05.2025 р

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Метод впровадження багатофакторної аутентифікації в Application Programming Interface»: 105 сторінок, 29 рисунків та 1 таблиця. 88 літературних джерел.

Актуальність теми: значна кількість успішних кібератак, пов'язаних із компрометацією облікових даних користувачів веб-ресурсів робить традиційні парольні системи недостатньо надійними для сучасних інформаційних систем. Розробка Application Programming Interface (API) з багатофакторною аутентифікацією дозволяє запропонувати рішення цієї проблеми, оскільки додаткові фактори автентифікації суттєво знижують ризик несанкціонованого доступу, особливо враховуючи поширену практику повторного використання скомпрометованих паролів на різних ресурсах.

У роботі проведено огляд і аналіз сучасних методів багатофакторної аутентифікації. Розглянуто популярні архітектури API. Досліджено поширені інструменти розробки API. Проаналізовано підходи до впровадження багатофакторної аутентифікації у API.

Розроблено метод для впровадження багатофакторної аутентифікації в API. Розглянуті пропозиції щодо практичної реалізації API з багатофакторною аутентифікацією на основі розробленого методу.

Мета роботи – розробка методу впровадження багатофакторної аутентифікації у API та його практична реалізація.

Об’єкт дослідження – процес аутентифікації користувачів у API.

Методи дослідження – методи наукової абстракції, індукції та дедукції (для аналізу та систематизації існуючих підходів до багатофакторної аутентифікації та визначення ключових концепцій); історико-логічний метод (для класифікації та групування існуючих технологій аутентифікації); метод порівняльного аналізу (для оцінки різних підходів до аутентифікації).

Наукова новизна: удосконалення методу аутентифікації у API за рахунок впровадження можливості використання багатофакторної аутентифікації.

Ключові слова: багатофакторна аутентифікація, веб-застосунок, аутентифікація, облікові дані, кібербезпека, API.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

БФА – Багатофакторна аутентифікація
API – Application Programming Interface
TLS – Transport Layer Security
SSL – Secure Sockets Layer
SSO – Single Sign-on
OTP – One-Time Password
HOTP – HMAC-based One-Time Password
TOTP – Time-based One-Time Password

ЗМІСТ

ВСТУП 8
РОЗДІЛ 1 МЕТОДИ БАГАТОФАКТОРНОЇ АУТЕНТИФІКАЦІЇ 12
1.1 Концепція багатофакторної аутентифікації 12
1.2 Класифікація методів аутентифікації 19
1.3 Сучасні технології та підходи у аутентифікації 25
Висновки за розділом 1 31
РОЗДІЛ 2 АРХІТЕКТУРИ ТА ІНСТРУМЕНТИ ДЛЯ РОЗРОБКИ API З БАГАТОФАКТОРНОЮ АУТЕНТИФІКАЦІЄЮ 34
2.1 Огляд архітектур API 34
2.2 Мови програмування та фреймворки для розробки API 45
2.3 Підходи для впровадження багатофакторної аутентифікації 53
Висновки за розділом 2 57
РОЗДІЛ 3 РОЗРОБКА МЕТОДУ ВПРОВАДЖЕННЯ БАГАТОФАКТОРНОЇ АУТЕНТИФІКАЦІЇ У API ТА ЙОГО ПРОГРАМНА РЕАЛІЗАЦІЯ 60
3.1 Метод впровадження багатофакторної аутентифікації та вимоги до програмного рішення 60
3.2 Інструменти для розробки програмного рішення 71
3.3 Огляд коду програмного рішення 76
3.4 Демонстрація роботи створеного рішення 83
Висновки за розділом 3 89
ВИСНОВКИ 92
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 96
ДОДАТОК А 106
ДОДАТОК Б 123

ВСТУП

У сучасному цифровому світі, де транзакції, комунікації і щоденна взаємодія все частіше відбуваються онлайн, надійні заходи безпеки стали незамінними частинами будь-якого процесу. Традиційні методи аутентифікації на основі паролів більше не є достатніми для захисту конфіденційної інформації від складних кіберзагроз [1]. Випадки витоку даних та несанкціонованого доступу підкреслюють критичну необхідність у більш просунутих рішеннях безпеки, таких як багатофакторна аутентифікація (БФА). БФА підсилює безпеку, вимагаючи від користувачів пройти додаткові етапи перевірки перед отриманням доступу до систем [2]. Цей підхід значно знижує ймовірність несанкціонованого доступу, навіть якщо один з факторів, таких як пароль, скомпрометовано. За даними компанії Google кількість зламів облікових записів зменшується на 50%, якщо використовується багатофакторна аутентифікація [3].

Крім того, статистика останніх років демонструє велику кількість успішних кібератак, пов'язаних з компрометацією облікових даних користувачів. За даними звіту Verizon Data Breach Investigations Report 2024, приблизно 24% успішних зламів корпоративних систем пов'язані із компрометацією облікових даних користувачів [4].

Компанія SpyCloud, що займається розробкою рішень із кібербезпеки, повідомляє про тривожну тенденцію, яка полягає у постійному повторному використання скомпрометованих паролів користувачами в різних додатках, що дає злочинцям більше можливостей для заволодіння обліковими записами. Згідно з нещодавнім звітом Identity Exposure Report, понад 74% паролів, які містилися у базі даних для досліджень, були використані повторно, не зважаючи на те, що вони вже було зафіксовані у щонайменше двох великих витоках.

Цей факт підкреслює необхідність впровадження додаткових рівнів захисту, одним з яких є багатофакторна аутентифікація, оскільки компанії, які заохочують працівників до поліпшення «гігієни» паролів, а також надають пріоритет захисту

облікових даних за допомогою таких заходів, як багатофакторна аутентифікація, знаходяться в набагато кращому становищі для захисту від кіберзагроз [5].

Значення БФА полягає в здатності вирішувати нинішній парадокс безпеки: необхідність впровадження суворих контрольних заходів безпеки без створення незручностей для користувачів або надмірного навантаження на ресурси організації. Цей виклик може бути достатньо суттєвим у організаціях, де постійно ведеться боротьба за підвищення безпеки в умовах обмежених бюджетів і існуючої технологічної інфраструктури. Тому вивчення та розробка ефективних методів БФА для Application Programming Interface (API) є не лише необхідним, але й актуальним.

Вирішуючи ці виклики, можна досягти більш безпечного середовища, що відповідає як технічним вимогам, так і очікуванням користувачів. Дане дослідження спрямоване на вивчення та покращення методів впровадження БФА у розробці API, маючи на меті досягти оптимального балансу між безпекою та зручністю використання.

Станом на сьогодні, у науковій літературі та дослідженнях значна увага приділяється питанням кібербезпеки, зокрема і питанню аутентифікації. На думку автора цієї роботи, незважаючи на значний обсяг досліджень у сфері багатофакторної аутентифікації, існує ряд невирішених проблем, особливо в контексті сучасних API-систем:

- відсутність єдиної методології впровадження БФА в API з урахуванням специфіки різних архітектурних підходів;
- складність балансування між безпекою та зручністю використання зі сторони користувачів;
- складності інтеграції БФА в існуючі системи без порушення їх функціональності.

Цей факт підкреслює важливість подальших досліджень, зокрема в розробці та впровадженні нових, більш ефективних методів для забезпечення безпеки API.

Таким чином, актуальність цього дослідження обумовлена значною кількістю успішних кібератак, пов'язаних із компрометацією облікових даних користувачів веб-ресурсів робить традиційні парольні системи недостатньо надійними для

сучасних інформаційних систем. Розробка API з багатофакторною аутентифікацією дозволяє запропонувати рішення цієї проблеми, оскільки додаткові фактори автентифікації суттєво знижують ризик несанкціонованого доступу, особливо враховуючи поширену практику повторного використання скомпрометованих паролів на різних ресурсах.

Метою роботи є розробка методу впровадження багатофакторної аутентифікації у API та його практична реалізація.

Для досягнення цієї мети визначено такі завдання:

- Провести огляд і аналіз сучасних методів багатофакторної аутентифікаці.
- Розглянути популярні архітектури API.
- Дослідити поширені інструменти розробки API.
- Провести огляд підходів до впровадження багатофакторної аутентифікації у API.
- Розробити метод для впровадження багатофакторної аутентифікації в API.
- Реалізувати API з багатофакторною аутентифікацією.

Об’єктом дослідження є процес аутентифікації користувачів у API.

Предметом дослідження є механізми впровадження багатофакторної аутентифікації у API.

Методами дослідження є методи наукової абстракції, індукції та дедукції (для аналізу та систематизації існуючих підходів до багатофакторної аутентифікації та визначення ключових концепцій); історико-логічний метод (для класифікації та групування існуючих технологій аутентифікації); метод порівняльного аналізу (для оцінки різних підходів до аутентифікації).

Наукова новизна цього дослідження полягає в удосконаленні методу аутентифікації у API за рахунок впровадження можливості використання багатофакторної аутентифікації.

Практичною цінністю є програмна реалізація методу впровадження багатофакторної аутентифікації у API.

Результати можуть бути застосовані розробниками програмного забезпечення та системними архітекторами для впровадження ефективних механізмів багатофакторної аутентифікації.

РОЗДІЛ 1
МЕТОДИ БАГАТОФАКТОРНОЇ АУТЕНТИФІКАЦІЇ

1.1 Концепція багатофакторної аутентифікації

В умовах сучасного інформаційного суспільства, де майже кожен аспект нашого життя пов'язаний із цифровими технологіями, проблема захисту особистої інформації стає дедалі актуальнішою. Щоденно люди залучені до численних взаємодій із різноманітними інформаційними системами, починаючи від банківських транзакцій та онлайн-шопінгу до соціальних мереж і корпоративних мереж підприємств. У такому контексті надійна аутентифікація користувачів є критично важливою для забезпечення інформаційної безпеки.

Потреба у більш захищених механізмах аутентифікації виникла на тлі численних випадків зламів систем і розкриття персональних даних, які стали можливими через вразливість традиційних підходів, найбільш поширеним з яких є звичайна однофакторна аутентифікація за допомогою пароля. Такі підходи вразливі до атак, пов'язаних із соціальною інженерією, підбором паролів, фішингом та іншими методами компрометації.

БФА постає як більш надійний спосіб захисту доступу до чутливої інформації. Вона передбачає використання кількох незалежних факторів для підтвердження особи користувача, значно зменшуючи ризик несанкціонованого доступу. Цей метод базується на простому і водночас потужному принципі: якщо один фактор може бути скомпрометований, то всі фактори одночасно – набагато складніше.

Багатофакторна аутентифікація – це багаторівневий метод захисту віртуального та фізичного доступу, коли система вимагає від користувача комбінації двох або більше аутентифікаторів для підтвердження особи користувача для входу в систему [6]. На відміну від однофакторної аутентифікації, яка зазвичай використовує лише пароль або інший секретний ключ, БФА підвищує рівень безпеки, комбінуючи кілька незалежних факторів (рис. 1.1).

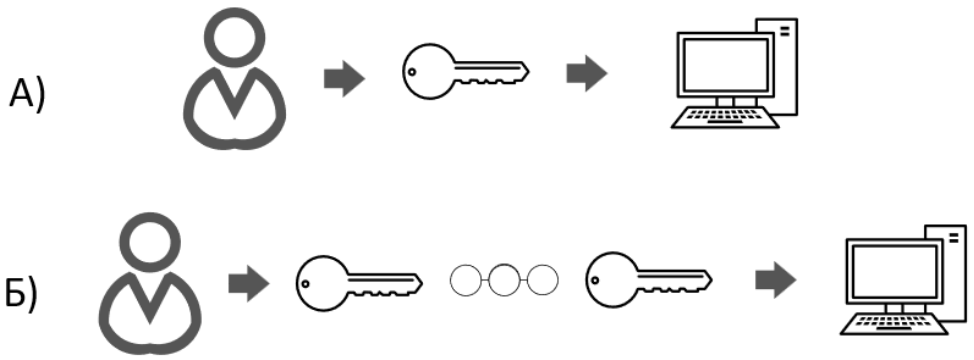

Рисунок 1.1 – Узагальнена схема однофакторної (А) та багатофакторної (Б) аутентифікації

Основні елементи, які можуть бути використані в контексті БФА та аутентифікації в цілому, зазвичай класифікуються трьома категоріями [7]:

- Те, що людина знає: це найпоширеніший тип фактора і включає паролі, ПІН-коди або відповіді на секретні запитання.
- Те, що людина має: цей фактор вимагає фізичного об'єкта, наприклад, смарт-карту, токен, мобільний телефон або інший пристрій, який може генерувати чи отримувати одноразові паролі (ОТП).
- Те, чим людина є**:** це біометричні дані, такі як відбитки пальців, розпізнавання обличчя, сканування сітківки ока або голосова ідентифікація.

Окрім цього може також використовуватися фактор місця та часу.

Багатофакторна аутентифікація ґрунтується на принципі, що об'єднання декількох незалежних компонентів значно знижує ймовірність компрометації захищеної системи. Злом однієї з частин аутентифікації не призведе до успіху злочинця, якщо інші фактори залишаються недоторканими та надійно захищеними.

Важливим аспектом у впровадженні БФА є баланс між безпекою та зручністю для користувача. Хоча додаткові фактори підвищують безпеку, вони також можуть ускладнювати доступ для користувачів. Тому ідеальне рішення повинно враховувати специфічні потреби системи і користувачів, а також можливі ризики.

БФА є важливим кроком уперед у забезпеченні безпеки ідентифікаційних процесів у сучасному цифровому світі. Це рішення базується на принципі комплексності, згідно з яким залучення кількох незалежних факторів надійності дозволяє значно знизити ймовірність несанкціонованого доступу, зокрема додавання додаткових факторів дозволяє заблокувати майже 100% автоматичних атак на облікові записи [8]. Усі фактори, що використовуються, мають працювати в тандемі, створюючи ефективний бар'єр проти атак на аутентифікацію. Використання декількох джерел перевірки вимагає не тільки знання, але й володіння певними засобами або підтвердження унікальних фізичних чи поведінкових рис користувача. Це забезпечує інтеграцію різноманітних технологій, таких як паролі, смарт-карти, токени та біометричні дані, які взаємодіють та підтримують одне одного.

Принципи безпеки БФА передбачають зменшення залежності від паролів як єдиного засобу, а натомість – перенаправлення акценту на багатошаровий підхід. Цей підхід є ефективним захистом від багатьох видів атак соціальної інженерії, який є все більш поширеним способом отримання несанкціонованого доступу [9]. Навіть якщо зловмисник отримає доступ до одного з факторів, решта факторів створює додаткові бар'єри, що ускладнює проникнення в систему. Крім того, завдяки диверсифікації факторів безпеки, системи можуть адаптуватися до різних сценаріїв і загроз, реагуючи на них своєчасно і максимально ефективно.

З іншого боку, важливо зазначити, що підвищена безпека, яку забезпечує багатофакторна аутентифікація, не повинна негативно впливати на зручність користувача. Сучасні системи БФА розроблені з урахуванням потреб кінцевих користувачів, що є критичним для їх прийняття і використання. Зручність використання залишається ключовим фактором для успіху впровадженої системи, оскільки складність або незручність можуть призвести до відмови користувачів від використання рішення.

Сучасні технології, такі як розпізнавання обличчя, відбитків пальців або використання мобільних додатків-аутентифікаторів, сприяють тому, що багатофакторна аутентифікація стає швидшою і менш нав'язливою для користувачів. Розробники можуть створити досвід аутентифікації, який органічно вписується у

повсякденне життя користувачів. Приділяючи особливу увагу інтерактивному дизайну і процесам. Особливо актуальним стає баланс між надійністю і зручністю в корпоративних середовищах, де велика кількість співробітників користується системами щоденно і труднощів з доступом можуть стати проблемою для функціонування системи.

Таким чином, багатофакторна аутентифікація є не просто технічним рішенням, а важливим аспектом дизайну користувацького досвіду та інжинірингу безпеки. Це стратегічний інструмент, який необхідно адаптувати під специфічні потреби кожної організації та її користувачів, гармонійно поєднуючи вимоги до безпеки та комфортності взаємодії з системами. Впровадження має враховувати поточні та майбутні тенденції технологічного розвитку, зберігаючи при цьому гнучкість у підходах і можливість швидкого реагування на нові загрози у кіберпросторі.

Для розуміння необхідності у поточних рішеннях для підтвердження особистості, необхідно розглянути історію аутентифікації, як такої. Можна вважати, що необхідність верифікації особи відносить, в переважній більшості, до IT-систем, але це не зовсім так. Підтвердження особистості використовувалося людьми задовго до появи будь-яких інформаційних систем.

Найдавнішими відомими механізмами аутентифікації є циліндричні печатки, знайдені в Месопотамії. Ці циліндри, зазвичай виготовлені з каменю або металу, мали унікальні гравірування чи символи. При прокачуванні по глиняних табличках, вони залишали відбитки, які слугували позначками для підтвердження особистості. З переходом древніх комунікацій від глиняних табличок до папірусу і паперу, циліндричні печатки замінили восковими або штемпелями з чорнилом. Часто для відбитка використовували перстні-печатки, які занурювали у віск або чорнило і наносили відбиток на папір [10].

Давню історію мають і звичні нам паролі. Давньогрецький історик Полібій писав, що римські вартові використовували усні паролі для контролю доступу до певних зон, змінюючи їх щодня [11]. Ймовірно, походження паролів є ще давнішим, проте зафіксована інформація щодо цього відсутня.

Використання власноручного підпису стало популярним в середньовічній Європі, поступово перетворюючи перстні-печатки на модний аксесуар. Люди почали проставляти свої імена або ініціали на договорах, листах чи указах як форму аутентифікації та підтвердження. У 1600-х роках підписи стали використовуватися на паперових документах, а у 1677 році англійський парламент ухвалив закон, що вимагав підпису на певних типах договорів для набуття ними юридичної сили [10].

Використання біометричних даних може здаватися відносно новим підходом через його поширеність в повсякденній діяльності, наприклад при розблокуванні електронних пристроїв, однак даний метод має власну історію. Перша система біометричної ідентифікації з’явилася у Франції в 1800-х роках, коли Альфонс Бертільйон розробив метод класифікації та порівняння злочинців за допомогою замірів конкретних частин тіла [12]. У 1880-х роках відбитки пальців почали використовувати як засіб ідентифікації злочинців і як підпис на документах. Існують дискусії щодо того, хто саме започаткував використання відбитків пальців для ідентифікації, але Едварду Генрі приписують розробку стандарту класифікації відбитків пальців, відомого як Система Генрі, яка розподіляла відбитки кінчиків пальців за певними категоріями [13].

У наступному сторіччі біометрія стала активно розвиватися як дослідницька галузь, а наприкінці 1900-х років цифрова епоха відкрила нові можливості для біометрії. За останнє десятиліття біометрична технологія стрімко розвинулась і стала частиною повсякденного життя.

У період, коли почали з’являтися перші комп’ютери. В епоху мейнфреймів, питання безпеки не займало центрального місця. Доступ строго контролювався, а користувачами були, головним чином, довірені особи, які працювали у фізично захищеному середовищі. Аутентифікація була примітивною і часто спиралася на прості імена користувачів і спільні паролі. Ці системи, хоча й були прийнятними для їх обмеженого контексту, не мали необхідної надійності для ширшого, більш відкритого доступу [14].

У цей період комп'ютери були великими, дорогими та повільними. Основними власниками цих машин були університети та великі підприємства. Щоб

задовольнити високий попит на обчислювальні ресурси, деякі провідні університети запровадили операційні системи з розподілом часу, такі як Compatible Time-Sharing System (CTSS). Це дозволило кільком користувачам ділитися ресурсами, які надавав один комп'ютер. Однак конфіденційність файлів у спільній системі стала проблемою. Рішення з'явилося у 1961 році, коли дослідник MIT Фернандо Корбато впровадив просту програму для зберігання паролів. На жаль, вони зберігалися у звичайному текстовому файлі у незашифрованому вигляді, що робило їх вразливими до зламу. Це продемонстрував Алан Шерр, який знайшов повний список користувацьких паролів, у спробах подовжити доступний час для використання машини [14].

На початку 1970-х років стало зрозуміло, що зберігання паролів у незашифрованому вигляді було жахливою ідеєю. Для вирішення проблеми Роберт Моррісон запозичив з криптографії хеш-функцію. Ця функція дозволяла легко обчислювати секретне значення в одному напрямку, але ускладнювала зворотний процес. Хакери згодом знайшли способи обходу хешування, що призвело до впровадження солей, які додавали елементи випадковості до хешу, роблячи його унікальнішим та складнішим для зламу. Крім того, у середині 70-х років почала розвиватися асиметрична криптографія, також відома як інфраструктура відкритих ключів. Однак широке використання вона не отримала до початку 90-х років. Причиною було те, що технологія була строго засекречена і використовувалася тільки в урядових установах. [15].

У 1980-х роках Леслі Лемпорт представив концепцію одноразових паролів, оскільки статичні паролі вже не були надійними. Хакери могли легко викрасти або перехопити паролі, що спонукало індустрію безпеки розробляти більш захищені рішення для аутентифікації. У відповідь на це і виникла концепція OTP (One-Time Password), де користувач мав використовувати інший пароль щоразу, коли хотів отримати доступ до сервісу. У наш час метод доставки таких паролів еволюціонував до використання SMS, електронної пошти або мобільних додатків [16].

У 1990-х роках інфраструктура відкритих ключів вийшла на загал, що було зумовлено потребою в безпечній аутентифікації в мережі Інтернет. Цифрові

сертифікати та пари відкритого/закритого ключів стали необхідними для перевірки ідентичності користувачів, які отримують доступ до конфіденційної інформації онлайн. Це призвело до розробки протоколів TLS (Transport Layer Security) і SSL (Secure Sockets Layer) [17].

У 2000-х роках отримали свій розвиток багатофакторна аутентифікація та рішення для єдиного входу. БФА вимагала використання кількох факторів аутентифікації для перевірки ідентичності користувача, поєднуючи знання (наприклад, PIN-код або пароль) та володіння (наприклад, цифровий сертифікат або мобільний пристрій). SSO (Single Sign-on), що базувалася на авторитетній третій стороні для перевірки ідентичності, у свою чергу усувала необхідність перевірки облікових даних на кожному сайті. Однак це рішення продемонструвало і певні недоліки, оскільки компрометація одного облікового запису могла поставити під загрозу інші [14].

На початку 2010-х років відбувся розвиток біометричної аутентифікації, яка використовувала фізичні особливості, такі як відбитки пальців, розпізнавання обличчя та голосових шаблонів, для перевірки ідентичності. Вперше можливість запровадив аутентифікацію за відбитками пальців з’явилася у смартфоні Pantech GI100, а згодом ініціативу підхопили і інші бренди. Біометрія стала частиною нашого повсякденного життя, від розблокування смартфонів до підтвердження онлайн-покупок [18].

У цей період виник також і інтерес до так званої «поведінкової аутентифікації», метою якої було створення безперешкодного користувацького досвіду. Цей метод має на меті використовувати унікальні шаблони користувача для визначення ідентичності. Прикладами поведінкової аутентифікації є аналіз почерку людини, аналіз шаблону/швидкості набору тексту, тиску пальців, підпису, голосу, жестів, швидкості ходьби та характеристик свайпів [19]. Такий підхід пропонує зручний досвід для користувачів завдяки своїй ненав'язливості.

Таким чином можна підсумувати, історія аутентифікації значно розвинулася протягом останніх шести десятиліть. Від простих паролів до складних методів багатофакторної та біометричної аутентифікації, що підкреслює безперервний

прогрес у технологіях безпеки. Сьогодні багато сервісів пропонують комплексні рішення для аутентифікації, інтегруючи різні інструменти, що дозволяє забезпечити зручність для користувачів та надійність з точки зору безпеки.

1.2 Класифікація методів аутентифікації

Аутентифікація з використанням одного фактора є одним з найпоширеніших способів перевірки особи користувача в інформаційних системах [20]. Вона передбачає використання лише одного рівня перевірки для підтвердження права доступу до певної інформації або сервісу. Цей вид аутентифікації заснований на принципі, що користувач може підтвердити свою особу, надавши певне знання або властивість, таку як пароль або пін-код.

Визначення однофакторної аутентифікації полягає в тому, що це процес перевірки особистості за допомогою одного єдиного методу [21]. Найчастіше цим методом є щось, що вам відомо (наприклад, пароль або пін-код) [20]. Однофакторна аутентифікація є одним з найстаріших та найпростіших способів захисту інформації, який широко використовується в різних сценаріях – від доступу до електронної пошти до використання банківських послуг онлайн. Вона забезпечує базовий рівень безпеки, завдяки чому стала стандартним засобом у багатьох системах.

Найпоширенішим методом однофакторної аутентифікації є використання паролів [20]. Паролі представляють собою секретну комбінацію символів, відомих лише користувачу та системі. Вони можуть бути різної довжини та складності, від простих, як "123456", до складних, що складаються з випадкових комбінацій літер, цифр та символів. Незважаючи на свою поширеність, паролі мають ряд вразливих місць, таких як можливість їх підбору за допомогою атак грубої сили чи викрадення через фішинг.

Ще одним розповсюдженим методом є використання пін-кодів, які найчастіше застосовуються у фінансових транзакціях або для розблокування мобільних пристроїв. Пін-коди можуть бути коротшими, ніж паролі, це зазвичай чотиризначні або шестизначні комбінації чисел. Вони мають свої переваги, наприклад, легкість

запам’ятовування та введення, однак їх менша довжина збільшує шанси на здогадування через метод вгадування або підбору.

Перевагами однофакторної аутентифікації є її простота та зручність. Користувачам не потрібно запам'ятовувати складні комбінації, а впровадження такої системи у додатки або сервіси вимагає менших ресурсів, ніж багатофакторна аутентифікація. Вона є економічно вигідною, адже не потребує додаткового обладнання чи складних алгоритмів обробки. Для багатьох серйозних задач однофакторна аутентифікація може бути достатньою, особливо якщо йдеться про системи з низьким рівнем потенційних ризиків.

Однак, з усіма її перевагами існують значні обмеження, які варто враховувати. Головною проблемою однофакторної аутентифікації є її вразливість до атак грубої сили та компрометації. Паролі можуть бути скомпрометовані, викрадені за допомогою кейлогерів або отримані через соціальну інженерію. Користувачі часто обирають найлегші і найбільш зрозумілі паролі, чим полегшують задачу зловмисникам [22]. Більше того, якщо один і той самий пароль використовується для кількох облікових записів, його компрометація призводить до серйозних наслідків.

В умовах зростаючих ризиків кіберзлочинності, багато організацій розглядають можливість переходу на багатофакторну аутентифікацію, яка пропонує додатковий рівень захисту. Однак однофакторна аутентифікація все ще залишається найпростішим рішенням, особливо коли йдеться про потреби малого бізнесу чи загального використання, де баланс між зручністю та безпекою виходить на перший план.

Багатофакторна аутентифікація являє собою надійніший підхід до захисту інформаційних систем. Даний метод аутентифікації вимагає від користувача підтвердження своєї особи за допомогою двох або більше незалежних факторів. Таке рішення виникло як наслідок необхідності посилення безпеки систем, усуваючи основні недоліки, притаманні однофакторній аутентифікації.

На відміну від однофакторної системи, де наявність одного елемента, такого як пароль чи пін-код, є достатньою для отримання доступу, багатофакторна аутентифікація значно ускладнює завдання можливим зловмисникам. Це досягається

шляхом інтеграції кількох рівнів безпеки, які знижують ймовірність неправомірного доступу навіть у випадку компрометації одного з факторів. БФА не тільки використовує те, наприклад, що користувач знає, як в однофакторній системі, але й доповнює процес додатковими рівнями перевірки.

Інтеграція різних факторів у багатофакторній аутентифікації дозволяє створити більш надійну і складну систему захисту. У якості факторів при БФА можуть використовуватися різні рішення. По-перше, це знання, або щось, що відомо лише користувачеві, наприклад, пароль або секретні питання. По-друге, володіння, яке передбачає наявність фізичного об'єкта або пристрою, як, наприклад, смартфон з встановленим додатком для генерування одноразових кодів, USB-токени, або смарт-карти. І, як приклад, приналежність, що стосується біометричних даних користувача, таких як відбитки пальців, розпізнавання обличчя чи сканування райдужної оболонки ока. Комбінація таких факторів значно посилює надійність системи, оскільки зловмиснику доведеться подолати не один, а кілька бар'єрів для несанкціонованого доступу.

БФА дозволяє вирішити низку питань, які виникають у розрізі безпеки. По-перше, такий підхід сильно знижує ризик компрометації облікового запису через фішингові атаки, брутфорс або крадіжку паролів [23]. Якщо навіть один з факторів (наприклад, пароль) буде скомпрометований, інші фактори, такі як код підтвердження, відправлений на мобільний телефон, або біометрична верифікація, залишаться непорушеними, забезпечуючи додатковий захист.

Крім того, багатофакторна аутентифікація допомагає у вирішенні проблеми «гнучкості»; у випадку виявлення підозрілої активності система може запросити додатковий фактор аутентифікації або тимчасово заморозити доступ до облікового запису, допоки не буде підтверджено особу користувача. Даний підхід є ефективним інструментом протидії загроз.

Хоча впровадження багатофакторної аутентифікації може виявитися дорожчим і тривалішим у реалізації, ніж однофакторні рішення, його переваги, безумовно, дозволяють забезпечувати високий рівень безпеки та захисту.

Власне, вибір оптимального методу аутентифікації є критично важливим завданням, яке стоїть перед організаціями та розробниками програмного забезпечення. Цей вибір залежить від множинних факторів, які потрібно ретельно оцінити, аби забезпечити баланс між безпекою, зручністю для користувачів та економічною доцільністю. На думку автора даної роботи, вибір механізму аутентифікації, можна здійснити, враховуючи наступні ключові аспекти: безпека та ризики, зручність для користувачів і вартість та технічні можливості.

Безпека є першочерговими фактором при розгляді можливих методів аутентифікації. Захист даних і забезпечення конфіденційності є основними цілями захисту інформації, особливо в умовах зростаючої кількості та складності кібератак. Вибір методу аутентифікації повинен враховувати потенційні загрози, специфічні для кожної організації. Такі технології, як багатофакторна аутентифікація, відразу привертають увагу, оскільки вони пропонують додаткові шари захисту. Проте варто зважати, що складні системи можуть бути надто затратними в контексті ресурсів і можуть не відповідати всім сценаріям, особливо якщо ризики для даних невисокі. Наприклад, для критичних систем, таких як фінансові або державні структури, має сенс обирати найбільш захищені варіанти, які б захистили від витоку або викрадення інформації.

Іншим важливим фактором є зручність користувацького досвіду. Навіть найнадійніша система безпеки може не принести бажаного ефекту, якщо вона буде надто складною у використанні. Якщо користувачам важко або незручно працювати з механізмом аутентифікації, це може призвести до пошуку обхідних шляхів, що знижує загальний рівень безпеки. Саме тому, зручність потрібно розглядати паралельно з безпекою. Системи, що використовують біометричні дані, такі як розпізнавання відбитків пальців або обличчя, є гарними варіантами з точки зору зручності, оскільки вони не потребують запам'ятовування паролів і досить швидкі в експлуатації. Проте, їх впровадження може бути обмежене через вимоги до спеціального обладнання.

Впровадження складних систем може вимагати значних фінансових вкладень, що включає вартість технологій, навчання персоналу, підтримку та обслуговування

системи. Малий та середній бізнес, який працює з обмеженими бюджетами, може знайти більш доцільним використання менш складних методів аутентифікації з додатковими заходами безпеки, такими як регулярна зміна паролів чи використання одноразових кодів безпеки. Важливо також оцінити, чи існуючі системи підтримують впровадження нового методу аутентифікації, чи потрібно здійснювати суттєве оновлення інфраструктури, що може вплинути на загальну продуктивність роботи та можливі витрати.

Найкращий підхід до аутентифікації може включати в себе використання декількох методів у різних комбінаціях, відповідно до оцінки ризиків, бюджетних можливостей та специфіки бізнес-процесів. Прийняття зваженого рішення стане гарантією не лише досягнення бажаного рівня безпеки, але і забезпечення зручності роботи та ефективності у витратах.

На тлі швидких технологічних змін вибір відповідних методів аутентифікації залишається критично важливим для забезпечення безпеки в різних середовищах. Розуміння особливостей кожного з середовища дозволяє підібрати відповідні системи захисту, які б задовольняли специфічні вимоги безпеки і зручності. Розглянемо приклади щодо вибору методів аутентифікації для трьох основних типів середовищ: корпоративні системи, приватні та персональні пристрої, а також інтернет-сервіси та мобільні додатки.

У корпоративних системах основний акцент має бути зроблено на багатошаровій безпеці й надійності. З огляду на те, що корпоративні дані часто є особливо цінними і конфіденційними, основною рекомендацією є впровадження багатофакторної аутентифікації. Вона може поєднувати у собі елементи, такі як паролі, одноразові коди, відправлені через захищені канали, і біометричну перевірку. Також можна використовувати технології перевірки на основі поведінки, які в режимі реального часу аналізують дії користувача і можуть вимагати додаткової аутентифікації при виявленні підозрілої активності. Іншим рішенням, яке може використовуватися у корпоративних системах є використання фізичних токенів та біометрії. Саме у корпоративному сегменті впровадити подібну систему легше за рахунок можливості зобов’язання працівників дотримуватися внутрішніх правил

компанії. Крім того важливо підтримувати регулярне оновлення систем безпеки, проводити навчання співробітників, а також впроваджувати політики керування паролями, які вимагатимуть використання складних паролів та їх регулярну зміну.

Для персональних пристроїв актуальним є баланс між зручністю і базовою безпекою. Власники таких пристроїв можуть скористатися біометричними методами аутентифікації, такими як відбитки пальців або розпізнавання обличчя, які є швидкими і надійними засобами підтвердження особи. У наші дні подібні рішення пропонуються у багатьох приладах, в свою чергу розмір ринку біометрії у користувацькій електроніці станом на 2024 рік сягнув майже 43 млрд. доларів США і очікується, що він буде лише зростати [24]. Додатковий рівень безпеки можна забезпечити за рахунок підтримки пристроями шифрування та можливості віддаленого блокування або видалення даних у разі втрати. Використання менеджерів паролів може значно спростити керування кількома обліковими записами, дозволяючи створювати і зберігати складні паролі без необхідності їх запам'ятовування.

Інтернет-сервіси та мобільні додатки стали невід'ємною частиною повсякденного життя, тому їх захист є пріоритетом. Для даного сегменту задля забезпечення достатнього рівня захисту можна поєднувати традиційні паролі з SMS-повідомленнями, електронними листами з кодами підтвердження або повідомленнями від push-додатків. Сервіси також можуть підтримувати захищені протоколи аутентифікації, такі як OAuth, для безпечності ідентифікації сторонніх додатків. Особливо в контексті бізнесу, для розробників важливо враховувати зручність для кінцевих користувачів, забезпечуючи простий і зрозумілий процес авторизації, який не відлякуватиме користувача від використання сервісу.

Усі ці рекомендації ґрунтуються на розумінні того, що жоден метод аутентифікації не може бути універсальним для всіх випадків. Вибір конкретного рішення залежить від задач, які стоять перед користувачами, рівня чутливості даних, бюджету та технічних можливостей. Виважений підхід до управління ідентифікацією та доступом дозволяє не лише адаптуватися до нових загроз, але й підвищити загальний рівень зручності та довіри користувачів системи.

1.3 Сучасні технології та підходи у аутентифікації

У попередньому підрозділі були розглянуті основні аспекти, що стосуються однофакторної та багатофакторної аутентифікації, але виникає питання, які способи аутентифікації існують загалом? Засобів, безумовно, велика кількість, але основними можна визначити наступні [25]:

- Використання пароля.
- Використання одноразових паролів.
- Вхід за допомогою ключа безпеки.
- Доступ під одним обліковим записом.
- Біометрична аутентифікація.

Паролі, як засіб аутентифікації, мають довгу історію розвитку, починаючи від свого зародження до сучасності. З поширенням комп'ютерних систем у 1960-х роках, паролі стали невід'ємною частиною захисту обчислювальних ресурсів. Як вже згадувалося раніше, перші комп'ютерні системи вимагали від користувачів вводити паролі для доступу до обчислювальних потужностей і файлів, і ця концепція досі залишається актуальною.

Еволюція паролів включала різні підходи до їхнього створення та використання в міру того, як технології та загрози розвивалися. Коли персональні комп'ютери почали з’являтися в офісах та домівках, паролі були одним з основних методів захисту даних. Розвиток обчислювальних потужностей техніки з часом призвів до необхідності створення складніших і більш довгих паролів, які було б важче викрити за допомогою атак грубої сили. Для прикладу, у наш час, згідно статистики, 70% слабких паролів можуть бути зламані лише за 1 секунду, використовуючи атаку грубої сили [26].

За період розвитку інформаційних технологій, підходи до створення та збереження паролів зазнали значних змін. Користувачі все частіше стикаються з вимогами створення складних паролів, які містять велику кількість символів, включаючи великі і малі літери, цифри та спеціальні символи, що стає бар'єром для злодіїв, які використовують методи підбору [27]. Такі паролі дозволяють знизити

ризик їх зламу. Однак зі зростаючою кількістю облікових записів, які вимагають паролі, управління ними стає складнішою задачею для користувачів. Тут на допомогу приходять менеджери паролів, які надійно зберігають та автоматично вводять паролі, спрощуючи процес їх використання [28].

Незважаючи на їхню розповсюдженість, використання паролів у чистому вигляді має суттєві недоліки. Однією з найсерйозніших проблем є повторне використання паролів одночасно для кількох сервісів, що може призвести до компрометації всіх акаунтів у разі зламу хоча б одного з них. Опитування Google показало, що щонайменше 65% людей повторно використовують паролі на декількох, якщо не на всіх, сайтах [29]. Крім того, такі інструменти як фішинг та соціальна інженерія в цілому є потужними інструментами в руках зловмисників, які намагаються отримати паролі шляхом обману користувачів [30].

Управління ризиками, пов'язаними з паролями, передбачає низку заходів безпеки. По-перше, важливо уникати використання простих та передбачуваних паролів. Необхідно визначити політику складних паролів та дотримуватися її. Крім того, рекомендується регулярно змінювати паролі та уникати їх повторного використання [31]. Крім того, як вже згадувалося раніше, використання багатофакторної аутентифікації дозволяє знизити ризик несанкціонованого доступу навіть у разі компрометації пароля.

Адміністратори систем повинні впроваджувати надійні методи захисту, як-от хешування паролів з використанням стійких алгоритмів, що може ускладнити злам паролів навіть у разі компрометації бази даних.

Якщо говорити про одноразові паролі, то можна сказати, що вони є важливим інструментом у сфері аутентифікації, який часто використовується як додатковий фактор та забезпечує додатковий рівень безпеки при доступі до інформаційних систем і сервісів. OTP (One-Time Password) – це код, який дійсний лише для одного сеансу входу або транзакції [32]. Їхня мета полягає у зменшенні ризику несанкціонованого доступу до облікових записів навіть якщо постійні паролі були скомпрометовані.

Генерація і використання OTP зазвичай здійснюється за допомогою алгоритмів, таких як HMAC-based One-Time Password (HOTP) та Time-based One-Time Password (TOTP). HOTP генерує паролі на основі лічильника і секретного ключа, відомого лише серверу і клієнту. Кожен виклик цього алгоритму збільшує значення лічильника, що робить кожен наступний пароль унікальним [33]. TOTP, у свою чергу, використовує поточний час як основу для генерації паролів, що робить їх актуальними лише протягом визначеного часового інтервалу [33]. Цей підхід забезпечує додатковий захист від атак типу «повторної передачі», дозволяючи отримати доступ лише тим користувачам, які можуть ввести коректний одноразовий пароль вчасно.

Існують також і інші підходи для створення одноразових паролів, зокрема генерація випадкових чисел чи використання підходу виклик-відповідь [33].

Що стосується програмних та апаратних засобів для роботи з OTP, існують різні рішення. Програмні засоби включають мобільні додатки, такі як Google Authenticator або Authy, які генерують OTP на основі протоколів HOTP або TOTP. Такі додатки легко використовувати, і вони забезпечують максимальну зручність, адже користувачеві не потрібно носити із собою додаткові пристрої. Апаратні рішення включають спеціальні токени, такі як YubiKey або RSA SecurID, які генерують OTP незалежно від інших пристроїв. Ці токени мають переваги з точки зору безпеки, оскільки менше підвладні ризикам компрометації через шкідливі програми на мобільних телефонах чи персональних комп'ютерах [34].

Переваги OTP очевидні: вони значно підвищують безпеку за рахунок унеможливлення повторного використання паролів, які могли бути перехоплені під час їх передачі або в результаті фішингових атак. Навіть якщо одноразовий пароль буде перехоплений, зловмисник не зможе використати його після закінчення короткого періоду його дії або після використання пароля самим користувачем.

Проте, існують і обмеження. По-перше, підтримка OTP може вимагати додаткових витрат на впровадження і підтримку систем, особливо якщо використовуються апаратні токени. По-друге, незважаючи на зручність програмних рішень, вони вимагають встановлення додаткових додатків на мобільних пристроях,

що робить отримання доступу у разі втрати або відсутності пристрою проблематичним.

Вхід за допомогою ключа безпеки є одним з найбільш надійних методів аутентифікації завдяки своїй стійкості до атак, таких як фішинг або атаки за методом «грубої сили» [35]. Безпековий ключ може підключатися до комп'ютерів або мобільних пристроїв через USB, NFC, або Bluetooth. У момент аутентифікації ключ гарантує шифрування обміну даними через криптографічні протоколи, такі як FIDO U2F (Universal 2nd Factor) та FIDO2 [35]. Це дозволяє користувачеві підтвердити свою особу шляхом простого фізичного дотику до ключа, що забезпечує зручність без втрати безпеки.

Смарт-карти, Touch Memory, та USB-ключі є різними типами апаратних ідентифікаторів, які мають свої особливості. Смарт-карти схожі на банківські картки та виготовляються з вбудованим мікропроцесором, який може зберігати криптографічні ключі та виконувати операції з ними [36]. Вони можуть використовуватися в системах, де потрібен високий рівень безпеки і гнучкості.

Touch Memory – це апаратні пристрої, які функціонують за принципом фізичного контакту; їх найчастіше застосовують в системах контролю доступу. USB-ключі, як і безпекові ключі, використовують USB-з'єднання для підключення до комп'ютерів. Вони можуть мати різний функціонал, включаючи зберігання криптографічних сертифікатів та генерацію одноразових паролів.

Фізична безпека цих пристроїв є одним з ключових завдань, щоб забезпечити їхню ефективність. Втрачений або викрадений пристрій може стати вразливим місцем у системі безпеки, тому важливо включати механізми захисту, такі як пін-коди або додаткові паролі для доступу до функцій ключа.

Забезпечення фізичної безпеки полягає також у створенні алгоритмів дій на випадок втрати або викрадення пристрою. Це може включати можливість видалення сертифікатів або ключів дистанційно, або блокування відповідних облікових записів до моменту заміни пристрою. Важливо враховувати, що фізичні ключі та апаратні ідентифікатори є не просто інструментами, але частиною комплексної системи безпеки, що інтегрує різні компоненти для боротьби із загрозами безпеки.

Безпекові ключі та апаратні ідентифікатори надають можливості для забезпечення надійної аутентифікації і захисту даних. Їхнє функціонування засноване на поєднанні сучасних криптографічних протоколів з фізичною безпекою, що робить їх привабливим вибором для захисту активів.

Розглянемо інший спосіб аутентифікації, а саме SSO, або система єдиного входу. Принципи доступу під одним обліковим записом базуються на ідеї централізації аутентифікації, де користувачам надається можливість проходити верифікацію тільки один раз, щоб отримати доступ до множинних додатків і сервісів [37]. Це вирішує одну з головних проблем – необхідність запам'ятовування різних паролів для кожного сервісу, які можуть створити ризик повторного використання менш надійних паролів. Використання єдиного облікового запису покликане підвищити безпеку, зменшивши вірогідність фішингу та соціальної інженерії, оскільки менша кількість аутентифікаційних дій означає менше можливостей для атак.

Єдиний вхід – це технологія, яка об'єднує кілька різних екранів для входу в додаток в один. За допомогою SSO користувачеві потрібно лише один раз ввести свої облікові дані (ім'я користувача, пароль тощо) на одній сторінці, щоб отримати доступ до всіх своїх додатків [38].

SSO часто використовується в бізнес-контексті, коли користувацькі додатки призначаються та управляються внутрішньою IT-командою [38]. Технології, такі як SAML (Security Assertion Markup Language) і OAuth (Open Authorization), є фундаментом для реалізації SSO систем. SAML дозволяє обмінюватися аутентифікаційними інформаційними повідомленнями між додатками, тоді як OAuth фокусується на легітимізації доступу до ресурсів без потреби передавати паролі [39, 40].

Переваги інтеграції доступу за допомогою SSO систем є значними: крім спрощення для користувачів, які отримують безшовний доступ до різних сервісів, SSO покращує адміністрування ідентифікацій та зменшує витрати на підтримку, оскільки централізоване управління полегшує додавання або видалення доступу, а також моніторинг активності. Більше того, зменшується кількість аналітичних точок

для контролю забезпечення безпеки, що полегшує виявлення та реагування на підозрілі дії.

Однак при використанні даного підходу є і недоліки. Основним є те, що система SSO повинна забезпечити високий рівень безпеки, оскільки компрометація єдиного облікового запису може відкрити доступ до всіх пов’язаних додатків. Крім того, впровадження SSO може бути технічно складним завданням, яке вимагає ретельного планування і правильної інтеграції з існуючими системами.

Загалом, інтеграція доступу до додатків шляхом реалізації єдиного облікового запису і використання систем єдиного входу має потенціал для значного покращення практик управління ідентифікацією в цифровому середовищі. Подібний підхід не лише підвищує зручність для користувачів, але й забезпечує кращий контроль з боку адміністративних команд.

Біометрична аутентифікація, як передова технологія в сфері кібербезпеки, відкриває нові горизонти для забезпечення безпеки та конфіденційності. Вона ґрунтується на унікальних фізичних або поведінкових характеристиках людини, що робить можливість підробки або викрадення таких даних значно складнішою в порівнянні з традиційними методами аутентифікації.

Біометрична аутентифікація визначається як захід безпеки, який відповідає біометричним характеристикам користувача, що бажає отримати доступ до пристрою або системи. Доступ до системи надається лише тоді, коли параметри збігаються з тими, що зберігаються в базі даних для цього користувача [41].

Типи біометричних даних, які використовуються для аутентифікації, є різноманітними і з кожним днем стають усе більш точними і надійними. Найпоширенішими є відбитки пальців, форма обличчя, сканування райдужки або сітківки ока, а також розпізнавання голосу [41]. Кожен із цих методів має свої переваги і недоліки, з погляду надійності, зручності використання і вартості реалізації.

Технологічні досягнення в біометрії призвели до значного підвищення точності ідентифікації та швидкості обробки даних. Сучасні системи можуть похвалитися мінімальним часом відповіді і високим рівнем точності завдяки

використанню технологій. Вони здатні адаптуватися до мінливих умов, таких як зміна освітлення або позиції користувача. Інтеграція біометричних систем у звичайні пристрої, на кшталт смартфонів, робить цю технологію більш доступною та поширеною.

Однак, разом з перспективами, біометрична аутентифікація створює низку викликів. По-перше, біометричні дані є невід'ємною частиною особи людини і не можуть бути змінені, як паролі чи картки доступу. Втрата чи компрометація таких даних може призвести до серйозних наслідків. Існують також занепокоєння щодо можливості надмірного стеження і втручання в особисте життя, адже біометричні системи можуть використовуватися не лише для аутентифікації, але й для ідентифікації і спостереження за людьми без їх згоди [42].

З правової точки зору, використання біометричних даних має відповідати нормативним вимогам і стандартам, які забезпечують захист особистих даних. У різних країнах ці норми відрізняються, але більшість з них передбачають прозорість у збиранні даних, отримання згоди користувачів і можливість видалення їхніх даних.

Висновки за розділом 1

У розділі було розглянуто різноманіття підходів до забезпечення безпеки та аутентифікації в інформаційних системах. Ґрунтуючись на основних концепціях і історичному розвитку багатофакторної аутентифікації, було проведено всебічний аналіз використання різних методів, їхніх переваг і викликів.

На початку було визначено концепцію багатофакторної аутентифікації та окреслено основні принципи, що забезпечують її ефективність. Було висвітлено історичний контекст і еволюцію цієї технології від використання простих паролів до впровадження складних багатокомпонентних систем, які комбінують кілька видів ідентифікаторів для підвищення рівня безпеки.

Подальша класифікація методів аутентифікації дозволила зрозуміти різницю між однофакторними та багатофакторними підходами. Однофакторна аутентифікація, яка базується на одному елементі, як-от пароль, хоча і забезпечує

простоту у використанні, поступається в захисті даних у порівнянні з багатофакторними системами. Багатофакторна аутентифікація об'єднує різні елементи захисту – від знання (паролі) і володіння (ключі безпеки) до біометричних даних – що значно ускладнює несанкціонований доступ.

Важливо зазначити, що кожен з методів, розглянутих у цьому розділі, має свої унікальні переваги та обмеження. Паролі, хоча і залишаються найпоширенішим засобом захисту, піддаються ризикам зламування та фішингових атак. Одноразові паролі підвищують безпеку за рахунок уникнення повторного використання, проте можуть бути незручними для користувачів. Вхід за допомогою ключа безпеки, а також системи єдиного входу (SSO), забезпечують зручність і високий рівень захисту, проте потребують значних ресурсів на їх впровадження та обслуговування.

Особливу увагу було приділено біометричній аутентифікації, яка, завдяки технологічним досягненням, пропонує один із найбільш надійних методів ідентифікації, хоч і використання даної технології має власні виклики. Біометрія забезпечує унікальність і практично унеможливлює підробку, проте вимагає додаткових заходів для захисту даних від компрометації.

Було розглянуто різноманіття технологій, які забезпечують безпеку та захист в інформаційних системах – від традиційних паролів до передових біометричних рішень. Кожна технологія має свої унікальні переваги, недоліки та сфери застосування. З розвитком технологій, інтеграція різних методів, таких як одноразові паролі, безпекові ключі та системи єдиного входу, стає невід'ємною частиною комплексної стратегії управління доступом. Водночас, слабкі місця у використанні цих технологій вимагають ретельного врахування при виборі рішення. Усе це підкреслює важливість адаптації до нових викликів та забезпечення балансу між безпекою та зручністю.

Таким чином, можна підкреслити важливість багатофакторної аутентифікації як ключової стратегії у забезпеченні інформаційної безпеки. Поєднання різних методів, адаптація їх до конкретних потреб і врахування динаміки технологічного та вразливого середовища є критично важливими для побудови надійних систем захисту. Кожна організація, яка прагне убезпечити дані, повинна ретельно зважити

всі переваги й ризики, пов'язані з обраними методами аутентифікації, що дозволить не лише підвищити рівень безпеки, але й забезпечити зручність і довіру користувачів.

РОЗДІЛ 2
АРХІТЕКТУРИ ТА ІНСТРУМЕНТИ ДЛЯ РОЗРОБКИ API З БАГАТОФАКТОРНОЮ АУТЕНТИФІКАЦІЄЮ

2.1 Огляд архітектур API

У сучасному світі інформаційних технологій API, або програмні інтерфейси прикладного програмування, стали фундаментальною складовою для розробки програмного забезпечення. Вони є своєрідними містками, які з'єднують різні частини складних програм, дозволяючи їм взаємодіяти та обмінюватися даними. API визначають правила і механізми, за допомогою яких різні програмні компоненти можна інтегрувати між собою. API можна уявити як своєрідні контракти між програмами: якщо один із компонентів дотримується умов, зазначених в API, то він може використовувати функціональність іншого компонента без зайвих складнощів.

API – це механізми, які дозволяють двом програмним компонентам взаємодіяти один з одним за допомогою набору визначень і протоколів [43]. API розшифровується як Application Programming Interface (інтерфейс прикладного програмування). У контексті API слово «додаток» означає будь-яке програмне забезпечення з певною функцією. Інтерфейс можна уявити як договір про надання послуг між двома додатками. Цей договір визначає, як вони спілкуються один з одним за допомогою запитів і відповідей. Документація до API зазвичай містить інформацію про те, як розробники повинні структурувати ці запити та, які відповіді можуть бути отримані [43].

Роль API у сучасних додатках складно переоцінити. У час, коли бізнес все більше орієнтується на технології й онлайн-послуги, API забезпечують гнучкість і швидкість розробки необхідних програмних рішень. Наприклад, у випадку розробки мобільного додатка, що взаємодіє із сервером, API може дозволити спросити обмін даними, що підвищує ефективність розробки та зменшує час виходу нового продукту на ринок.

Однак, для того щоб API справді принесли користь, необхідно враховувати важливість правильної архітектури додатків. Архітектура програмного забезпечення багато в чому визначає, наскільки ефективно та надійно працюватиме система в цілому. Правильно спроектована архітектура слугує основою для ефективності, масштабованості та безпеки системи, що особливо важливо в умовах постійно зростаючих вимог до продуктивності і доступності сучасних програм.

Ефективність архітектури відображається в здатності програми обробляти запити швидко і з мінімальними витратами ресурсів [44]. Якщо API спроектовані правильно, вони можуть значно оптимізувати обробку даних, зменшуючи затримки і навантаження на сервери. Це критично важливо для додатків, де швидкість відгуку є ключовим фактором.

Масштабованість стосується здатності системи обробляти зростаючі обсяги роботи без деградації продуктивності [45]. Завдяки API, можна легко розширювати функціональність додатка або інтегрувати його з іншими сервісами без значних змін у коді. Наприклад, сервіс, що росте та потребує обробки все більшої кількості даних клієнтів, може скористатися API для інтеграції із зовнішніми системами аналізу даних або підтримки клієнтів.

Безпека – це ще один аспект, який неможливо ігнорувати. API, які відкриті для зовнішнього світу, можуть стати мішенню для кібератак, якщо не вжити належних заходів захисту. Архітектура повинна враховувати способи аутентифікації і авторизації доступу, шифрування даних та інші методи захисту. Приміром, технології OAuth або JWT широко використовуються для забезпечення безпеки комунікацій через API, гарантують, що лише авторизовані користувачі можуть отримати доступ до чутливих даних.

На підставі вищесказаного можна зробити висновок, що API виступають не тільки як інструменти комунікації між різними програмними компонентами, але і як важливий елемент у проектуванні ефективних, масштабованих і безпечних сучасних додатків. Правильне проектування архітектури з врахуванням всієї складності та особливостей роботи з API сприяє досягненню високих стандартів у розробці програмного забезпечення, що в свою чергу веде до успіху в бізнесі.

Згідно інформації, яку збирає компанія Postman, що спеціалізується на розробці інструментів, що дозволяють спростити роботу з API, у 2023 році найбільш популярними технологіями, що використовувалися у розробці API були [46]:

- REST (86%)
- Webhooks (36%)
- GraphQL (29%)
- SOAP (26%)

Повні результати опитування представлені на рисунку 2.1.

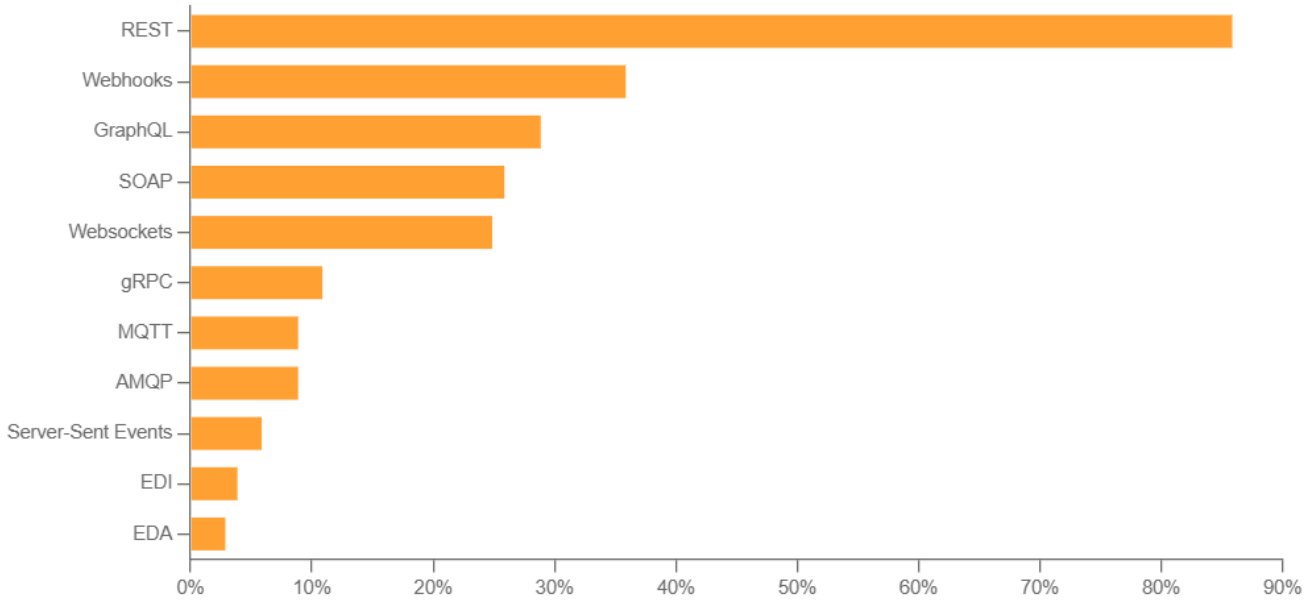

Рисунок 2.1 – Результати опитування Postman щодо популярності технологій API

Таким чином можна стверджувати, що REST є найпопулярнішим підходом при розробці API. Для більшого розуміння принципів і особливостей кожної із технологій, розглянемо деякі з них. Почнемо із найпоширенішої – REST.

REST (Representational State Transfer), або передача репрезентацій стану, є архітектурним стилем, який був запропонований Роєм Філдінгом у його дисертації у 2000 році [47]. Цей стиль здобув широку популярність завдяки своїй простоті та здатності забезпечувати ефективну взаємодію в розподілених системах. Історія REST почалася з необхідності створити стандарт, що дозволить клієнтам та

серверам ефективно спілкуватися, дотримуючись певного набору принципів, які гарантують надійність і гнучкість архітектури.

Основні принципи REST базуються на кількох ключових концепціях [48]:

- Розділення клієнта і сервера.
- Відсутність стану.
- Єдність інтерфейсу.
- Кешування.
- Багаторівнева архітектура.
- Код на вимогу.

Розглянемо їх детальніше. По-перше, важливим є чітке розділення клієнта і сервера. Ролі клієнта та сервера є окремими та незалежними одна від одної. Клієнт відповідає за користувацький інтерфейс та досвід користувача, тоді як сервер управляє даними та бізнес-логікою. По-друге, важливим принципом у REST є відсутність стану. Кожен HTTP-запит від клієнта до сервера повинен містити всю інформацію, необхідну серверу для виконання запиту. Сервер не може зберігати жодну інформацію про клієнта між запитами. Цей принцип дозволяє обробляти кожен запит незалежно [49].

REST визначає уніфікований інтерфейс між клієнтом та сервером для спрощення та роз’єднання архітектури. Це включає використання стандартних HTTP-методів (GET, POST, PUT, DELETE), взаємодії без збереження стану та URI (Uniform Resource Identifier), що базуються на ресурсах [49].

Іншою важливою властивістю є можливість кешування відповідей. Відповіді сервера мають бути чітко позначені як кешовані або некешовані. Якщо відповідь може бути кешована, клієнти можуть зберігати її і використовувати для майбутніх запитів, зменшуючи навантаження на сервер і затримки. Ще одним принципом є багаторівнева архітектура, що розуміє під собою те, що клієнт не повинен мати можливості визначити, чи підключений він безпосередньо до кінцевого сервера або до проміжного елемента. Це сприяє масштабованості та керованості через балансування навантаження, спільні кеші та інші проміжні елементи [49].

Принцип коду на вимогу є не обов’язковим. Під ним розуміється, що сервери можуть за бажанням надсилати виконуваний код клієнтам для розширення їх функціональності. Наприклад, клієнту може бути надісланий JavaScript код для виконання в браузері, що дозволяє динамічну поведінку без подальшої взаємодії з сервером [48].

Окремо варто відмітити, що ресурси представлені таким чином, щоб відображати їх стан, зазвичай у форматах JSON, XML або HTML. Клієнти взаємодіють з цими поданнями для управління станом ресурсів.

Як вже було згадано, принцип відсутності стану, що лежить в основі REST, означає, що кожен запит від клієнта до сервера має містити всю необхідну інформацію для розуміння та обробки запиту. Сервер не зберігає стан сесії між запитами, що робить REST додатки особливо зручними для обробки великої кількості запитів. Такий підхід спрощує горизонтальне масштабування, адже кожен сервер може бути незалежно обробити будь-який запит без потреби знати про попередні запити клієнта.

Використання HTTP методів є одним з фундаментальних елементів REST архітектури. Ці методи забезпечують стандартизовані способи взаємодії з ресурсами. Метод GET використовується для отримання представлення ресурсу без зміни його стану. Це ідеально підходить для запитів, які потребують лише відображення інформації. POST використовується для створення нових ресурсів на сервері. Наприклад, при створенні нового користувача в базі даних може використовуватися саме POST запит. Метод PUT здебільшого застосовується для оновлення існуючих ресурсів – якщо потрібно змінити інформацію про користувача, клієнт може надіслати PUT запит з новими даними. DELETE, в свою чергу, надає можливість видалити ресурс. Цей метод, як випливає з назви, сигналізує про необхідність усунути певний об’єкт з системи [50].

URI в контексті REST використовується для ідентифікації певних ресурсів у веб-додатках [51]. Кожен ресурс у системі REST має унікальний URI, за допомогою якого можна виконати необхідні операції, використовуючи описані HTTP методи. Наприклад, уявімо REST API для онлайн-магазину. Ресурс, що представляє певний

товар, може бути ідентифікований URI на кшталт /products/123, де 123 – це унікальний ідентифікатор товару. Використовуючи відповідні методи HTTP, клієнт може отримати деталі товару, створити новий товар, оновити інформацію або видалити його з бази.

Таким чином, REST не тільки задає правила і механізми для ефективної і надійної взаємодії між компонентами однієї системи чи різних систем, але й спрощує розробку, масштабування та управління веб-додатками завдяки своїй простоті та логічності. Це робить REST одним з найбільш популярних архітектурних стилів у сучасному розробці веб-сервісів.

На другому місці згідно опитування Postman, що згадувалося раніше, стоїть технологія webhooks (веб-хуки), тож розглянемо її детальніше.

Webhooks є ключовим елементом у сучасній інтеграції веб-додатків і стали невід'ємною частиною багатьох технічних рішень, що відзначаються своєю здатністю впроваджувати реактивні механізми автоматизації. На відміну від традиційних API, які вимагають від клієнта постійних запитів для отримання актуальних даних, webhooks пропонують більш ефективний і проактивний метод обміну інформацією між системами.

Замість того, щоб надсилати інформацію у відповідь на запит іншого додатка, webhooks надсилає інформацію або виконує певну функцію у відповідь на тригер, наприклад, час доби, натискання кнопки або отримання форми. Webhooks часто називають зворотними API, оскільки вони покладають відповідальність за комунікацію на сервер, а не на клієнта. [52].

Концепція webhooks базується на подіях, які відбуваються в одній системі, і автоматичній передачі інформації про ці події іншим зацікавленим системам. Це можна уявити, як підписку на новини: замість того, щоб постійно перевіряти новинний сайт в очікуванні нових статей, ви підписуєтеся на повідомлення, які вам автоматично надходять, коли з'являється новий контент. Webhooks працюють аналогічно: коли в системі відбувається певна подія, така як зміна статусу замовлення або додавання нового елемента, система відразу ж відправляє HTTP

запит на заздалегідь визначену URL-адресу сторонньої системи, повідомляючи її про зміни.

Незважаючи на свою назву, webhooks не є API у звичайному розумінні; вони працюють разом. Для використання webhook, програма повинна мати API. Щоб налаштувати webhook, клієнт надає унікальну URL-адресу API сервера і вказує, про яку подію він хоче дізнатися. Після налаштування webhook клієнту більше не потрібно робити запит на сервер; сервер автоматично відправить відповідне корисне навантаження на URL-адресу клієнтського webhook, коли відбудеться вказана подія [53].

Webhooks є реактивним рішенням для автоматизації, оскільки вони дозволяють системам негайно реагувати на події без постійних запитів даних. Це не тільки знижує навантаження на ресурси сервера, але й забезпечує надзвичайно швидкі оновлення інформації. Така реактивна модель взаємодії є надзвичайно цінною в багатьох сферах, включаючи електронну комерцію, фінанси, управління клієнтськими відносинами і багато інших.

Один з ключових моментів у роботі webhooks – це повідомлення про події між сервісами. Коли подія відбувається, система-ініціатор відправляє повідомлення у вигляді HTTP POST запиту на URL-адресу одержувача. Це повідомлення зазвичай містить всі необхідні дані про подію, і сервер-одержувач може відповідно реагувати на ці події, оновлюючи внутрішні дані, ініціюючи додаткові процеси або змінюючи статус ресурсів.

Сценарії використання webhooks є різноманітними. Один з найпоширеніших випадків – це оповіщення про зміни в ресурсах, таких як зміна статусу замовлення в інтернет-магазині. Наприклад, як тільки замовлення переходить у статус "відправлено", система автоматично відправляє webhook для повідомлення зовнішньої системи доставки про цю подію. Завдяки цьому кур'єрська служба може негайно отримати інформацію і почати обробку доставки без затримок. Існують також численні інші сценарії використання, такі як інтеграція з платіжними сервісами, управління інвентарем або синхронізація даних між різними CRM-системами.

Webhooks мають кілька переваг. По-перше, вони ефективні та швидкі, оскільки усувають потребу в постійних запитах або оновленні для перевірки наявності нових даних. Webhooks, надсилають дані абоненту, як тільки відбувається подія, що зменшує затримку і покращує взаємодію з користувачем. По-друге, webhooks є гнучкими у налаштуванні, оскільки вони дозволяють видавцю і підписнику визначати події, дані і формат, якими вони хочуть обмінюватися. Webhooks також можуть підтримувати кілька підписників на одну подію, що дозволяє інтегрувати та автоматизувати різні веб-додатки [54].

Однак, webhooks також мають свої недоліки. Одним із головних викликів є забезпечення надійності та безпеки комунікацій. Оскільки webhooks ініціюють зовнішні запити, існує ризик маніпуляцій з даними або перехоплення конфіденційної інформації. Щоб уникнути цього, необхідно використовувати HTTPS для шифрування даних, а також реалізовувати механізми аутентифікації, такі як токени безпеки або інші види криптографічного захисту. Ще однією проблемою є обробка великих обсягів подій, що може перевантажувати систему, особливо якщо виклики вимагають складних обчислювальних процесів.

Загалом, webhooks є потужним інструментом у світі веб-розробки, що надає можливість створювати ефективні, інтерактивні механізми обробки подій. Правильно впроваджені webhooks можуть значно покращити взаємодію між системами, надаючи своєчасну і точну інформацію, тим самим прискорюючи бізнес-процеси та підвищуючи продуктивність.

Перейдемо до наступної API-технології, а саме GraphQL (Graph Querry Language). GraphQL, створений компанією Facebook у 2012 році і випущений у відкритий доступ у 2015 році, є мовою запитів для взаємодії з API, яка пропонує альтернативу традиційним REST-інтерфейсам. Основна мета створення GraphQL полягала в тому, щоб забезпечити більш гнучкий, ефективний і зручний спосіб запиту та взаємодії з даними. Історія GraphQL розпочалася з того, що інженери Facebook прагнули вирішити проблеми, які виникали у Facebook додатках, особливо на мобільних платформах, де обмеження на трафік і швидкість з'єднання вимагали оптимізації обміну даними [55].

Офіційний сайт GraphQL надає наступний опис для даної технології: GraphQL – це мова запитів для API і середовище виконання для виконання цих запитів з наявними даними. GraphQL надає повний і зрозумілий опис даних у API, дає клієнтам можливість запитувати саме те, що їм потрібно, і нічого більше, полегшує розробку API з часом і надає потужні інструменти для розробників [56].

Основні концепції GraphQL включають наступні ключові елементи [57]:

- запити (queries);
- мутації (mutations);
- схеми (schema);
- резолвери (resolvers).

Запити дозволяють клієнту отримувати дані з сервера, вибираючи лише ті поля, які йому дійсно потрібні, що уникає проблеми надмірного обсягу даних, типового для REST API. Мутації слугують для зміни або оновлення даних на сервері, аналогічно до POST, PUT або DELETE методів в REST.

Схема – це набір типів GraphQL. GraphQL використовує її для опису форми доступних даних. Схема є одним з ключових понять при роботі з GraphQL API. Вона визначає, як клієнт може запитувати дані, і визначає можливості API. Розв’язувач, в свою чергу, це функція, яка генерує відповіді на запити та мутації GraphQL, діючи як обробник запитів GraphQL. У схемі GraphQL визначено, що певний запит використовуватиме саме цей обробник, тому він відповідає за отримання даних, обробку даних, а потім перетворення отриманих і оброблених даних у формат масиву GraphQL. Після цього він повертає результати у вигляді функції, яку можна викликати.

GraphQL також дозволяє використовувати так звані підписки, які дозволяють клієнтам отримувати реальні оновлення, коли дані змінюються на сервері, що є особливо корисним для інтерактивних додатків, таких як чати або соціальні мережі [58].

Як і у кожна технологія, GraphQL має свої сильні та слабкі сторони. Розглянемо їх детальніше [59].

Однією з головних переваг GraphQL є можливість ефективного отримання даних. Клієнти можуть визначати точний обсяг інформації, необхідної їм, що уникає проблем перевантаження або обмеженості даних, характерних для RESTful API. Це дозволяє запитувати лише ті дані, які дійсно потрібні, що робить обмін інформацією більш цільовим і ощадливим.

GraphQL також використовує декларативний підхід до запитів, що дозволяє легко виражати складні потреби в даних без заглиблення в технічні деталі їх отримання. Це спрощує процес отримання і аналізу даних, роблячи його більш інтуїтивно зрозумілим для розробників.

Ще одна важлива особливість – це сильно типізована схема, яка виступає чітким планом структури даних і їх типів для сервера та клієнта. Це усуває можливу двозначність, характерну для традиційних REST API, і сприяє безперебійній комунікації між різними частинами застосунка.

GraphQL також дозволяє об'єднувати дані з різних джерел в одну точку доступу, що спрощує процес отримання інформації і робить його більш зручним. Ще однією перевагою є можливість оновлення та міграції структур GraphQL без шкоди для працездатності існуючих клієнтів.

Втім, використання GraphQL має і певні недоліки. Серед них варто відзначити складність налаштування сервера і схеми, яка є безумовно вищою, ніж в RESTful API. Створення оптимізованої системи вимагає знайомства з концепціями GraphQL і дотримання найкращих практик, створюючи дещо круту криву навчання.

Хоча GraphQL суттєво знижує проблему отримання надмірної кількості даних, вона не зникає зовсім, тому слід обережно підходити до оптимізації запитів і організації схеми.

Додаткової уваги також потребують питання безпеки. Можливі ризики пов'язані з надмірною «глибиною» запитів або атаками, які експлуатують складність запитів. Для запобігання вразливостей необхідно застосовувати засоби захисту, такі як обмеження швидкості і валідацію запитів.

У підсумку, можна сказати, що GraphQL став потужним і зручним інструментом для розробки сучасних веб-інтерфейсів, особливо у випадках, коли

гнучкість і точність запитів є ключовими вимогами. Він забезпечує оптимальний потік даних між клієнтами та серверами, проте потребує ретельного планування та роботи над питаннями безпеки й продуктивності. Популярність даної технології продовжує зростати, хоча безумовним лідером все ще залишається архітектура REST.

На завершення можна зробити кілька важливих висновків. У сучасному динамічному середовищі розробки веб-додатків знання і розуміння принципів роботи API є критично важливими для розробників, архітекторів програмного забезпечення та компаній в цілому. Кожна з розглянутих технологій – REST, webhooks і GraphQL – має свої унікальні особливості, які роблять її незамінною в певних сценаріях.

REST залишається однією з найпопулярніших архітектур завдяки своїй простоті та надійності. Вона ідеально підходить для побудови масштабованих і розподілених систем, де важлива чітка і зрозуміла взаємодія між клієнтами та серверами. Одночасно, webhooks забезпечують ефективну реактивну модель комунікації, що дозволяє системам миттєво реагувати на події, оптимізуючи роботу і знижуючи затрати на ресурси.

GraphQL, у свою чергу, приніс новий підхід до обміну даними, надаючи клієнтам можливість гнучко вибирати саме ті дані, які їм потрібні, що значно підвищує ефективність розробки, дозволяючи створювати багатофункціональні додатки з мінімальними затратами часу і ресурсів.

Проте, кожна з цих технологій має свої недоліки, що потребує обережного підходу при виборі конкретного рішення для певної задачі. Враховуючи рівень складності та специфічні вимоги до безпеки і продуктивності, необхідно ретельно аналізувати особливості кожної з підходів перед впровадженням.

При виборі технологічного рішення варто керуватися не лише поточними потребами проєкту, а й перспективою його розвитку в довгостроковій перспективі. Вартість впровадження та подальшої підтримки системи, наявність досвідчених фахівців – ключові фактори, які впливають на кінцевий успіх.

Загалом, розуміння і грамотне застосування цих технологій API дозволяє ефективно будувати зручні, надійні та масштабовані програми, які задовольняють потреби бізнесу і користувачів. Вибір правильного підходу визначає успіх розробки, що у свою чергу впливає на довіру користувачів і конкурентоспроможність сервісу на ринку.

2.2 Мови програмування та фреймворки для розробки API

API є однією з ключових складових сучасних систем. Вибір мови програмування та фреймворку для створення API значною мірою впливає на ефективність, продуктивність, надійність та підтримку кінцевого продукту. Саме тому тема вибору відповідних технологій є надзвичайно актуальною у контексті створення якісних та масштабованих рішень.

Вибір мови програмування та фреймворку для розробки API вимагає глибокого розуміння специфіки проекту та цілей, які він переслідує. Наприклад, такі мови програмування, як JavaScript, Python, Java чи Go, набули значної популярності в даній галузі завдяки своїм властивостям та особливостям. JavaScript, разом із фреймворком Node.js, надає можливість розробляти серверні частини веб-додатків завдяки своїй асинхронності та легкості використання. З іншого боку, Python зі своїм Flask або Django забезпечує швидкість розробки та читабельність коду, що робить його ідеальним для побудови прототипів і простих API. Java, відома своєю надійністю та стабільністю, часто використовується у великих корпоративних системах, а фреймворк Spring Boot додає гнучкості та функціональності [60].

Важливим аспектом у виборі мови програмування є продуктивність, яка може суттєво відрізнятися залежно від обраної технології. Наприклад, Go відрізняється високою швидкістю виконання завдань і низькою латентністю, що робить його відмінним вибором для високонавантажених систем [60]. Проте, вибираючи мову програмування, не варто зосереджуватись лише на продуктивності. Надійність коду, можливість його тестування та розширення також мають велике значення.

Роль надійності при виборі технологій не можна недооцінювати. Надійність системи визначає її здатність до стабільного функціонування під час робочих

навантажень та правильного оброблення помилок. Забезпечення надійності зазвичай пов'язане з правильним вибором архітектури, методів обробки винятків та тестування. Фреймворки, такі як Ruby on Rails, Django, або Spring Boot, надають вбудовані функції для забезпечення стабільності та обробки помилок, що спрощує завдання розробника.Отже, можна підсумувати, що вибір мови програмування та фреймворку для створення API є багатогранним завданням, що потребує ретельного аналізу всіх аспектів від продуктивності до надійності та підтримки. Це один з ключових етапів у процесі розробки програмного забезпечення, що здатний вплинути на успіх кінцевого продукту.

Розглянемо детальніше, які мови програмування та фреймворки використовуються для розробки REST API, як найбільш поширеного рішення.

У таблиці 2.1 представлено популярні мови програмування та відповідні фреймворки, які є популярними рішеннями для створення REST API [60].

Таблиця 2.1

Мови програмування та фреймворки для розробки REST API

Мова	Фреймворки
JavaScript	Express, Hapi, LoopBack, Sails
Python	Django REST Framework, Flask-RESTPlus, Falcon, FastAPI
Java	Spring Boot, RESTEasy, RESTlet, Jersey
PHP	Laravel, Slim, CakePHP, Lumen
C#	ASP.NET Web API, ServiceStack, NancyFx
Golang	Gin, Beego, Revel, Gorilla

Кожен з цих фреймворків надає свій власний набір переваг для розробки REST API, від прискорення процесу розробки до пропозиції широкого спектру функцій, які можуть бути адаптовані до конкретних потреб проекту.

Розглянемо популярні мови програмування, що використовуються для створення API, зокрема JavaScript/TypeScript, Python, Java, Ruby, C# і Go, а також порівняємо їх переваги та недоліки.

JavaScript/TypeScript є однією з найпопулярніших мов програмування завдяки своїй універсальності та широкому застосуванню на стороні клієнта та сервера.

Завдяки Node.js JavaScript перетворився на багатофункціональну мову серверної розробки, що забезпечує високу продуктивність завдяки неблокуючій архітектурі вводу/виводу [61]. TypeScript, в свою чергу, пропонує всі можливості JavaScript, а також додатковий функціонал, а саме систему типів [62]. Основними перевагами JavaScript є [63]:

- швидкодія;
- можливість зменшення навантаження на сервер;
- простота використання;
- універсальність;
- широкий функціонал;
- розповсюдженість;
- незалежність від платформи;
- велика кількість багатофункціональних фреймворків;
- регулярні оновлення.

До слабких сторін даної мови можна віднести безпеку на стороні клієнта, засоби дебагінгу, підтримку великої кількості браузерів, що вимагає адаптації продукту під різні версії та вендорів, а також повну зупинку рендерінгу у випадку виникнення будь-якої помилки [63].

Python – це мова, яка завоювала популярність завдяки своїй простоті, читабельності синтаксису та велику кількість бібліотек для різних задач. Дана мова може використовуватися у різних галузях – від скриптингу до машинного навчання та веб-розробки [64]. У контексті API використовуються такі бібліотеки як Flask, Django або FastAPI, які мають велику кількість інтегрованих функцій та дозволяють швидко закласти основу для проекту.

До переваг Python можна віднести простоту використання, читабельність та велику спільноту розробників. Однак також існують і деякі обмеження, такі як слабша продуктивність порівняно з компільованими мовами, проблеми з керуванням пам'яттю, динамічною типізацією та сумісністю з версіями [65].

Іншою поширеною мовою для розробки API є Java. Ця мова надає безліч рішень для покращення функціональності та продуктивності API, наприклад, паралелізм, який дозволяє ефективно обробляти численні запити одночасно [66].

Мова Java відома своєю простотою, надійністю та функціями безпеки, що робить її популярним вибором для додатків корпоративного рівня, зокрема і для API. Програми на Java компілюються у байт-код, який можна запускати на будь-якій віртуальній машині Java. Синтаксис Java подібний до C/C++. Java спрощує написання, компіляцію та налагодження програм. Вона допомагає створювати багаторазовий код і модульні програми [67].

Сильними сторонами є [68]:

- незалежність від платформи;
- розподіленість;
- безпека;
- розподіл пам'яті;
- багатопоточність;
- забезпечення автоматичної «збірки сміття».

Незважаючи на свої сильні сторони, Java, як і будь яка інша мова, має власні слабкі місця, які проявляються у [68]:

- керуванні пам’яттю;
- необхідності значної кількості апаратних ресурсів
- відносно непростому синтаксисі
- довгому часі запуску програм.

PHP – це серверна мова сценаріїв з відкритим вихідним кодом, розроблена спеціально для веб-розробки. І хоча першочергово вона створена для веб-рішень, PHP також використовується як мова програмування загального призначення. На відміну від клієнтських мов, таких як JavaScript, які виконуються в браузері користувача, PHP-скрипти виконуються на сервері. Результати потім надсилаються у веб-браузер клієнта у вигляді звичайного HTML [69].

Сильні сторони PHP [70]:

- відкритий вихідний код;

- незалежність від платформи;
- масштабованість.

У той же час слабкими сторонами є [70]:

- обмеженість використання;
- слабка безпека;
- обмежені інструменти для відладки.

Іншою популярною мовою є C# у поєднанні з .NET Core широко використовується для розробки API. C# (сі-шарп) – це об'єктно-орієнтована мова програмування, розроблена компанією Microsoft в рамках платформи .NET. Вперше вона була випущена у 2000 році і з того часу стала однією з найбільш широко використовуваних мов для створення Windows-додатків, веб-сервісів та іншого. C# поєднує в собі потужність C і C++ із простотою мов, таких як Java та Visual Basic [71].

Переваги C# [72]:

- частина платформи .NET;
- вбудований збирач сміття;
- забезпечує безпеку типів;
- розгорнута документація.

Недоліки C# [72]:

- продуктивність;
- залежність від платформи .NET;
- складність вивчення.

Go (Golang) – це відносно нова мова програмування з відкритим кодом, розроблена компанією Google. Go створена для забезпечення простоти, ефективності та надійності. Вона здобула популярність завдяки своїй здатності справлятися з сучасними обчислювальними викликами. Дана мова поєднує у собі переваги продуктивності та безпеки мов C і C++ з простотою використання мов, таких як Python і JavaScript [73].

Переваги Golang:

- продуктивність;

- активна спільнота та документація;
- масштабованість;
- широкий спектр застосувань.

Недоліки Golang:

- молода мова;
- відсутність підтримки генералізованих функцій;
- менша спільнота та менша кількість навчальних матеріалів ніж у старіших мов.

Мова Golang є перспективною і підходить для вирішення задач, пов'язаних із продуктивністю та масштабованістю. Однак, через свою молодість, вона має певні обмеження, які варто враховувати.

Однак, варто пам’ятати, що мова є лише інструментом. При розробці будь-яких рішень багато чого вирішують саме фреймворки, які орієнтовані на спрощену імплементацію певного функціоналу. Тому у контексті створення API важливо детальніше розглянути не лише самі мови програмування, а і фреймворки, які можуть бути використані.

Express.js є представником легких фреймворків для веб-застосунків на Node.js, який надає широкий спектр функцій, необхідних для створення API та веб-програм в цілому. Завдяки цим функціям фреймворк вважається одним з найбільш зручних серед JavaScript-розробників [74].

Однією з ключових переваг Express.js є його розширена підтримка так званих проміжних обробників. Ці обробники є важливою частиною будь-якої програми, оскільки дозволяють програмісту ефективно управляти запитами і відповідями або їхньою взаємодією в модульний спосіб. Це включає допомогу в таких загальних завданнях, як аутентифікація, логування, обробка помилок і парсинг вмісту [75]. Крім того, Express.js має велику і активну спільноту користувачів, що дозволяє використовувати велику кількість бібліотек і навчальних матеріалів [74].

Express.js також отримує додаткову підтримку завдяки своїй легкій інтеграції з іншими JavaScript-технологіями чи фреймворками, що робить його ідеальним

вибором для розробників, які вже використовують JavaScript. Узгодженість між чотирма рівнями стеку є вигідною у розробці та підвищує ефективність.

Django REST Framework (DRF) – це фреймворк Python для створення веб-API, які взаємодіють із веб-застосунками або сервісами, і є розширенням стандартного фреймворку Django. DRF відомий своєю здатністю інтегрувати повний набір функцій з вбудованою підтримкою численних можливостей, що робить процес розробки ефективних і захищених API дуже простим [76].

Однією з ключових переваг DRF є його власний інструментарій аутентифікації та прав доступу, що включає аутентифікацію на основі токенів, сесійну аутентифікацію і підтримку OAuth [77]. Ці можливості роблять API на основі DRF стійкими до загроз і здатними підтримувати численні сценарії аутентифікації ще на етапі розробки.

Як компонент Django, DRF дає змогу розробникам повною мірою використовувати потужність ORM та інший інструментарій Django під час створення API. Ця інтеграція дозволяє розробляти стійкі API, які при цьому залишаються легкими в управлінні.

Spring Boot – це фреймворк для створення застосунків на основі мови програмування Java. Він розроблений на базі Spring Framework і надає всі його можливості, а також додаткову зручність. Однією з особливих переваг використання Spring Boot є можливість створення незалежних застосунків з мінімальною конфігурацією [78].

Spring Boot також має відмінну підтримку для реалізації RESTful API. Він оснащений інтегрованими інструментами для розробки, захисту та архівування API, що робить його придатним для використання великими компаніями, які шукають стабільні та стійкі рішення для API. У рамках Spring Boot доступні багато інших фреймворків, таких як Spring Security, Spring Data, Spring Cloud та інші [79]. Такий широкий спектр можливостей дозволяє розробникам створювати застосунки з численними функціями та можливостями, здатними працювати з складною бізнес-логікою або великим навантаженням.

При використанні мови PHP, для розробки веб-додатків, популярним є фреймворк Laravel. Він відомий своєю продуктивністю та надає розробникам набір інструментів і ресурсів, які значно спрощують процес створення веб-додатків, роблячи його більш структурованим [80].

Однією з ключових переваг Laravel є його спроможність забезпечити швидкий старт проекту завдяки вбудованим можливостям, таким як маршрутизація, сесії, кешування та просунутий механізм шаблонів Blade. Laravel також має потужну систему ORM під назвою Eloquent, яка забезпечує зручну роботу з базами даних, перетворюючи складні SQL-запити в простий і зрозумілий код. Крім того, Laravel підтримує міграції баз даних, що полегшує управління змінами в структурі даних [81].

Завдяки потужній системі бібліотек і великій спільноті, Laravel забезпечує розробникам доступ до безлічі готових рішень і розширень, які дозволяють швидко додавати нові функціональні можливості.

Перейдемо до останнього фреймворку, який буде розглянуто у даному підрозділі, а саме Gin. Це високопродуктивний фреймворк для створення веб-додатків і API на мові програмування Go (Golang). Його розроблено з акцентом на простоту, швидкість та ефективність, що робить його відмінним вибором для розробників, які прагнуть створювати легкі, але потужні веб-рішення [82].

Перевагою Gin є його швидкість. Завдяки мінімальному перевантаженню та чіткій архітектурі, Gin забезпечує високу продуктивність і здатний обробляти велику кількість запитів [82]. Gin надає дуже простий і зрозумілий інтерфейс для маршрутизації запитів. Він підтримує групи маршрутів, проміжне програмне забезпечення для обробки запитів на різних етапах їхнього життєвого циклу та має вбудовані засоби для роботи з JSON, що значно полегшує розробку RESTful API.

Підтримка обробки помилок і відновлення від панік (panic recovery) дозволяє зберегти стабільність програми навіть у випадку виникнення несподіваних помилок [82].

Gin також вирізняється розширюваністю завдяки можливості інтеграції з іншими бібліотеками та фреймворками Go. Це дає змогу розробникам адаптувати фреймворк під свої конкретні потреби.

Для розробки API можна використовувати безліч мов програмування та фреймворків, кожен з яких має свої унікальні особливості та переваги. Вибір залежить від конкретних вимог проекту, технічної обізнаності команди та умов використання. Незалежно від обраного підходу, важливо розуміти можливості та обмеження кожного рішення, щоб створювати ефективні та надійні продукти, при цьому важливо пам’ятати, що мова – це лише інструмент для реалізації.

2.3 Підходи для впровадження багатофакторної аутентифікації

Багатофакторна аутентифікація може використовуватися у різний спосіб залежно від бажаного балансу між безпекою та зручністю. Як вже згадувалося раніше, рішення з використанням БФА робить систему більш безпечною, оскільки при застосуванні більше ніж одного фактора аутентифікації надається сильніший доказ ідентичності користувача.

БФА слід розглядати як підхід, а не конкретне рішення. Фактори, які використовуються, конфігурація БФА, або для кого вона примусово застосовується, залежать від конкретного сценарію використання. При розробці рішення БФА слід починати з аналізу сценарію використання та розуміння вимог і обмежень.

Коли ви керуєте системою, де дозволяєте всім реєструватися, ви матимете мало інформації та контролю над системами та пристроями, які використовують ці користувачі. Ви не знатимете, який у них рівень навичок у використанні більш складних факторів аутентифікації або чи мають вони легкий доступ до додаткового обладнання (наприклад, ключів безпеки). Це означає, що зазвичай ви дозволите людям вибирати, чи використовувати БФА і які саме фактори застосовувати. Однак іноді ви можете управляти сервісом, що містить конфіденційні дані, або мати сервіс, розрахований на технічно підкованих користувачів. У такому випадку може бути доцільним примусове застосування БФА, навіть якщо це означатиме втрату частини

вашої користувацької бази – безпека даних користувачів повинна мати пріоритет над максимізацією кількості реєстрацій.

Коли необхідно керувати системою, яка дозволяєте реєструватися всім бажаючим, інформація та контроль над системами та пристроями, які використовують ці користувачі майже відсутні. Відсутня інформація про те, який у них рівень навичок у використанні більш складних факторів аутентифікації або чи мають вони легкий доступ до додаткового обладнання (наприклад, ключів безпеки). Це означає, що бажано дозволити людям вибирати, чи використовувати БФА і які саме фактори застосовувати. Однак у деяких випадках, сервіс може містити конфіденційні дані, або бути розрахований на технічно підкованих користувачів. У такому випадку може бути доцільним примусове застосування БФА, навіть якщо це означатиме втрату частини користувацької бази, оскільки безпека даних користувачів наявних користувачів повинна бути пріоритетом.

У ситуаціях, коли необхідно контролювати доступ співробітників до корпоративних систем, можна запровадити БФА. Наприклад, можна забезпечити, щоб під час прийому нового працівника для нього заздалегідь налаштовувався БФА. Можливо також використовувати більш надійні рішення безпеки та надавати всім співробітникам ключі безпеки.

Якщо система дозволяє доступ до «чутливих» даних, наприклад медичних чи фінансових, коли необхідно ретельно перевірити особу користувача, БФА використовується не лише для посилення безпеки доступу, але й для того, щоб впевнитися, що користувач, який отримує доступ до системи, дійсно є тією особою, за яку себе видає.

У попередньому розділі розглядалися різні типи факторів, зокрема те, що людина знає, те, що людина має та те, чим людина є. Важливим аспектом багатофакторної аутентифікації є те, що фактори повинні поєднуватися з принаймні двох різних груп. Наприклад, запит користувача на введення двох різних паролів, хоч формально і можна вважати БФА, але все ж використовувати не варто.

Розглянемо декілька можливих підходів до БФА [83]:

Перший підхід до БФА – це підхід «завжди ввімкнено». Він передбачає, що користувач має використовувати два або більше фактори під час кожного входу. І хоча з точки зору безпеки це може бути ефективним рішенням, з точки зору зручності використання такий підхід має недоліки. Користувачів часто відлякують часті запити на введення додаткових факторів, і не всі дані потребують однакового рівня захисту. Більш гнучкий підхід до БФА часто краще задовольняє інтереси і сервісу, і користувачів.

Одним із таких гнучких підходів є дозвіл користувачам вибирати, чи застосовувати БФА, особливо в ситуаціях, коли вони головним чином стурбовані захистом власних даних, а ці дані не є надто чутливими. Цей підхід надає користувачам більше можливостей для прийняття рішень у процесі, оскільки вони можуть самі обрати, чи бажають вони використовувати багатофакторну аутентифікацію, а також можуть обрати найзручніший для себе спосіб з кількох варіантів.

Іноді користувачу потрібен доступ лише до інформації, що потребує мінімального захисту. У таких випадках вхід за допомогою одного фактора, зазвичай пароля, може бути більш доцільним, ніж використання БФА з самого початку. Це знімає з користувача додаткові вимоги для входу, а постачальнику даних не потрібно надмірно захищати менш чутливу інформацію.

Однак, як тільки користувач намагається отримати доступ до більш чутливої інформації або здійснити транзакцію, може бути запрошено додаткові фактори, підвищуючи рівень аутентифікації з одного фактора до двох або більше (рис. 2.2).

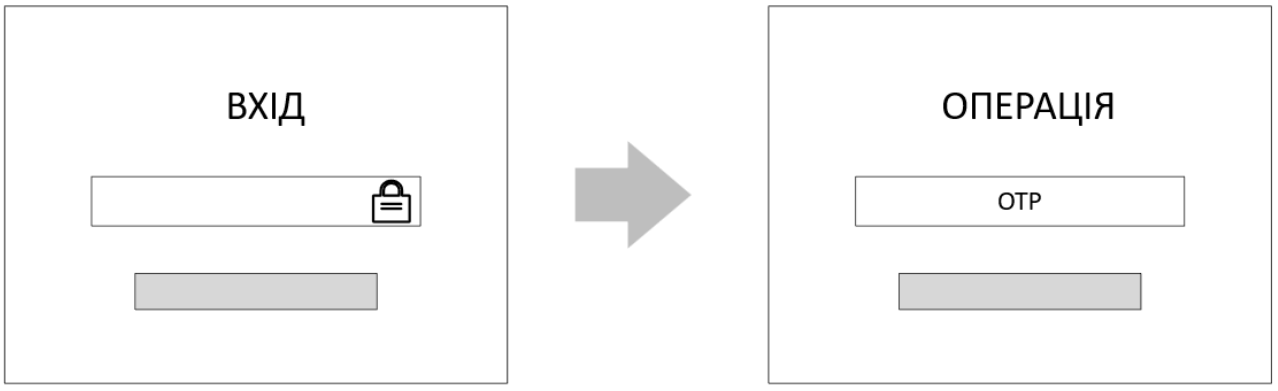

Рисунок 2.2 – Аутентифікація з підвищенням рівня

Такий підхід часто використовується у фінансовій сфері, де, наприклад, клієнти можуть просто бажати переглянути баланс свого рахунку або кредитний рейтинг. Якщо БФА використовувалася б з самого початку, користувачам може бути набридливо заходити на сайт для таких простих запитів. Можливість доступу до базової інформації за допомогою входу з одним фактором покращує досвід користувача без загрози для безпеки.

Водночас, такий підхід до аутентифікації забезпечує те, що у випадку, коли користувачі бажають отримати доступ до більш чутливих даних, наприклад, при бажанні переказати гроші або подати заявку на кредит, їх просять пройти додаткову аутентифікацію. Користувач використовує БФА лише тоді, коли виникає потреба у підвищеній безпеці.

Ще одним підходом до БФА, який може покращити зручність використання, зберігаючи при цьому безпеку, є точне налаштування терміну дії (TTL) різних факторів аутентифікації. У цьому випадку користувач вперше входить у систему з використанням двох або більше факторів, встановлюючи таким чином БФА з самого початку. Однак, якщо користувач продовжує використовувати той самий браузер і довірений пристрій, проходити весь процес знову не потрібно протягом тривалого часу (рис. 2.3).

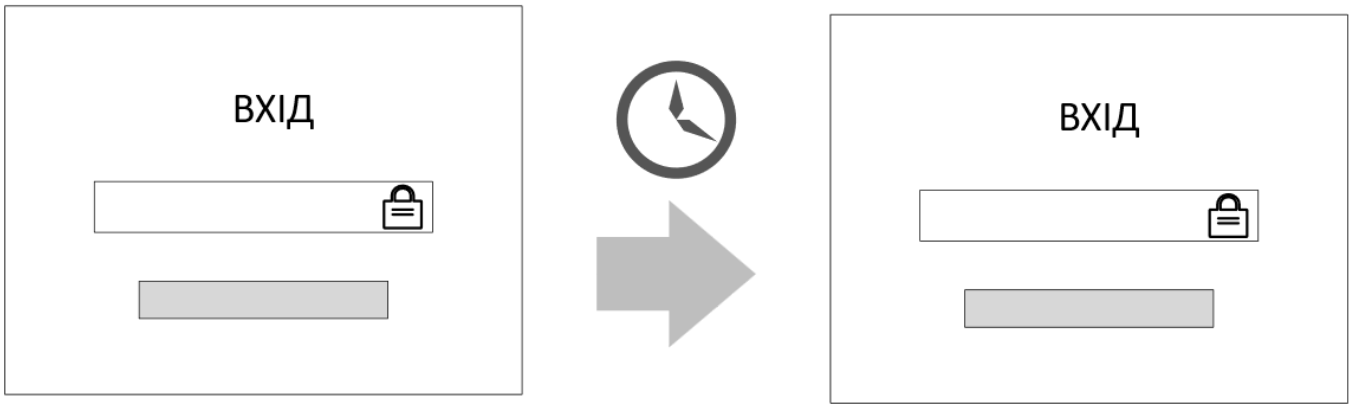

Рисунок 2.3 – Аутентифікація з TTL

Google використовує такий підхід для своїх облікових записів. Якщо користувач вмикає БФА для свого облікового запису, йому спочатку потрібно увійти з використанням двох факторів: пароля і коду, наданого через SMS, електронну пошту або голосовий дзвінок. Після цього, за умови подальшого використання того ж самого браузера та комп'ютера, користувачеві може лише іноді пропонуватися введення пароля, а запити на код доступу трапляються ще рідше.

TTL для двох факторів встановлюються на різні періоди, з коротшим терміном для пароля. Однак, якщо користувач намагається отримати доступ до того ж облікового запису з іншого пристрою або в іншому браузері, обидва фактори аутентифікації знову знадобляться для повторної перевірки.

У даному підході TTL різних факторів може бути налаштоване на будь-яку тривалість, яка необхідні для оптимізації досвіду користувача при збереженні належного рівня безпеки.

БФА може застосовуватися до різних сценаріїв використання і вимагає, щоб система управління ідентифікацією та доступом була розрахована на впровадження різних засобів. Різні підходи до БФА включають компроміси між зручністю використання і безпекою, але кожен із них має свій баланс. Підхід «завжди ввімкнено» ставить на перше місце потреби безпеки, нехтуючи зручністю. Підхід «за бажанням» дозволяє користувачам вирішувати, наскільки їм потрібна безпека, дозволяючи віддавати перевагу безпеці над зручністю, якщо вони бажають. Аутентифікація з підвищенням рівня допускає легкість використання під час початкового входу, переходячи до БФА, коли потрібен вищий рівень безпеки. Перевірка з урахуванням часу, навпаки, використовує БФА з самого початку, але розтягує процес повторної верифікації, щоб знизити навантаження на користувача, не жертвуючи рівнем захисту БФА.

Висновки за розділом 2

У даному розділі було ґрунтовно розглянули ключові аспекти, які відіграють важливу роль у процесі розробки сучасних API, включаючи архітектурні підходи,

вибір мов програмування та фреймворків, а також важливість інтеграції багатофакторної аутентифікації. Починаючи з огляду архітектур API, було встановлено, що API є ланцюгом взаємодії між різними компонентами застосунків, дозволяючи їм обмінюватися інформацією та функціональністю в надійний і ефективний спосіб.

Розглянувши архітектуру REST, було зроблено висновок, що дана архітектурна парадигма домінує у світі веб-розробки завдяки своїй простоті і зрозумілості. REST ґрунтується на підході відсутності стану та використовує HTTP методи для взаємодії із кінцевими точками. Перевагами REST є його широка підтримка, легкість у реалізації та інтеграції, проте він має певні обмеження, зокрема можливість надмірних запитів і обмеження у гнучкості обробки даних.

Webhooks, як частина екосистеми API, виступають важливими інструментами для реалізації асинхронних подій і повідомлень. Вони дозволяють скоротити час реакції системи та зменшити навантаження на сервер, оскільки сервер стає ініціатором взаємодії лише у разі необхідності.

GraphQL представляє собою сучасну альтернативу REST, пропонуючи єдину кінцеву точку для запитів та більшу гнучкість у виборі даних, які необхідні для застосунку. Використання схем для опису даних забезпечує чіткість у структурі API. Попри переваги, такі як зменшення надлишкових даних і кількості запитів, GraphQL може вимагати складнішої конфігурації та додаткової оптимізації для збереження безпеки.

Важливою частиною розробки API є вибір мов програмування, які суттєво впливають на продуктивність та масштабованість рішень. Розглядаючи JavaScript/TypeScript, Python, Java, PHP, C#, і Go, було проаналізовано їх у контексті підтримки паралелізму, продуктивності, швидкості розробки та активності спільноти. Кожна з мов має свої сильні сторони, і правильний вибір залежить від специфічних потреб проєкту.

Окрім мов програмування, було приділено увагу до фреймворків, які використовуються для розробки REST API. Для кожної з мов існують різні фреймворки, кожен з яких має різні особливості та функціонал. Було розглянуто

деякі фреймворки для кожної мов, зокрема їх функціонал, а також сильні і слабкі сторони.

На завершення розділу було розглянуто роль БФА в забезпеченні безпеки сучасних веб-застосунків. Введення різних підходів до аутентифікації, таких як «завжди ввімкнено», «за бажанням», «з підвищенням рівня» та «з урахуванням часу», демонструє різноманітність способів забезпечення безпеки, які можуть бути адаптовані до специфічних потреб користувачів, зберігаючи при цьому баланс між зручністю та безпекою.

Підсумовуючи, всі розглянуті аспекти розробки API – від архітектурних підходів та вибору мов програмування до забезпечення безпеки – є взаємопов'язаними і вимагають комплексного підходу. Ретельно проаналізовані та виважені рішення на етапі проектування і реалізації API здатні забезпечити високоякісне, надійне і масштабоване програмне забезпечення, яке задовольнятиме потреби бізнесу та кінцевих користувачів.

РОЗДІЛ 3
РОЗРОБКА МЕТОДУ ВПРОВАДЖЕННЯ БАГАТОФАКТОРНОЇ АУТЕНТИФІКАЦІЇ У API ТА ЙОГО ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Метод впровадження багатофакторної аутентифікації та вимоги до програмного рішення

Створення рішення передбачає використання технології GraphQL. У межах розробки API з підтримкою багатофакторної аутентифікації ключовими є вимоги, що забезпечують одночасно простоту використання та високу безпеку при роботі з даними. Передусім програма повинна адмініструвати облікові записи користувачів таким чином, щоб під час реєстрації було передбачено процес увімкнення другого фактору. Йдеться про генерацію секрету для використання TOTP, який згодом увесь час повинен бути доступний у користувача, наприклад, через мобільний застосунок чи інший спеціалізований засіб. З технічного погляду це означає, що під час створення акаунта з сервера необхідно надіслати користувачеві унікальний секрет (наприклад, у вигляді base32 чи посилання, що дозволить згенерувати QR-код), а валідацію введених користувачем одноразових кодів слід виконувати під час сесій, пов’язаних із «чутливими» операціями.

Важливим функціональним аспектом проєкту є розмежування доступу до даних: запити на отримання інформації, що не містить чутливої інформації, наприклад, перевірка власного балансу, мають бути доступні вже після базової аутентифікації за допомогою логіна та пароля. Тобто достатньо успішного входу в систему без підтвердження TOTP-кодом. У цьому випадку система вважає, що якщо користувач уже правильно ідентифікував себе на початковому етапі, то простий перегляд персональних даних не містить значного ризику зловживання, який вимагав би обов’язкового захисту другим фактором. Натомість мутації, що можуть змінювати дані, зокрема робити нові записи про операції з балансом чи коригувати наявні, мусять додатково вимагати підтвердження особи через TOTP-код. Це

гарантує, що неавторизована особа, випадково чи зловмисно отримавши логін і пароль, не зможе вчиняти критично важливі дії. Отже, у разі виклику GraphQL-мутації, яка змінює баланс, застосунок повинен запросити користувача ввести одноразовий код, розрахований згідно з алгоритмом TOTP.

У процесі взаємодії з сервером через GraphQL після успішної першої (базової) аутентифікації користувач отримує токен. Коли ж користувач викликає «чутливу» мутацію, сервер повинен виконати функцію, яка й відповідає за валідацію TOTP. У більш складних проєктах може існувати навіть окремий сервіс валідації, що допомагає розподілити навантаження; проте в рамках демонстраційного рішення валідація може відбуватися безпосередньо на сервері, який оброблює GraphQL-запити. Якщо TOTP правильний, внесення змін у баланс дозволяється; якщо ж код невірний або термін його дії закінчився, то операцію слід відхилити. Загальна схема програми представлена на рисунку 3.1.

Таким чином, у вимогах до API зосереджено увагу на двох рівнях аутентифікації: спочатку користувач проходить звичайну процедуру авторизації та отримує базовий доступ, а згодом, за потреби виконувати чутливу операцію, система вимагає підтвердити другий фактор. Усе це спрямовано на забезпечення максимальної зручності для користувача (адже прості перегляди інформації зайвих механізмів майже не потребують) та водночас надійного захисту «вразливих» операцій. Така модель, з одного боку, робить систему стійкою перед можливою компрометацією логіна й пароля, а з іншого – дає змогу впроваджувати її у проєкти, де передбачені подальше розширення, як-от робота з додатковими користувацькими даними чи використання інших факторів аутентифікації (SMS, пуш-сповіщення тощо).

Розглянемо детальніше наступні елементи API:

- Реєстрація.
- Вхід до системи.
- Операції.

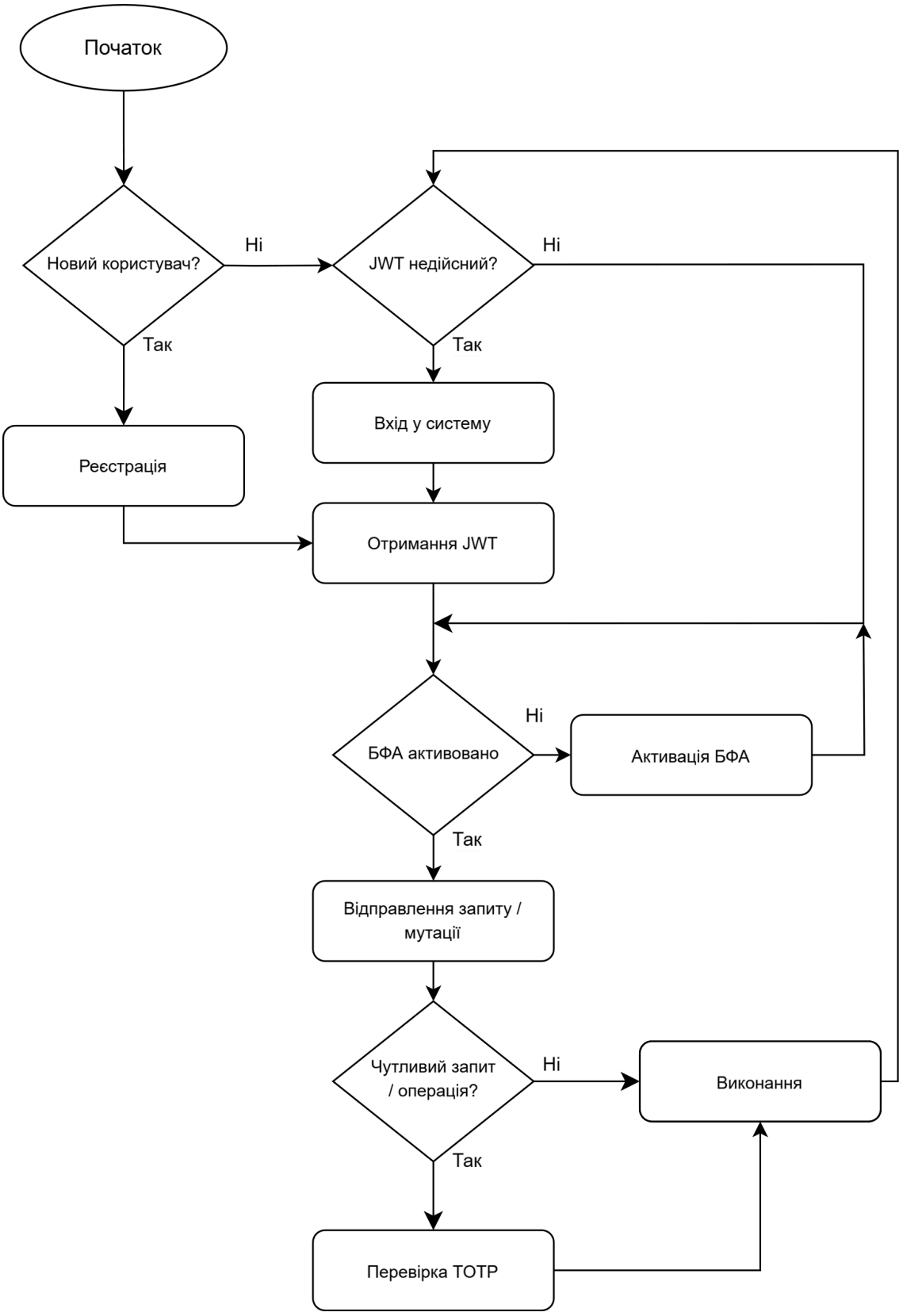

Рисунок 3.1 – Узагальнена схема програмного рішення

Вхід у систему відбувається шляхом відправлення мутації, яка містить зміні із електронною поштою та паролем. Мутацію авторизації можна математично

представити загальною формулою 3.1. Ця функція приймає впорядковану пару (email, пароль) з декартового добутку E×P та відображає її на елемент із множини T, тобто JWT токен, або помилки, у разі її виникнення.

$$(E \times P) \rightarrow (T \cup \{Error\}) \quad (3.1)$$

де E – множина усіх можливих email (рядки);

P – множина усіх можливих паролів (рядки);

T – множина всіх можливих JWT-токенів;

Error – множина можливих помилок.

Для виконання зі сторони сервера виконуються функції пошуку користувача (3.2), функція перевірки відповідності пароля і значення хешу (3.3) та функція генерації JWT (3.4).

$$E \rightarrow U \cup \{Error\} \quad (3.2)$$

де E – множина усіх можливих email (рядки);

U – множина користувачів;

Error – множина можливих помилок.

$$P \times H \rightarrow \{true, false\} \quad (3.3)$$

де P – множина усіх можливих паролів (рядки);

H – множина можливих значень хешів.

$$U \rightarrow T \quad (3.4)$$

де U – множина користувачів;

T – множина всіх можливих JWT-токенів.

Процес авторизації обробляє вхідні дані, перевіряє їхню достовірність з базою даних і, у разі успішної автентифікації, генерує унікальний JWT токен, який містить

закодовану інформацію про користувача та має обмежений термін дії. Цей токен потім використовується для подальших авторизованих запитів до системи. (рис. 3.2).

Рисунок 3.2 – Схема мутації для входу у систему

Процес реєстрації передбачає декілька кроків, а саме: відправлення запиту на реєстрацію, власне реєстрацію та активацію БФА (рис. 3.3).

Перша мутація (формула 3.5) приймає значення email (e) з множини всіх можливих email (E) та відображає його на елемент множини всіх можливих реєстраційних токенів (R) або на елемент «Error» у випадку невдачі. Процес виконання цієї мутації включає генерацію нового реєстраційного токена за допомогою функції яка детерміністично або псевдовипадково створює токен, після чого отриманий токен зберігається у відповідність до вказаного email через функцію, яка представлена формулою 3.6.

$$E \to R \cup \{Error\} \tag{3.5}$$

де E – множина усіх можливих email (рядки);

R – множина усіх можливих реєстраційних токенів;

Error – множина можливих помилок.

$$E \times R \to \{ok, Error\} \tag{3.6}$$

де E – множина усіх можливих email (рядки);

R – множина усіх можливих реєстраційних токенів;

Error – множина можливих помилок.

Сервер генерує токен, робить відповідний запис у базі даних та повертає клієнту згенерований токен, який потім необхідно надати для продовження процесу реєстрації.

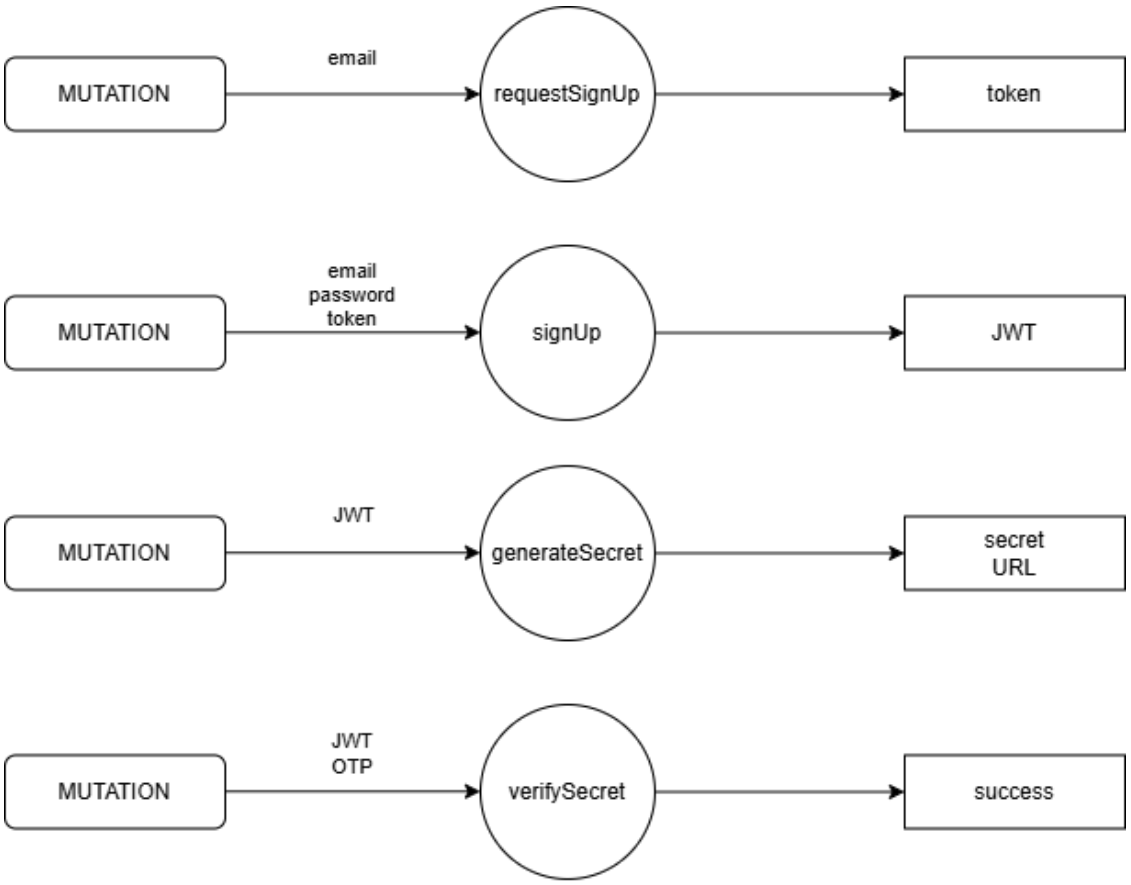

Рисунок 3.3 – Схема мутацій для роботи з БФА

Друга мутація (формула 3.7) є композицією кількох операцій: вона приймає три значення (email, реєстраційний токен, пароль) та у разі успіху повертає JWT-токен з множини (T) або «Error» у випадку проблем.

$$E \times R \times P \to T \cup \{Error\} \tag{3.7}$$

де E – множина усіх можливих email (рядки);

R – множина усіх можливих реєстраційних токенів;

P – множина усіх можливих паролів;

T – множина всіх можливих JWT-токенів.

Error – множина можливих помилок.

Виконання мутації реєстрації передбачає перевірку валідності реєстраційного токена для вказаного email через функцію, описану формулою 3.8.

$$E \times R \to \{true,\ false\} \tag{3.8}$$

де E – множина усіх можливих email (рядки);

R – множина усіх можливих реєстраційних токенів.

Якщо токен дійсний, система хешує пароль за допомогою відповідного алгоритму (формула 3.9), створює нового користувача, зберігаючи email та значення хешу пароля (формула 3.10), і нарешті генерує JWT-токен, як і випадку безпосереднього входу у систему, який і повертається як результат успішного виконання мутації (формула 3.4).

$$P \to H \tag{3.9}$$

де P – множина усіх можливих паролів;

H – множина усіх можливих значень хешів.

$$E \times H \to U \tag{3.10}$$

де E – множина усіх можливих email;

H – множина усіх можливих значень хешів;

U – множина користувачів.

Для подальшої роботи із API токен має бути присутнім у кожному наступному запиті чи мутації. Передача токену виконується за рахунок заголовку «Authorization» із приписом «Bearer».

Далі користувачу необхідно налаштувати БФА. У системі визначено дві основні мутації, що забезпечують даний функціонал: генерацію секрету TOTP та підтвердження активації TOTP.

Перша мутація, що описана формулою 3.11, відображає JWT-токен з множини всіх дійсних токенів T на елемент множини всіх можливих секретів S або на елемент «Error» у випадку невдачі.

$$T \to S \cup \{Error\} \quad (3.11)$$

де T – множина всіх можливих JWT-токенів;

S – множина усіх можливих секретів для TOTP;

Error – множина можливих помилок.

Виконання цієї мутації починається з валідації переданого JWT-токена за допомогою відповідної функції (формула 3.12), яка повертає користувача з множини зареєстрованих користувачів U або помилку, якщо токен недійсний.

$$T \to U \cup \{Error\} \quad (3.12)$$

де T – множина всіх можливих JWT-токенів;

U – множина користувачів;

Error – множина можливих помилок.

Якщо токен валідний і користувач ідентифікований, система генерує новий секрет для TOTP, після чого зберігає згенерований секрет у профілі користувача (формула 3.13). У випадку успішного завершення всіх операцій, мутація повертає згенерований секрет, інакше – помилку.

$$U \times S \to \{ok\ Error\} \quad (3.13)$$

де U – множина користувачів;

S – множина усіх можливих секретів для TOTP;

Error – множина можливих помилок.

Друга мутація приймає пару значень (JWT-токен, одноразовий пароль) з декартового добутку множин T × O та відображає її на повідомлення про успіх або на помилку (формула 3.14).

$$T \times O \to \{true,\ Error\} \quad (3.14)$$

де’ T – множина всіх можливих JWT-токенів;

O – множина усіх одноразових паролів;

Error – множина можливих помилок.

Процес підтвердження активації починається з перевірки токена та отримання відповідного користувача, потім система отримує раніше збережений секрет цього користувача через функцію, яка представлена формулою 3.15 і перевіряє коректність переданого одноразового пароля (формула 3.16).

$$U \rightarrow S \cup \{Error\} \tag{3.15}$$

де U – множина користувачів;

S – множина усіх можливих секретів для TOTP;

Error – множина можливих помилок.

$$S \times O \rightarrow \{true, false\} \tag{3.16}$$

де S – множина усіх можливих секретів для TOTP;

O – множина усіх одноразових паролів.

Якщо одноразовий пароль підтверджено, система активує двофакторну автентифікацію для користувача та повертає оновлений JWT-токен, який містить інформацію про активований TOTP; у випадку будь-якої помилки мутація повертає відповідне повідомлення про помилку. Відправлення TOTP відбувається за рахунок використання заголовку «X-2FA».

Після реєстрації чи входу у систему відкривається можливість здійснювати операції. Для демонстрації API містить один запит для виконання якого достатньо бути лише авторизованим у системі (мати JWT) та одну мутацію, виконати яку можна лише у випадку надання правильного TOTP (формула 3.17, 3.18).

$$T \rightarrow (ID \times E \times B) \cup \{Error\} \tag{3.17}$$

де T – множина можливих JWT-токенів;

ID – множина значень id;

E – множина значень email;

B – множина значень балансу;

Error – множина можливих помилок.

$$T \times O \rightarrow (B \times NB) \cup \{Erro$$ (3.18s)

де T – множина можливих JWT-токенів;

O – множина усіх одноразових паролів;

B – множина значень балансу;

NB – множина нових значень балансу;

Error – множина можливих помилок.

Запит «userInfo» дозволяє переглядати інформацію про користувача, зокрема його ID, email та баланс. Мутація «changeBalance», у свою чергу, дозволяє користувачу змінити значення балансу (рис. 3.4).

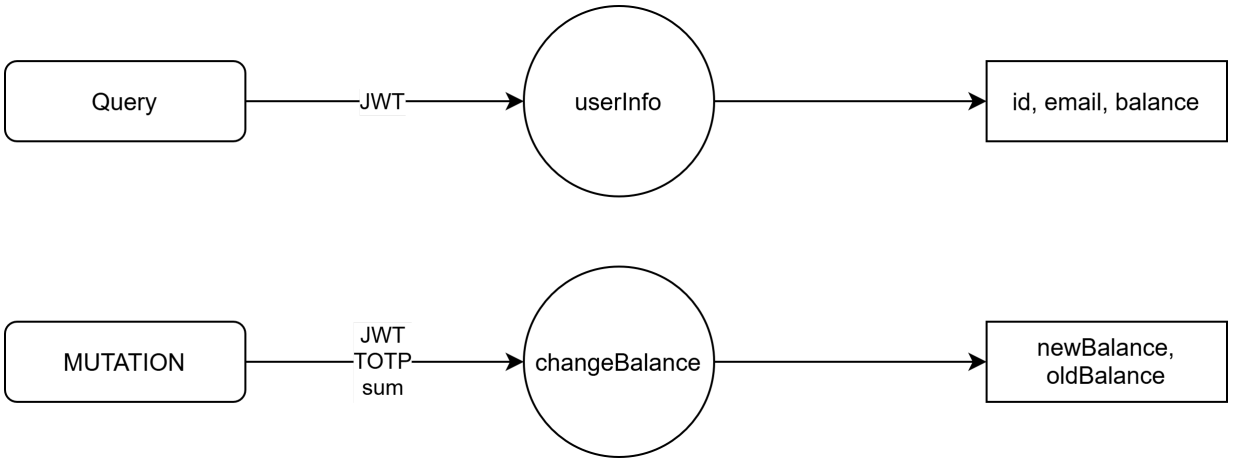

Рисунок 3.4 – Схема запитів та мутацій, пов’язаних із користувачем

Система має клієнт-серверну архітектуру та складається із трьох компонентів: клієнт, сервер та база даних (рис. 3.5). Клієнт ініціює комунікацію із сервером та відповідає за відправлення запитів. Сервер оброблює запити, отримані від клієнта, та повертає відповіді.

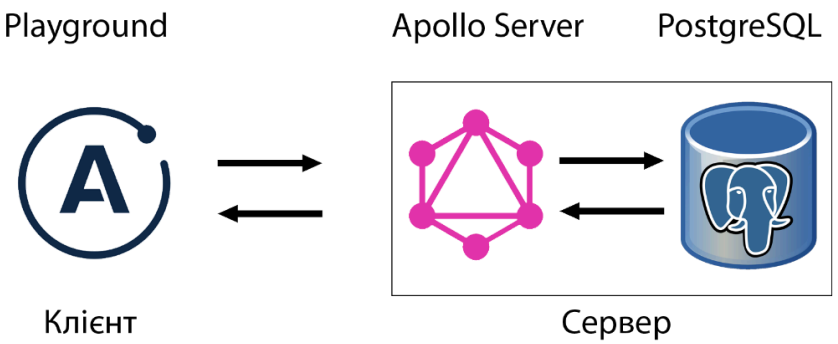

Рисунок 3.5 – Схема системи та інструментів

Клієнт ініціює комунікацію із сервером та відповідає за відправлення запитів. Сервер оброблює запити, отримані від клієнта, та повертає відповіді. Обробка запитів виконується за рахунок ресурсів сервер та комунікації із базою даних. База даних містить інформацію про користувача та токени для реєстрації. Вміст відповідних таблиць представлено на рисунках 3.6 та 3.7.

Users			
id	text	PK	Первинний ключ
email	text	UQ	Унікальний email користувача
passwordHash	text		Хеш-значення пароля
secret	text		OTP-секрет
secret_verified	boolean		Чи підтверджено секрет
balance	numeric		Поточний баланс
createdAt	timestamp		Дата та час створення
updatedAt	timestamp		Дата та час модифікації

Рисунок 3.6 – Вміст таблиці «users»

Таблиця «users» містить інформацію про користувачів, а саме їх ID, email, хеш-значення пароля, інформацію про БФА, значення балансу та технічну інформацію про дату створення та модифікації запису.

signUpRequests			
id	text	PK	Первинний ключ
email	text	UQ	Унікальний ключ запиту (email)
tokenHash	text		Хеш-значення токену
redeemedAt	timestamp		Дата та час використання
expiresAt	timestamp		Дата та час закінчення життя
createdAt	timestamp		Дата та час створення
updatedAt	timestamp		Дата та час модифікації

Рисунок 3.7 – Вміст таблиці «signUpRequests»

Таблиця «signUpRequests» зберігає інформацію про запити на реєстрацію, які створюються, коли користувач починає процес створення облікового запису.

3.2 Інструменти для розробки програмного рішення

У попередньому розділі було концептуально описано API рішення, зокрема його архітектуру, запити та структуру, але для практичної реалізації необхідно обрати відповідні інструменти, які допоможуть у створенні програми. Як вже згадувалося раніше, для роботи необхідно три складові частини, а саме: клієнт, сервер та база даних. Розглянемо детальніше рішення, які будуть використовуватися.

Для розробки програмного рішення було обрано технологію GraphQl, враховуючи її відносну новизну та активне поширення. Однією з переваг GraphQl є можливість використовувати її у якості обгортки для звичних REST API [84]. Таким чином, існує можливість розробки рішення, яке буде універсальним, не залежно від джерела даних

Клієнт у контексті API - це певне програмне забезпечення, яке відправляє запити та очікує відповіді. API мають працювати однаково з усіма клієнтами, які підтримують відповідну технологію, тому для демонстрації роботи можна використати будь-який інструмент, який дозволить взаємодіяти із API.

Сервер GraphQl може бути реалізовано великою кількість інструментів на різних мовах програмування, хоча вперше GraphQL було представлено на конференції React, і GraphQL.js є однією з найбільш широко використовуваних реалізацій на сьогоднішній день [84].

Що стосується джерела даних, то, як вже було згадано, GraphQL дозволяє використовувати, як бази даних, так і інші API. Бази даних можуть бути, як SQL, так і noSQL.

Таким чином, для розробки програмного рішення у даній роботі було обрано наступні інструменти:

- Мова програмування - TypeScript.
- Платформа - Node.js.
- Клієнт - GraphQL Playground.
- Сервер - Apollo Server.
- База даних - PostgreSQL.

Розглянемо ці технології детальніше. Node.js є однією з платформ для побудови сучасних вебзастосунків, зокрема завдяки можливості ефективного опрацювання великої кількості одночасних з’єднань. З технічного погляду вона являє собою середовище виконання JavaScript на боці сервера, базоване на рушії V8, розробленому компанією Google [85]. За своєю суттю Node.js працює в одному потоці, але водночас дозволяє асинхронне оброблення операцій, що істотно полегшує обмін даними з клієнтами та дає змогу масштабувати застосунки без необхідності створювати складні синхронізовані механізми [85]. Для GraphQL API, у якому запити можуть надходити від багатьох користувачів одночасно, це особливо корисно, оскільки так можна забезпечити ефективне транспортування даних, тобто швидко обробляти як запити на отримання інформації, так і “важчі” мутації, не блокуючи основний потік.

Іншою перевагою Node.js є величезна екосистема пакетів, доступних через менеджер npm, яку можна використовувати для швидкого підключення та налаштування будь-яких специфічних інструментів.

Загалом використання Node.js у контексті архітектури під багатофакторну аутентифікацію пов’язане з декількома чинниками. По-перше, опрацювання аутентифікації та додаткового фактора, такого як TOTP, передбачає низку асинхронних викликів: читання та збереження даних користувача, генерування і валідація секретів тощо. Node.js дає змогу мінімізувати час очікування під час виконання таких процедур. По-друге, оскільки розробка пов’язана зі створенням GraphQL API, JavaScript на рівні сервера сприяє швидкій інтеграції з фронтенд-складовою: і за допомогою відповідних бібліотек, і завдяки використанню знайомої розробникам мови програмування. По-третє, Node.js чудово масштабуються: при зростанні кількості користувачів можна просто підвищити кількість паралельно запущених процесів через вбудований механізм cluster або використовувати зовнішню інфраструктуру контейнеризації та оркестрації.

TypeScript є надбудовою над JavaScript, яка забезпечує статичну типізацію та дозволяє виявляти логічні помилки ще на етапі розробки, перш ніж код фактично запускається у виробничому середовищі [86]. При створенні великих та складних застосунків, особливо таких, де важлива архітектурна й логічна цілісність, статична типізація стає надзвичайно корисною. Зокрема, використання TypeScript допомагає чіткіше описувати структуру об'єктів, валідувати вхідні дані для функцій і методів, а також підтримує кращу навігацію у вихідному коді завдяки підказкам і підсвічуванню синтаксису у сучасних середовищах розробки.

У контексті розробки GraphQL API з багатофакторною аутентифікацією TypeScript дозволяє встановити строгі типи для різних сутностей: наприклад, для “моделі” користувача можна визначити, які саме поля містить об’єкт (логін, пароль, секрет для TOTP тощо), а також деталізувати, у яких форматах ці поля передаються. Крім того, завдяки статичній типізації можна забезпечити суворіший контроль даних при взаємодії між бекендом і фронтендом або між окремими частинами серверного коду. Це особливо актуально для GraphQL, де передбачений власний механізм опису типів, адже TypeScript може знизити ризик розсинхронізації схеми.

Ще однією перевагою TypeScript є те, що, попри необхідність додаткового етапу трансляції до JavaScript, фактично немає проблем із сумісністю, адже

підсумковий код представлений на JavaScript. Для проєктів на Node.js ця компіляція не створює перешкод: один раз налаштувавши конфігурацію (наприклад, tsconfig.json), можна запустити процес збірки, що автоматично сформує файл або набір файлів, призначених для виконання у середовищі Node.js. До того ж, завдяки IDE та іншим інструментам стає можливо відслідковувати помилки на етапі написання коду.

Коли ж ідеться про проєкт із багатофакторною аутентифікацією, дуже важливо тримати під контролем усі сутності, які описують користувача, його права та роль в системі, механізм генерації й підтвердження TOTP. TypeScript убезпечує від багатьох потенційних багів, коли, наприклад, деякому функціоналу передається невірний формат даних або пропущено обов’язковий параметр. А оскільки для процедури валідації другого фактора доведеться працювати з множиною різних полів (секрет, одноразовий код, проміжки часу валідації тощо), точні та суворі визначення типів дають змогу уникнути нечітких або двозначних ситуацій у коді й легше відстежувати, де саме відбувається обробка кожного параметра.

Apollo Server дає змогу швидко та структуровано підняти GraphQL-сервер у середовищі Node.js, забезпечуючи зручний механізм визначення схем та резолверів. З його допомогою можна конфігурувати, як різні оператори (запити, мутації, підписки) будуть взаємодіяти з бізнес-логікою та даними, без потреби ручного налаштування або обробки HTTP-запитів. Apollo Server «приховує» більшість рутинних кроків, завдяки чому розробник безпосередньо зосереджується на описі GraphQL-схеми, логіці аутентифікації й авторизації та на підключенні до бази даних чи інших сервісів [87].

Особливо важливо, що підтримується гнучке налаштування контексту для кожного запиту, де можна верифікувати користувача, аналізувати його TOTP-код і відразу в рамках того ж контексту передавати відповідні об’єкти резольверам. Це засадничо спрощує реалізацію багатофакторної аутентифікації, адже немає потреби щоразу самостійно писати додаткові рішення. Будь-який “чутливий” метод можна забезпечити перевірками на рівні доступу (role-based або privilege-based), якщо

втручається логіка другого фактору – там же, у функції, що обробляє мутацію, перевіряється чинний TOTP-код.

Натомість GraphQL Playground, який часто інтегрують безпосередньо в Apollo Server або запускають окремо, демонструє зручну інтерактивну консоль для тестування запитів і мутацій. З нього можна в реальному часі звертатися до сервера, змінювати структуру запитів, досліджувати, які поля доступні, і бачити результат виконання. Крім того, Playground автоматично формує документацію за схемою, що суттєво полегшує розуміння зв’язків між сутностями, вхідними та вихідними даними. У контексті проєкту, де потрібно реалізувати багатофакторну аутентифікацію, Playground стає надзвичайно швидким інструментом перевірки, чи правильно працює валідатор другого фактору, чи вчасно надсилаються помилки та повідомлення про невдалі спроби, і як саме відповідає сервер, коли TOTP-код відсутній або некоректний. Крім того, під час розробки можна оперативно експериментувати з різними схемами та налаштуваннями доступу, виявляти несподівані взаємодії між елементами схеми, щоб ужити заходів для уникнення потенційних вразливостей.

Особливість Apollo Server полягає в інтеграції з екосистемою Apollo, що включає низку корисних інструментів. Одним із таких інструментів є Apollo Playground, який забезпечує зручний веб-інтерфейс для тестування GraphQL-запитів. При запуску сервера розробник може відкрити Playground у браузері й безпосередньо відправляти запити чи мутації, вказуючи потрібні параметри. Це дуже пришвидшує процес налагодження, адже можна відразу оцінити, чи правильно побудовано схему, чи коректні назви типів і полів, та чи повертаються очікувані результати. Крім того, Playground має можливість автоматично документувати запити та мутації безпосередньо на основі оголошеної GraphQL-схеми. Тобто розробник або тестувальник може швидко переглянути структуру схем, опис та доступні параметри, а також побачити приклади побудови запиту.

Ще одна перевага полягає в тому, що Playground динамічно відстежує зміни в схемі, якщо сервер перезапускається з оновленими параметрами, й одразу дає змогу працювати з оновленим набором полів, типів і мутацій. У контексті рішення, що

розробляється, це дозволяє поетапно тестувати процеси реєстрації, перегляду балансу чи зміни цього балансу під час різних фаз аутентифікації й оперативно перевіряти, чи коректно виконується логіка перевірки коду TOTP.

PostgreSQL є однією з найпопулярніших реляційних систем управління базами даних із відкритим кодом, яка виділяється надійністю, широкими можливостями масштабування та активною спільнотою розробників [88]. У контексті створення GraphQL API з підтримкою багатофакторної аутентифікації PostgreSQL слугує сховищем користувацьких та інших операційних даних, що дозволяє системі бути цілісною з погляду безпеки та керованості даних. Для конфігурації користувачів, які застосовують TOTP, таблична модель дає змогу зберігати інформацію про користувача, унікальний секрет, потрібний для генерування одноразових кодів, а також усі відповідні налаштування доступу і параметри аутентифікації.

У рамках багатофакторної аутентифікації часто виникає потреба тримати записи, які описують стан та історію авторизації: коли користувачі востаннє успішно виконували вхід, коли було активовано чи дезактивовано другий фактор, наскільки часто виникали помилкові коди тощо. PostgreSQL у такому разі дає змогу зручно і надійно зберігати, обробляти й аналізувати подібні дані, використовуючи не лише стандартні реляційні інструменти, а й розширені можливості, як-от вбудовані функції для аналітики й створення матеріалізованих уявлень. Це, до речі, стане у пригоді, якщо знадобиться відстежувати спроби зловживань або виконувати додаткові перевірки безпеки.

3.3 Огляд коду програмного рішення

Розглянемо основні елементи програмного коду API, яке було реалізовано на платформі Node.js. Основним файлом програми є «index.ts» (рис. 3.8).

Спочатку імпортуються необхідні модулі, включаючи «ApolloServer» та «startStandaloneServer» з відповідних пакетів. Далі код імпортує резолвери (authResolvers, userResolvers, mfaResolvers) та типи (typeDefs) з локального каталогу

«./gql/index.js», а також функцію «verifyJWT» з модуля аутентифікації та «GraphQLError» для обробки помилок.

src > TS index.ts > ...

```ts
import { ApolloServer } from "@apollo/server";
import { startStandaloneServer } from "@apollo/server/standalone"

// resolvers and types
import { authResolvers, mfaResolvers, typeDefs, userResolvers } from "./gql/index.js";
import { verifyJWT } from "modules/auth/auth.js";
import { GraphQLError } from "graphql";

const resolvers = [authResolvers, userResolvers, mfaResolvers];

// server setup
const server = new ApolloServer({
    typeDefs,
    resolvers,
})

//@ts-ignore
const { url } = await startStandaloneServer(server, {
    context: async ({ req }) => {
        const authToken = req.headers.authorization?.split(' ')[1];
        const sfToken = req.headersDistinct["x-2fa"]?.at(0);
        if (!authToken) return {};
        try {
            const user = await verifyJWT(authToken);
            return {user, sfToken};
        } catch (e) {
            throw new GraphQLError("Invalid token", {extensions: {code: 'NOT_AUTHENTICATED'}})
        };
    }
})

console.log("Server ready at port", url)
```

Рисунок 3.8 – Вміст файлу index.ts

Всі резолвери об'єднуються в один масив. Потім створюється новий екземпляр сервера «Apollo» з вказаними типами даних та резолверами. Запуск сервера відбувається за допомогою функції «startStandaloneServer», де також налаштовується контекст запитів. У контексті відбувається обробка авторизаційного токена (JWT) та токена БФА. Для цього з заголовків запиту витягуються відповідні дані, а токен JWT верифікується. Якщо токен валідний, до контексту додаються дані користувача та одноразовий пароль, що дозволяє використовувати ці дані в резолверах. У випадку

неправильного токена викидається помилка «NOT_AUTHENTICATED». Після успішного запуску сервера в консоль виводиться повідомлення з URL сервера.

У файлі «auth.ts» виконується опис резолверів аутентифікації. Спочатку імпортуються необхідні залежності: «GraphQLError» для обробки помилок, база даних та функція генерації унікальних ідентифікаторів, «bcrypt» для хешування паролів, «nanoid» для створення випадкових токенів, «DateTime» з бібліотеки «luxon» для роботи з датами та функція «generateJWT» для створення JWT-токенів. Константа «SALT_ROUNDS» встановлює рівень захисту для хешування паролів. Експортований об'єкт «authResolvers» містить три мутації (рис. 3.9, 3.10, 3.11).

```
11  export const authResolvers = {
12      Mutation: {
13          requestSignUp: async (_, {email}) => {
14              await db.signUpRequest.deleteMany({where: {email}});
15
16              const token = nanoid(36);
17              await db.signUpRequest.create({
18                  data: {
19                      id: genId(),
20                      email,
21                      tokenHash: await bcrypt.hash(token, SALT_ROUNDS),
22                      expiresAt: DateTime.now().plus({minutes: 10}).toJSDate(),
23
24                  }
25              });
26
27              // Send token
28              console.log("TOKEN:", token)
29              return {success: true}
30          },
```

Рисунок 3.9 – Код мутації «requestSignUp»

Перша мутація «requestSignUp» обробляє запит на реєстрацію: видаляє попередні запити для вказаної електронної пошти, генерує унікальний токен, зберігає новий запит у базі даних з хешованим токеном та терміном дії 10 хвилин, а потім виводить токен у консоль.

Друга мутація «signUp» завершує процес реєстрації: перевіряє існування та термін дії запиту на реєстрацію, валідує токен, перевіряє, чи не існує вже користувач

з такою поштою, створює нового користувача з хешованим паролем та позначає запит на реєстрацію як використаний, після чого генерує та повертає JWT-токен.

Третя мутація «signIn» відповідає за вхід користувача: знаходить користувача за електронною поштою, перевіряє правильність пароля та, в разі успіху, генерує й повертає JWT-токен для аутентифікації

```
32      signUp: async (_, {token, email, password}) => {
33          const request =await db.signUpRequest.findUnique({where: {email}});
34          if (!request) throw new GraphQLError("Sign up request not found");
35          if (request.expiresAt < new Date()) throw new GraphQLError("Sign up request expired")
36          // compare token with token hash
37          const validToken = await bcrypt.compare(token, request.tokenHash);
38          if (!validToken) throw new GraphQLError("Invalid token");
39
40          // check if user already exists
41          const existingUser = await db.user.findUnique({where: {email}})
42          if (existingUser) throw new GraphQLError("User already exists")
43
44
45
46          // create user and redeem token
47          const [user] = await db.$transaction([
48              db.user.create({
49                  data: {
50                      id: genId(),
51                      email,
52                      secret: null,
53                      secret_verified: null,
54                      passwordHash: await bcrypt.hash(password, SALT_ROUNDS),
55                  }
56              }),
57              db.signUpRequest.update({
58                  where: {id: request.id},
59                  data: {redeemedAt: new Date()},
60              }),
61
62          ])
63
64
65          // generate auth token
66          const authToken = generateJWT(user.id);
67          console.log(authToken);
68          return { authToken };
69      },
```

Рисунок 3.10 – Код мутації «signUp»

```
71      signIn: async (_, {email, password}) => {
72          // check if user exists
73          const user = await db.user.findUnique({where: {email}});
74          if (!user) throw new GraphQLError("Invalid email and/or password");
75
76          // check pswd
77          const correctPassword = await bcrypt.compare(password, user.passwordHash);
78          if (!correctPassword) throw new GraphQLError("Invalid email and/or password");
79
80          // gen token
81          const authToken = generateJWT(user.id);
82          console.log(authToken)
83
84          return { authToken };
85      }
```

Рисунок 3.11 – Код мутації «signIn»

У файлі «mfa.ts» описуються мутації, пов’язані із БФА (рис. 3.12, 3.13).

```
7       generateSecret: async (_, __, {user}) => {
8           // check if user exists
9           const userId = user?.id;
10          console.log(userId);
11          if (!userId) throw new GraphQLError("User not found");
12
13          const existingUser = await db.user.findUniqueOrThrow({where: {id: userId}});
14          if (!existingUser) throw new GraphQLError("User not found");
15
16          // check if secret is already enabled
17          if (existingUser.secret_verified) throw new GraphQLError("Secret is already verified");
18
19          // generate secret
20          const secret = generateSecret();
21
22          // save secret in db
23          await db.user.update({
24              where: {id: userId},
25              data: {
26                  secret: secret.base32,
27                  secret_verified: false
28              }
29          });
30
31          return {secret: secret.base32, url: secret.otpauth_url};
32      },
```

Рисунок 3.12 – Код мутації «generateSecret»

Перша мутація «generateSecret» дозволяє користувачеві створити секретний ключ для БФА: перевіряє існування користувача з контексту запиту, переконується, що секретний ключ ще не верифікований, генерує новий секретний ключ за

допомогою функції «generateSecret», зберігає згенерований ключ у базі даних у форматі «base32» з позначкою, що він ще не верифікований, і повертає клієнту ключ та URL для створення QR-коду в додатку аутентифікації.

Друга мутація «verifySecret» використовується для підтвердження, що користувач правильно налаштував двофакторну аутентифікацію: перевіряє наявність користувача та згенерованого секретного ключа, перевіряє, що ключ ще не верифікований, валідує одноразовий код доступу (OTP), переданий через «sfToken», за допомогою функції «verifySecret», і в разі успіху позначає секретний ключ як верифікований у базі даних.

```
34      verifySecret: async (_, __, {user, sfToken}) => {
35
36        // check user id
37        const userId = user?.id;
38        if (!userId) throw new GraphQLError("User not found");
39
40        // check if secret is already generated
41        if (!user.secret) throw new GraphQLError("Please generate secret first");
42
43        // check if already verified
44        if (user.secret_verified) throw new GraphQLError("Already verified");
45
46        const isValid = await verifySecret(userId, sfToken);
47        if (!isValid) throw new GraphQLError("Wrong OTP");
48
49        await db.user.update({
50            where: {id: userId},
51            data: {
52                secret_verified: true
53            }
54        })
55
56        return {success: isValid};
57    }
```

Рисунок 3.13 – Код мутації «verifySecret»

Цей код реалізує процес налаштування БФА, забезпечуючи додатковий рівень безпеки для користувачів системи, відповідно до визначеної архітектури.

У файлі «user.ts» визначається резолвер для роботи з інформацією користувача у API. Він містить як запит, так і мутацію (рис. 3.14). Запит «userInfo» просто

повертає інформацію про поточного аутентифікованого користувача з контексту, що дозволяє клієнту отримати дані профілю. Мутація «changeBalance» забезпечує зміну балансу користувача з додатковим рівнем захисту: спочатку перевіряється валідність одноразового пароля (OTP) за допомогою функції «validateOTP», використовуючи дані користувача та одноразовий пароль з контексту.

Якщо перевірка успішна, відбувається оновлення балансу користувача в базі даних до вказаної суми після чого повертається об'єкт, що містить як новий, так і старий баланс.

```
5   export const userResolvers = {
6     Query: {
7       userInfo: (_, __, {user}) => {
8         return user;
9       }
10    },
11
12    Mutation: {
13      changeBalance: async (_, {sum}, {user, sfToken}) => {
14        const isValid = await validateOTP(user, sfToken);
15        if (!isValid) throw new GraphQLError("OTP is not valid");
16
17        await db.user.update({
18          where: {id: user.id},
19          data: {balance: sum},
20        });
21
22        return {newBalance: sum, oldBalance: user.balance};
23      }
24    }
25  };
```

Рисунок 3.14 – Код резолвера «userResolver»

Цей підхід гарантує, що операція захищена додатковим шаром безпеки через додатковий фактор. Повний код програми, де представлені інші елементи програми наведено у Додатку А.

3.4 Демонстрація роботи створеного рішення

Розглянемо, як працює створене рішення на практиці. Першочергово користувачу необхідно створити обліковий запис у системі. Процес реєстрації складається з кількох етапів: запит на реєстрацію та безпосередня реєстрація. Для використання мутації, яка відповідає за створення облікового запису, клієнт має надати значення реєстраційного токена, який отримується за допомогою іншої мутації – «requestSignUp», функцією якого є генерація токена, що надається клієнту у певний спосіб, наприклад у посиланні, відправленому на вказану електронну пошту. При успішному відпрацюванні сервер повертає значення «success: true» у відповідь (рис. 3.15).

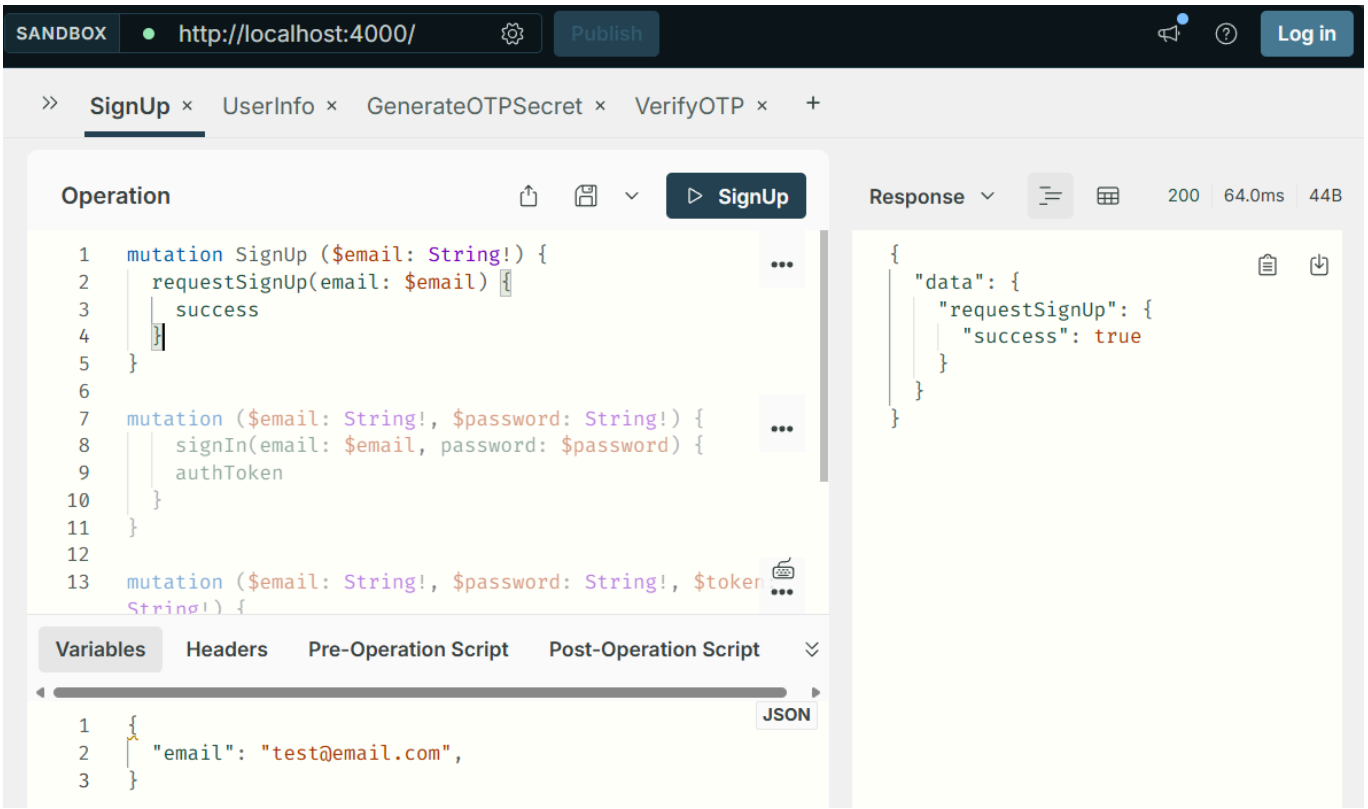

Рисунок 3.15 – Виконання мутації «requestSignUp»

Для демонстрації значення токена повертається у консоль виконуваної програми, проте, як вже згадувалося, метод передачі може бути будь-яким (рис. 3.16).

```
Server ready at port http://localhost:4000/
prisma:info Starting a postgresql pool with 7 connections.
prisma:query DELETE FROM "public"."signUpRequests" WHERE "public
"."signUpRequests"."email" = $1
TOKEN: 3QFwwlPF0V2J0k7rBt67fCtGn5gmpzPgrwp-
prisma:query INSERT INTO "public"."signUpRequests" ("id","email"
```

Рисунок 3.16 – Згенерований токен

Отримавши токен можна перейти до безпосередньої реєстрації. Для цього у змінних мутації «signUp» передаються значення пароля, електронної пошти та самого токена. Якщо під час створення користувача не виникає ніяких помилок, сервер генерує та повертає JWT, який використовується для авторизації під час наступних операцій (рис. 3.17).

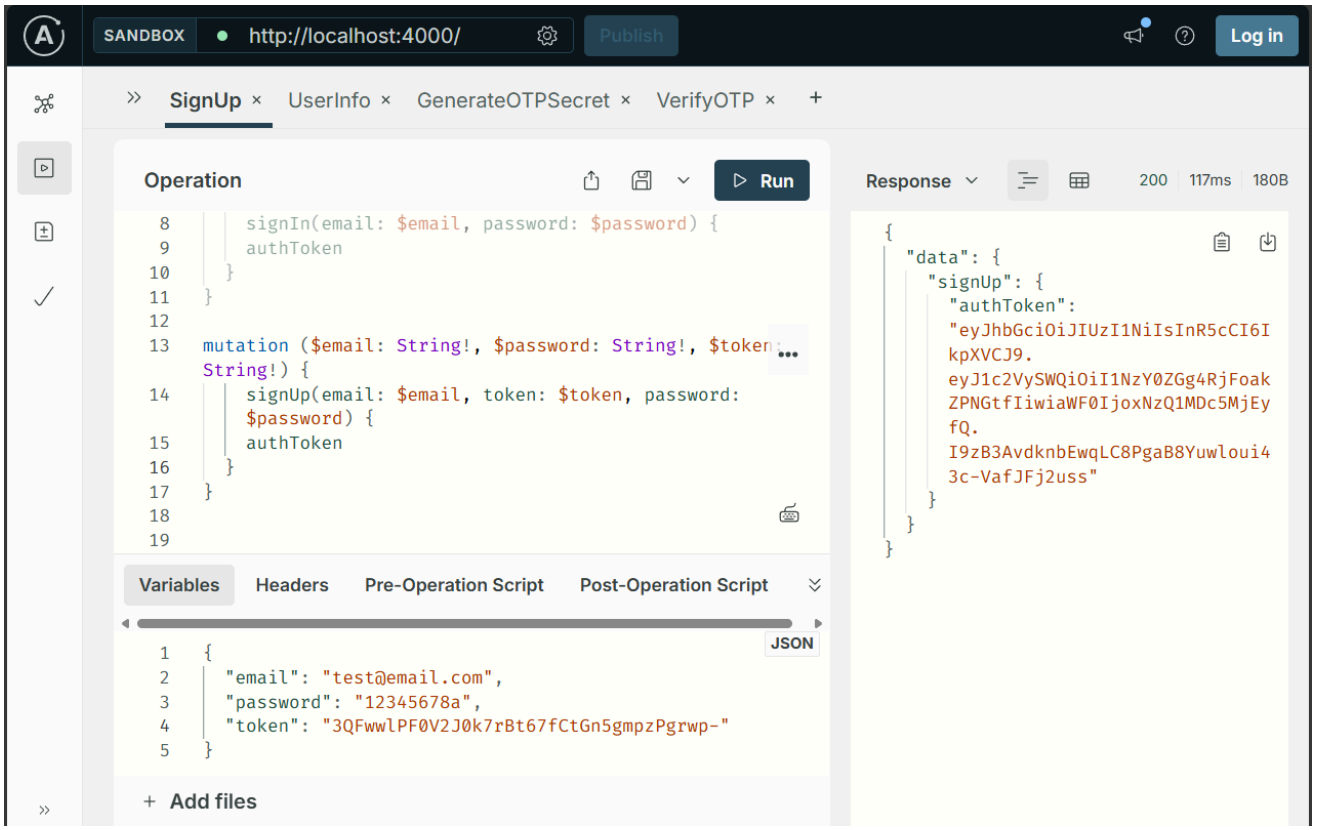

Рисунок 3.17 – Виконання мутації «signUp»

Отриманий JWT містить закодоване значення ID користувача та час закінчення його дії (рис. 3.18).

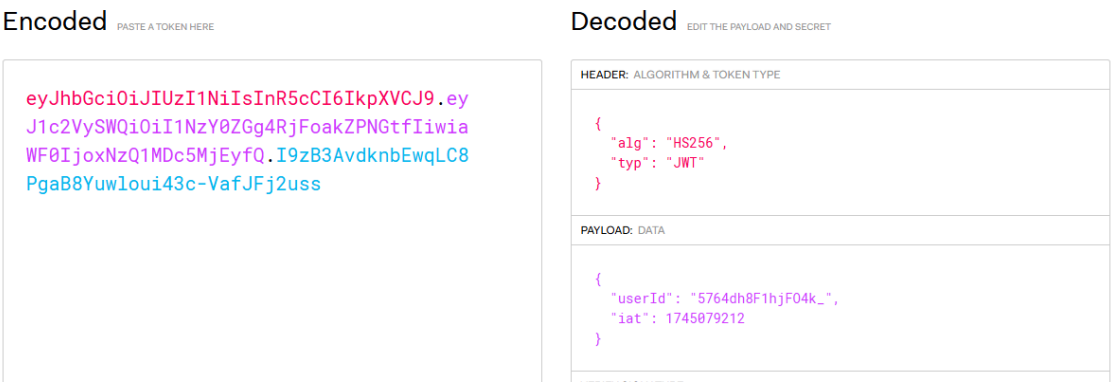

Рисунок 3.18 – JWT у закодованому та розкодованому вигляді

Наступним кроком є «активація» БФА, для цього необхідно виконати дві мутації. Перша – генерує секрет, друга – перевіряє, чи все правильно працює, шляхом одноразової перевірки TOTP. У разі успішного підтвердження функціонал БФА буде увімкнено для даного облікового запису.

Для генерації секрету використовується мутація «generateSecret», яка має містити заголовок авторизації із JWT. У відповідь можна отримати секрет у вигляді base32 та URL, для зручнішого генерування QR-коду (рис. 3.19).

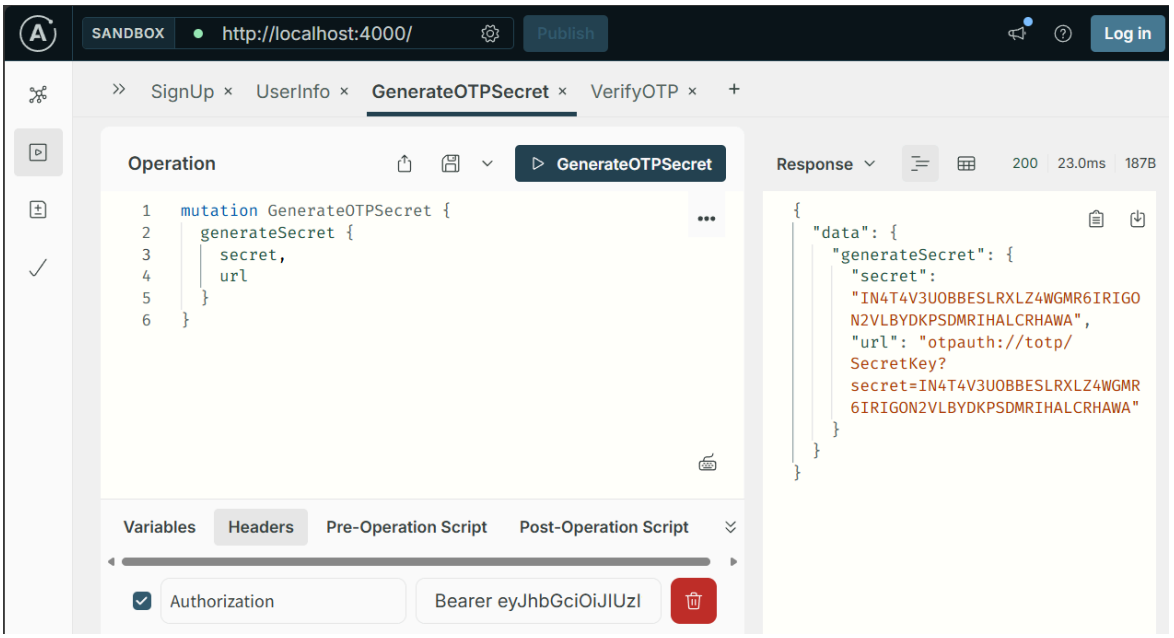

Рисунок 3.19 – Виконання мутації «generateSecret»

Для підтвердження активації необхідно використати мутацію «verifySecret». Перед цим необхідно отримати актуальний одноразовий пароль (рис. 3.20).

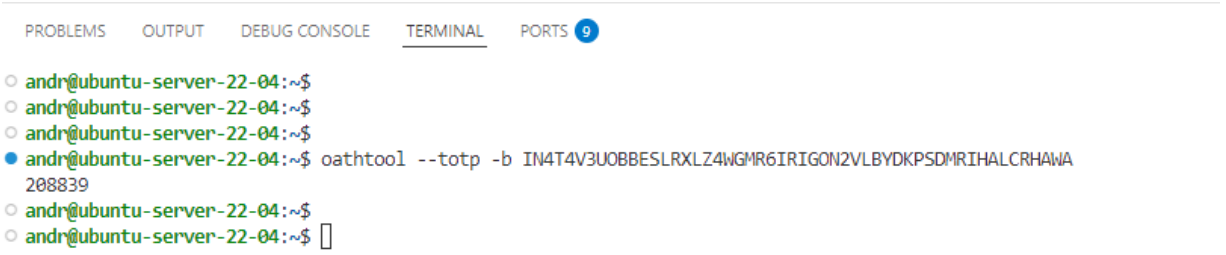

Рисунок 3.20 – Генерація TOTP-коду

Згенерований пароль необхідно відправити у заголовку «X-2FA» разом із JWT мутації «verifySecret». У якості відповіді сервер поверне повідомлення про успіх чи помилку, якщо така виникне у процесі (рис. 3.21).

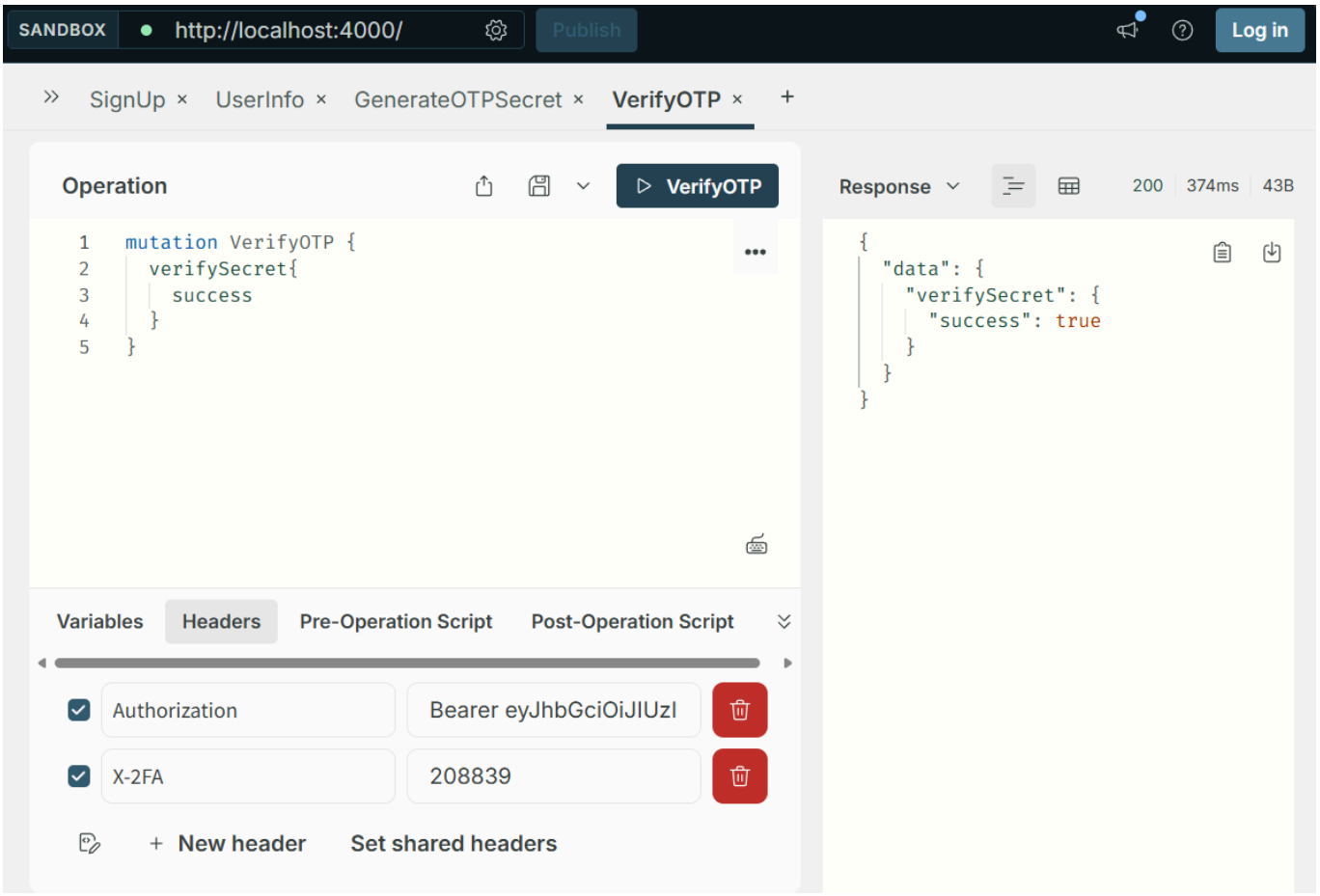

Рисунок 3.21 – Виконання мутації « verifySecret»

Після підтвердження активації БФА процес створення та налаштування облікового запису завершено. API використовує під до аутентифікації «з підвищенням рівня», тому користувачу необхідно надати додатковий фактор лише у випадку виконання «чутливої» операції. У даному демонстраційному варіанті програми такою операцією є мутація «changeBalance», задачею якої є зміна значення балансу користувача.

Для перегляду інформації про користувача, в тому числі поточне значення балансу, використовується запит «userInfo», який може повертати email, ID та баланс. Виконання цього запиту не потребує перевірки додаткового фактору, тому може виконуватися лише із наданням JWT у заголовку авторизації (рис. 3.22).

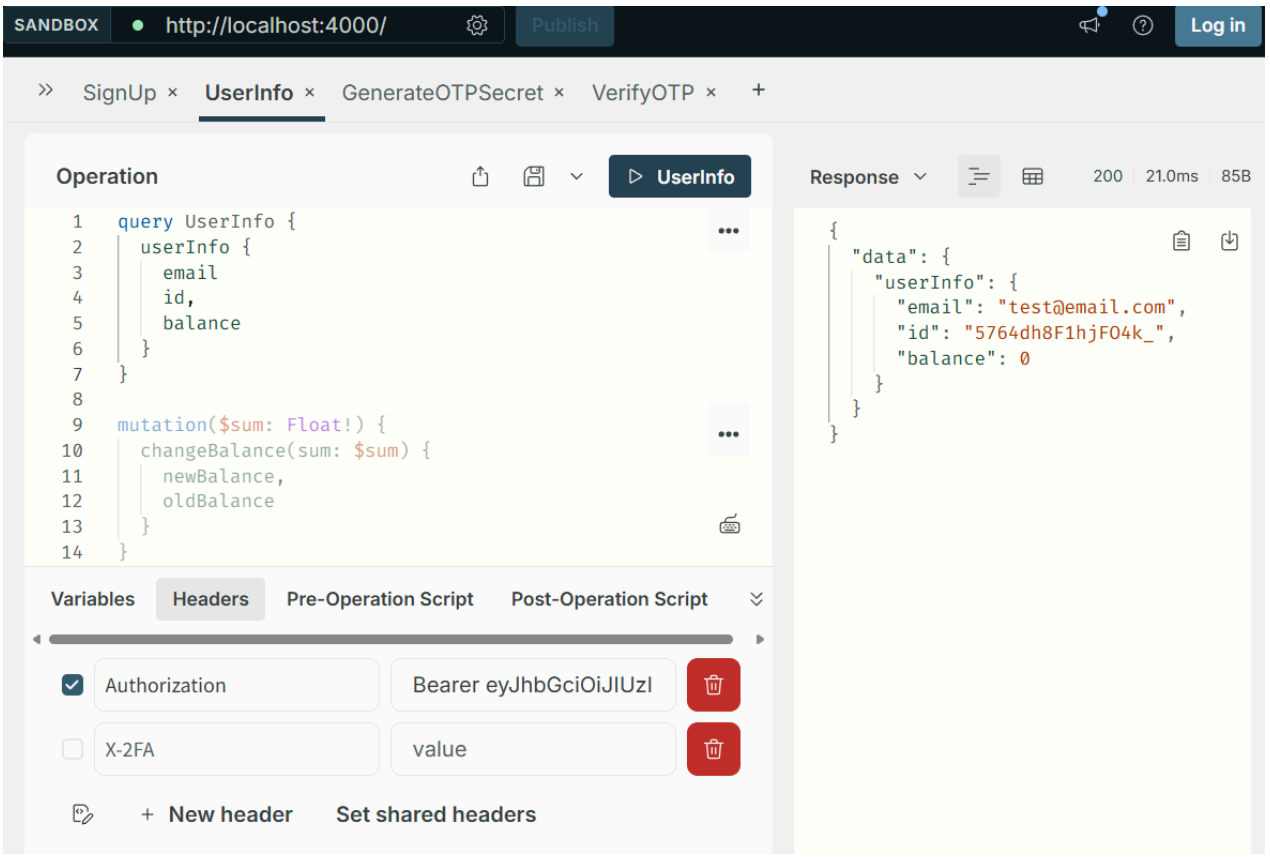

Рисунок 3.22 – Виконання запиту «userInfo»

Мутація «changeBalance» передбачає відправлення запиту, що містить нове значення балансу у вигляді змінної, а також заголовків «Authorization» та «X-2FA» із значеннями JWT та TOTP відповідно.

У випадку, якщо заголовок «X-2FA» не включено у запит або відправлений код є недійсним, сервер поверне повідомлення про помилку (рис. 3.23).

Рисунок 3.23 – Виконання мутації «changeBalance» без заголовку «X-2FA»

Якщо ж заголовок присутній, а код є правильним, то сервер повертає у відповідь попереднє та нове значення балансу (рис. 3.24).

Для демонстрації було змінено значення балансу користувача з 0 до 10. Нове значення балансу передається у змінній запиту «sum», яка очікує дані типу Float. Після успішної обробки запиту сервер повертає значення «newBalance» та «oldBalance» із попереднім та поточним балансом відповідно.

Рисунок 3.24 – Виконання мутації «changeBalance»

Для додаткової перевірки успішності операції можна виконати запит «userInfo» (який не потребує одноразового пароля) та переглянути значення балансу (рис. 3.25).

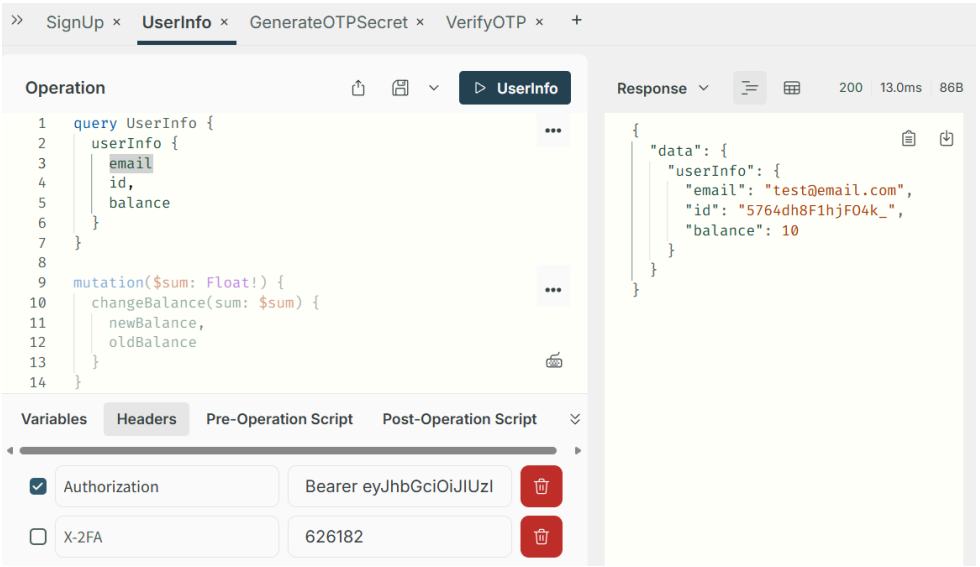

Рисунок 3.25 – Виконання запиту «userInfo» для перевірки оновленого балансу

Таким чином, було продемонстровано роботу рішення, що розроблялося в рамках даної кваліфікаційної роботи, а саме GraphQL API, що впроваджує БФА із використанням TOTP в якості додаткового фактору з технічної сторони та реалізує стратегію «з підвищенням рівня» з точки зору архітектурного підходу та досвіду користувачів.

Висновки за розділом 3

У даному розділі було розглянуто особливості розробки програмного рішення, а саме API із можливістю багатофакторної аутентифікації. З точки зору досвіду користувача було обрано стратегію «з підвищенням рівня», суть якої полягає у запиті додаткового фактору лише у випадках, коли необхідно виконати операцію, яка є «чутливішою» за інші.

З технічної сторони, для реалізації рішення було обрано технологію GraphQL, яка набирає популярність за рахунок своєї гнучкості, платформу Node.js, мову

TypeScript та Apollo Server у якості сервера GraphQL, а також бази даних PostgreSQL.

Node.js служить надійним середовищем для реалізації серверної частини проєкту, де потрібна як продуктивність при роботі з великою кількістю запитів, так і гнучкість у налаштуванні різноманітних компонентів – від авторизації та аутентифікації до динамічної обробки даних.

Використання TypeScript у розробці серверної частини дозволяє підтримати високий рівень якості коду, підвищити надійність обробки запитів у API і зробити підхід до написання коду системним. В проєктах, де безпека є пріоритетом, коректність коду та зручність його модернізації мають велике значення, а TypeScript дозволяє водночас забезпечити й те, й інше.

Зв’язка «Apollo Server + Apollo Playground» суттєво полегшує всю процедуру створення й обслуговування GraphQL-програм. Зокрема, у контексті побудови системи з багатофакторною аутентифікацією цей тандем гарантує зручне налагодження процесу аутентифікації та авторизації, документацію і швидкий пошук потенційних. Такий підхід сприяє прозорості в роботі з кодом і забезпечує гнучкість у впровадженні додаткових функцій і сервісів, пов’язаних із механізмом безпеки або з розширенням бізнес-логіки.

Обираючи PostgreSQL для реалізації бази даних для GraphQL API з багатофакторною аутентифікацією, можна отримати надійне, масштабоване і безпечне рішення для зберігання та управління даними. Це сприяє стабільній роботі системи, одночасно забезпечуючи високий рівень захисту конфіденційної інформації користувачів.

Розроблене рішення має декілька умовних складових частин, що відповідають за авторизацію, БФА та дії, пов’язані із користувачем. Частина, що відповідає за авторизацію, складається із трьох мутацій: «requestSignUp», «signUp» та «signIn».

Мутація «requestSignUp» відповідає за створення попереднього запиту на реєстрацію та повертає токен, який є необхідний для створення облікового запису за допомогою мутації «signUp», яка, окрім токена, очкує у якості змінної email та пароль користувача. У разі успішної реєстрації чи входу до системи за допомогою

мутації «signIn»., клієнт отримує від сервера JWT, який має передаватися у заголовку «Authorization» для виконання запитів у системі.

Користувач має налаштувати БФА, шляхом генерації та підтвердження секрету за допомогою мутації «generateSecret» та «verifySecret» відповідно. Після генерації секрету сервер повертає значення секрету у вигляді base32 та URL. Передавати значення TOTP-коду необхідно із використанням заголовку «X-2FA».

Для демонстрації система має запит («userInfo») та мутацію («changeBalance»), які дозволяють отримати інформацію про користувача (ID, email, баланс) та змінити поточне значення балансу. Запит не потребує використання додаткового фактору для виконання, у той час як мутація – вимагає відправлення одноразового пароля зі сторони клієнта.

Було реалізовано GraphQL API із функцією БФА, де у якості другого фактору використовується TOTP, з використанням підходу «з підвищенням рівня», де надання додаткового фактору необхідно лише у випадку виконання деяких визначених запитів чи операції.

ВИСНОВКИ

У даній кваліфікаційній роботі було досліджено питання багатофакторної аутентифікації у контексті API. Перший розділ роботи було присвячено ґрунтовному аналізу концепції багатофакторної аутентифікації: було розглянуто її визначення, основні принципи та історію розвитку. У процесі дослідження було чітко окреслено, що багатофакторна аутентифікація – це процедура підтвердження особи користувача, яка базується на поєднанні кількох незалежних факторів, що значно підвищує надійність захисту порівняно з традиційними методами, де використовується лише один фактор. Історичний огляд допоміг зрозуміти еволюцію підходів до аутентифікації, починаючи від використання простих паролів і закінчуючи застосуванням складних біометричних технологій та криптографії.

У ході дослідження було детально розглянуто класифікацію методів аутентифікації, що дозволило глибше зануритися в механізми їх роботи та проаналізувати переваги і недоліки кожного підходу. Зокрема, було приділено увагу порівнянню однофакторної та багатофакторної аутентифікації, де було продемонстровано, що, хоч однофакторні методи характеризуються простотою та зручністю, вони значно поступаються за рівнем безпеки багатофакторним рішенням, які, хоч і більш складні в імплементації, забезпечують набагато більший захист від потенційних атак. Також у цьому розділі було детально описано конкретні засоби та методи аутентифікації, які використовуються в сучасних інформаційних системах, зокрема, традиційні паролі, одноразові паролі (OTP), апаратні ключі безпеки, методики одноразового входу (SSO), а також біометричну аутентифікацію, що базується на унікальних фізичних характеристиках користувача, таких як відбитки пальців або розпізнавання обличчя.

Другий розділ роботи був присвячений практичним аспектам впровадження багатофакторної аутентифікації у сучасні цифрові сервіси, що реалізуються через API. Було проведено ґрунтовний огляд архітектур API, серед яких найбільш популярними є REST і GraphQL, а також розглянуто інші менш поширені підходи.

Особливу увагу у другому розділі було приділено аналізу мов програмування та фреймворків, які використовуються у розробці API. У роботі розглянуто популярні технології, такі як Node.js, Django, Spring Boot та інщі, що демонструють різні підходи до структурування проектів і реалізації безпеки. Так, використання Node.js характеризується високою гнучкістю, широким набором бібліотек і модулів, що дозволяє швидко інтегрувати різноманітні методи аутентифікації, у тому числі багатофакторні. Django, як фреймворк Python, надає стійку основу із вбудовании механізмами контролю доступу і аутентифікації. Spring Boot, як фреймворк Java, орієнтується на корпоративний сегмент, пропонуючи гнучкі налаштування безпеки і високу масштабованість, що є важливим для реалізації складних і надійних систем багатофакторної аутентифікації. Вибір мови програмування та інструментарію є одним із ключових факторів, що визначають успіх у реалізації безпечної багатофакторної аутентифікації на рівні API, адже від цього залежить не лише швидкість розробки, а й довгострокова підтримка.

Іншою важливою частиною другого розділу було дослідження підходів для впровадження багатофакторної аутентифікації у API та веб-рішеннях загалом. Дані рішення розглядалися не стільки із технічної сторони, скільки зі сторони досвіду користувачів. Було визначено чотири основних підходи: «завжди ввімкнено», «з можливістю вибору», «з підвищенням рівня» та «з урахуванням часу».

Підхід «завжди ввімкнено» передбачає, що БФА використовується кожного разу, коли користувач заходить у систему. Підхід «з можливістю вибору» надає можливість користувачам системи самим обрати, чи бажають вони підвищити рівень захисту свого облікового запису за рахунок використання БФА. Стратегія «з підвищенням рівня» передбачає, що перевірка додаткових факторів виконується у випадку, коли користувач намагається виконати певні «чутливі» операції. Якщо ж користувач виконує лише звичайні дії, достатньо використовувати лише один фактор. Підхід «з урахуванням часу», в свою чергу, ґрунтується на принципі, що перший раз користувач входить у систему, з використанням декількох факторів, але у випадку використання того самого пристрою чи браузера, проходити всі етапи

перевірки потворно не потрібно до моменту, поки не пройде встановлений системою час життя певного фактору.

Дані підходи дозволяють розробникам «маневрувати», щоб знайти оптимальний баланс між безпекою та зручністю для свого програмного рішення.

У третьому розділі кваліфікаційної роботи було виконано розробку GraphQL API із БФА, впровадження якої виконано із використанням підходу «з підвищенням рівня» зі сторони досвіду користувача, та використанням TOTP із технічної сторони. Створене програмне рішення складається із декількох логічних компонентів, а саме: блоку аутентифікації, блоку БФА та блоку запитів «всередині» системи.

Частина, що відповідає за аутентифікацію, реалізує мутації для створення запити на реєстрацію, безпосередньої реєстрації та входу у систему. Для реєстрації необхідно попередньо отримати токен, а також надати пароль та пошту у вигляді змінних мутації. У випадку успішної аутентифікації, система поверне JWT, який потрібно надавати у заголовку «Authorization» у подальших запитах до сервера.

Блок БФА реалізує дві мутації, які дозволяють згенерувати секрет для TOTP та підтвердити активацію, шляхом відправлення одноразового пароля. Після генерації секрету, клієнт отримує від сервера відповідь із значенням, яке необхідно використати у рішеннях для генерації одноразових паролів.

Користувачам, які увійшли в систему, дозволяється робити запит, який не потребує надання TOTP та мутації, яка не буде працювати без заголовку «X-2FA», що містить актуальний одноразовий пароль. Таким чином було реалізовано підхід «із підвищенням рівня».

Відповідно до проведеного дослідження можна стверджувати, що багатофакторна аутентифікація сьогодні є необхідним елементом будь-яких інформаційних систем, що прагнуть забезпечити високий рівень захисту даних і зменшити можливості для компрометації облікових записів. Поєднання різних факторів аутентифікації утворює надійний бар’єр проти більшості сучасних видів атак, що робить систему безпеки більш стійкою та адаптивною. Разом із тим, впровадження багатофакторних механізмів вимагає ретельного вибору підходу, що

дозволяє як забезпечити високий рівень захисту, так і не погіршити досвід користувачів, зберігаючи зручність і швидкість доступу до сервісів.

Таким чином, результати даної роботи мають практичну цінність і можуть бути використані розробниками та IT-фахівцями для побудови сучасних систем аутентифікації API з урахуванням актуальних вимог до інформаційної безпеки. Використання отриманих даних сприятиме підвищенню надійності, масштабованості та зручності цифрових сервісів, що позитивно вплине на довіру користувачів і сприятиме успішному функціонуванню інформаційних систем у довгостроковій перспективі. Впровадження багатофакторної аутентифікації, у свою чергу, є необхідним кроком на шляху до створення безпечних цифрових рішень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. BARE Cybersecurity - Build A Resilient Enterprise. Is a Strong Password Enough to Protect Your Business? The Simple Step to Block 99.9% of Attacks [Електронний ресурс] / BARE Cybersecurity - Build A Resilient Enterprise // LinkedIn: Log In or Sign Up. – Режим доступу: https://www.linkedin.com/pulse/strong-password-enough-protect-your-business-simple-ihr4f

2. Багатофакторна аутентифікація (БФА) | Захисний комплекс Microsoft [Електронний ресурс] // Захисний комплекс Microsoft. – Режим доступу: https://www.microsoft.com/uk-ua/security/business/identity-access/microsoft-entra-mfa-multi-factor-authentication#:~:text=Що%20таке%20багатофакторна%20аутентифікація?,чи%20ввести%20код%20із%20телефона.

3. Google Says Using Two-Factor Authentication Reduces Account Hacks 50 Percent [Електронний ресурс] // Extremetech – Режим доступу: https://www.extremetech.com/internet/331471-google-says-using-two-factor-authentication-reduces-account-hacks-50-percent

4. 2024 Data Breach Investigations Report [Електронний ресурс] // Verizon: Wireless, Internet, TV and Phone Services | Official Site. – Режим доступу: https://www.verizon.com/business/resources/Te3/reports/2024-dbir-data-breach-investigations-report.pdf

5. Breaking Down the 2024 Verizon Data Breach Investigations Report [Електронний ресурс] // SpyCloud. – Режим доступу: https://spycloud.com/blog/verizon-2024-data-breach-report-insights/

6. Багатофакторна аутентифікація [Електронний ресурс] // iIT Distribution. – Режим
доступу: https://www.linkedin.com/pulse/strong-password-enough-protect-your-business-simple-ihr4f

7. Дія.Бізнес [Електронний ресурс] // Дія.Бізнес Старт. – Режим доступу: https://business.diia.gov.ua/entrepreneur-handbook/item/bagatofaktorna_avtentifikaciya_abo_mfa

8. SuperTokens. The Multifaceted Benefits of Multi-Factor Authentication [Electronic resource] / SuperTokens // SuperTokens, Open Source User Authentication. – Режим доступу: https://supertokens.com/blog/benefits-of-multi-factor-authentication

9. 2024 Data Breach Investigations Report [Electronic resource] // Verizon Business. – Режим доступу: https://www.verizon.com/business/resources/reports/dbir

10. The History of Authentication | OmniCyber Security [Electronic resource] // OmniCyber Security. – Режим доступу: https://www.omnicybersecurity.com/history-authentication/

11. World Password Day: A Brief (yet Long) History of Passwords - The LastPass Blog [Electronic resource] // The LastPass Blog - The Last Password You'll Ever Need. – Режим доступу: https://blog.lastpass.com/posts/world-password-day-a-brief-yet-long-history-of-passwords

12. Альфонс Бертільон - Google Arts & Culture [Електронний ресурс] // Google Arts & Culture. – Режим доступу: https://artsandculture.google.com/entity/m03rb_v?hl=uk

13. University of Glasgow - MyGlasgow - Library - Collections - Medical Humanities - Forensic Medicine Collections - Case Files - Evidence of Fingerprints [Електронний ресурс] // The University of Glasgow, Scotland, UK. A world top 100 university. – Режим доступу: https://www.gla.ac.uk/myglasgow/library/collections/medicalhumanities/forensic%20medicine/casefiles/evidenceoffingerprints/#:~:text=It%20was%20however,%20Sir%20Edward,,%20loops,%20whorls%20and%20composites.

14. History of Authentication: From Zero to Hero - Cybersecurity ASEE [Електронний ресурс] // Cybersecurity ASEE. – Режим доступу: https://cybersecurity.asee.io/blog/history-of-authentication/

15. History of the Internet [Електронний ресурс] // Evolution of the Internet: The Development of PKI. – Режим доступу: https://www.globalsign.com/en/blog/sg/history-internet-development-pki

16. What Does OTP Mean? Everything You'd Ever Need to Know | [Електронний ресурс] // SMS Marketing Solution Provider | SMS & Voice Marketing APIs - SMSCountry. – Режим доступу: https://www.smscountry.com/blog/what-is-otp/

17. TLS Security 2: A Brief History of SSL/TLS | Acunetix [Електронний ресурс] // Acunetix. – Режим доступу: https://www.acunetix.com/blog/articles/history-of-tls-ssl-part-2/

18. Weatherbed J. 10 years ago, Apple’s Touch ID convinced us to lock our phones [Електронний ресурс] / Jess Weatherbed // The Verge. – Режим доступу: https://www.theverge.com/23868464/apple-iphone-touch-id-fingerprint-security-ten-year-anniversary

19. What is Behavioral Authentication? | IT Wiki | GateKeeper 2FA Identity Management [Електронний ресурс] // GateKeeper Proximity Passwordless 2FA. – Режим
доступу: https://gkaccess.com/support/information-technology-wiki/behavioral-authentication/

20. 5 User Authentication Methods that Can Prevent the Next Breach [Electronic resource] // ID R&D. – Режим доступу: https://www.idrnd.ai/5-authentication-methods-that-can-prevent-the-next-breach/

21. Що таке однофакторна аутентифікація? Плюси та мінуси SFA | Hideez [Електронний ресурс] // Passwordless Workforce Identity Solutions | Hideez. – Режим доступу: https://hideez.com/uk-ua/blogs/news/single-factor-authentication

22. Top 200 Most Common Passwords [Електронний ресурс] // NordPass. – Режим доступу: https://nordpass.com/most-common-passwords-list/

23. 2025 Multi-Factor Authentication (MFA) Statistics & Trends to Know [Електронний ресурс] // JumpCloud. – Режим доступу: https://jumpcloud.com/blog/multi-factor-authentication-statistics

24. Consumer Electronics Biometric Market Size to Hit USD 160.50 Billion by 2034 [Electronic resource] // Precedence Research - Statistical Platform for Market Intelligence, Market Research & Consulting. – Режим доступу: https://www.precedenceresearch.com/consumer-electronics-biometric-market

25. The Kernel. What you need to know about managing authentication in your organization [Електронний ресурс] / The Kernel // LinkedIn. – Режим доступу: https://www.linkedin.com/pulse/what-you-need-know-managing-authentication-your-organization-

26. 50+ Password Statistics & Trends to Know in 2024 [Електронний ресурс] // JumpCloud. – Режим доступу: https://jumpcloud.com/blog/password-statistics-trends

27. Create a strong password & a more secure account - Google Account Help [Електронний ресурс] // Google Help. – Режим доступу: https://support.google.com/accounts/answer/32040

28. Що таке менеджер паролів і як він працює? [Електронний ресурс] // NordVPN. – Режим доступу: https://nordvpn.com/uk/password-manager/

29. 8 Scary Statistics about the Password Reuse Problem [Електронний ресурс] // Enzoic. – Режим доступу: https://www.enzoic.com/blog/8-stats-on-password-reuse/

30. Bonnie E. 60+ Social Engineering Statistics [Updated 2025] [Електронний ресурс] / Emily Bonnie // Secureframe. – Режим доступу: https://secureframe.com/blog/social-engineering-statistics

31. Crafting the Perfect Password Policy: 12 Must-Have Elements [Електронний ресурс] // GIBRALTAR. – Режим доступу: https://gibraltarsolutions.com/blog/password-policy/

32. What is One-Time Password (OTP)? [Електронний ресурс] // Yubico. – Режим доступу: https://www.yubico.com/resources/glossary/otp/

33. One-Time Password (OTP) [Електронний ресурс] // LicenseSpring. – Режим доступу: https://licensespring.com/blog/glossary/one-time-password

34. Richards K. What is a one-time password (OTP)? Definition from SearchSecurity [Електронний ресурс] / Kathleen Richards, Ivy Wigmore // Search

Security. – Режим доступу: https://www.techtarget.com/searchsecurity/definition/one-time-password-OTP

35. How to Secure Your Online Identity with Security Keys [Електронний ресурс] // CIS. – Режим доступу: https://www.cisecurity.org/insights/blog/how-to-secure-your-online-identity-with-security-keys

36. Lutkevich B. What is smart card? | Definition from TechTarget [Електронний ресурс] / Ben Lutkevich, Linda Rosencrance, Michael Cobb // Search Security. – Режим доступу: https://www.techtarget.com/searchsecurity/definition/smart-card

37. Дослідження Single Sign-On: реалізація, плюси, мінуси і ключові особливості [Електронний ресурс] // DOU. – Режим доступу: https://dou.ua/forums/topic/47968/

38. What is SSO? | How single sign-on works [Електронний ресурс] // Cloudflare. – Режим доступу: https://www.cloudflare.com/learning/access-management/what-is-sso/

39. Що таке SAML (мова розмітки тверджень системи безпеки)? | Захисний комплекс Microsoft [Електронний ресурс] Microsoft. – Режим доступу: https://www.microsoft.com/uk-ua/security/business/security-101/what-is-security-assertion-markup-language-saml

40. OAuth 2.0 – OAuth [Електронний ресурс] // OAuth Community Site. – Режим доступу: https://oauth.net/2/

41. What Is Biometric Authentication? Definition, Benefits, and Tools - Spiceworks [Електронний ресурс] // Spiceworks Inc. – Режим доступу: https://www.spiceworks.com/it-security/identity-access-management/articles/what-is-biometric-authentication-definition-benefits-tools/

42. How China uses facial recognition to control human behavior [Електронний ресурс] // CNET. – Режим доступу: https://www.cnet.com/news/politics/in-china-facial-recognition-public-shaming-and-control-go-hand-in-hand/

43. What is an API? - Application Programming Interface Explained - AWS [Електронний ресурс] // Amazon Web Services, Inc. – Режим доступу: https://aws.amazon.com/what-is/api/

44. Maximizing Software Performance - The Role of Efficiency | Reintech media [Електронний ресурс] // Hire Developers for SaaS teams - Software Developers as a Service | Reintech. – Режим доступу: https://reintech.io/terms/category/efficiency-software-development

45. ByteByteGo. A Crash Course on Scaling the API Layer [Електронний ресурс] / ByteByteGo // ByteByteGo Newsletter | Alex Xu | Substack. – Режим доступу: https://blog.bytebytego.com/p/a-crash-course-on-scaling-the-api

46. 2023 State of the API Report | API Technologies [Електронний ресурс] // Postman API Platform. – Режим доступу: https://www.postman.com/state-of-api/2023/api-technologies/

47. The History of REST APIs - ReadMe: Resource Library [Електронний ресурс] // ReadMe: Resource Library. – Режим доступу: https://readme.com/resources/the-history-of-rest-apis

48. What is REST?: REST API Tutorial [Електронний ресурс] // REST API Tutorial. – Режим доступу: https://restfulapi.net/

49. Bennett T. REST API Principles | A Comprehensive Overview [Електронний ресурс] / Terence Bennett // Blog. – Режим доступу: https://blog.dreamfactory.com/rest-apis-an-overview-of-basic-principles

50. McKenzie C. HTTP request methods explained [Електронний ресурс] / Cameron McKenzie // TheServerSide | Your Java Community discussing server side development. – Режим доступу: https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/HTTP-methods

51. REST API URI [Електронний ресурс] // Security-first, AI-powered networking | HPE Aruba Networking. – Режим доступу: https://arubanetworking.hpe.com/techdocs/AOS-CX/10.13/HTML/rest_v10-0x/Content/Chp_Intro/uri-10.htm

52. Strauss L. Webhook vs. API: Differences + when to use each | Zapier [Електронний ресурс] / Luke Strauss // Automate without limits | Zapier. – Режим доступу: https://zapier.com/blog/webhook-vs-api/

53. What is a webhook? [Електронний ресурс] // Red Hat - We make open source technologies for the enterprise. – Режим доступу: https://www.redhat.com/en/topics/automation/what-is-a-webhook

54. What are the benefits and drawbacks of webhooks for real-time data delivery? [Електронний ресурс] // LinkedIn: Log In or Sign Up. – Режим доступу: https://www.linkedin.com/advice/0/what-benefits-drawbacks-webhooks-real-time-data

55. Gudabayev T. A Brief History of GraphQL [Електронний ресурс] / Tamerlan Gudabayev // DEV Community. – Режим доступу: https://dev.to/tamerlan_dev/a-brief-history-of-graphql-2jhd

56. GraphQL | A query language for your API [Електронний ресурс] // GraphQL | A query language for your API. – Режим доступу: https://graphql.org/

57. What Is GraphQL API & How Does It Work? - scandiweb [Електронний ресурс] // scandiweb. – Режим доступу: https://scandiweb.com/blog/what-is-graphql-api-how-does-it-work/

58. GraphQL | How to Use GraphQL Subscriptions to Build Real-Time Apps on AWS | Amazon Web Services (AWS) [Електронний ресурс] // Amazon Web Services, Inc. – Режим доступу: https://aws.amazon.com/ru/graphql/graphql-subscriptions-real-time/

59. GeeksforGeeks. Advantage & Disadvantage of GraphQL - GeeksforGeeks [Електронний ресурс] / GeeksforGeeks // GeeksforGeeks. – Режим доступу: https://www.geeksforgeeks.org/advantage-disadvantage-of-graphql/

60. Best Programming Languages to Develop REST API [Електронний ресурс] // ToolJet. – Режим доступу: https://blog.tooljet.ai/best-programming-languages-for-rest-api-development/

61. JavaScript language overview - JavaScript | MDN [Електронний ресурс] // MDN Web Docs. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_overview

62. Documentation - TypeScript for JavaScript Programmers [Електронний ресурс] // TypeScript: JavaScript With Syntax For Types. – Режим доступу: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html

63. Ó Tuama D. Advantages of JavaScript [Електронний ресурс] / Daragh Ó Tuama // Code Institute. – Режим доступу: https://codeinstitute.net/global/blog/advantages-of-javascript/

64. Python Introduction - GeeksforGeeks [Електронний ресурс] // GeeksforGeeks. – Режим доступу: https://www.geeksforgeeks.org/introduction-to-python/

65. Gavrilova Y. Pros and Cons of Python [Electronic resource] / Yulia Gavrilova // Pros and Cons of Python. – Режим доступу: https://serokell.io/blog/python-pros-and-cons

66. Designing Top-Tier APIs in Java for Web & App Environments. [Електронний ресурс] // NeoSOFT. – Режим доступу: https://www.neosofttech.com/blogs/java-api-development

67. Introduction to Java - GeeksforGeeks [Електронний ресурс] // GeeksforGeeks. – Режим доступу: https://www.geeksforgeeks.org/introduction-to-java/

68. Pros and Cons of Java Development in 2023 [Електронний ресурс] // Software Development Company | Netguru. – Режим доступу: https://www.netguru.com/blog/java-pros-and-cons

69. GeeksforGeeks. PHP Introduction - GeeksforGeeks [Електронний ресурс] / GeeksforGeeks // GeeksforGeeks. – Режим доступу: https://www.geeksforgeeks.org/php-introduction/

70. Advantages and Disadvantages of PHP - ellow.io [Електронний ресурс] // Ellow Talent. – Режим доступу: https://ellow.io/advantages-and-disadvantages-of-php/

71. GeeksforGeeks. Introduction to C# - GeeksforGeeks [Електронний ресурс] / GeeksforGeeks // GeeksforGeeks. – Режим доступу: https://www.geeksforgeeks.org/introduction-to-c-sharp/

72. Team A. E. The Good and the Bad of C# Programming [Електронний ресурс] / AltexSoft Editorial Team // AltexSoft. – Режим доступу: https://www.altexsoft.com/blog/c-sharp-pros-and-cons/

73. Ahangari O. Go for Beginners: A Comprehensive Introduction to Golang [Електронний ресурс] / Omid Ahangari // Medium. – Режим доступу: https://medium.com/@omidahn/go-for-beginners-a-comprehensive-introduction-to-golang-fca685759fd8

74. Express web framework (Node.js/JavaScript) - Learn web development | MDN [Електронний ресурс] // MDN Web Docs. – Режим доступу: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/Express_Nodejs

75. Використання проміжних обробників Express [Електронний ресурс] // Express - Node.js web application framework. – Режим доступу: https://expressjs.com/uk/guide/using-middleware.html

76. Christianto D. DJango Rest Framework [Електронний ресурс] / David Christianto // Medium. – Режим доступу: https://medium.com/@david.christianto05/django-rest-framework-6466ffb09b78

77. Tomazic N. Django REST Framework: Pros and Cons [Електронний ресурс] / Nik Tomazic // Test-Driven Development, Microservices, Web Development Courses | TestDriven.io. – Режим доступу: https://testdriven.io/blog/drf-pros-cons/

78. Spring Boot [Електронний ресурс] // Spring Boot. – Режим доступу: https://spring.io/projects/spring-boot

79. Building REST services with Spring [Електронний ресурс] // Spring. – Режим доступу: https://spring.io/guides/tutorials/rest.

80. Laravel - Overview [Електронний ресурс] // Tutorials Point. – Режим доступу: https://www.tutorialspoint.com/laravel/laravel_overview.htm

81. Database: Getting Started - Laravel 12.x - The PHP Framework For Web Artisans [Електронний ресурс] // Laravel - The PHP Framework For Web Artisans. – Режим доступу: https://laravel.com/docs/12.x/database

82. Introduction [Електронний ресурс] // Gin Web Framework. – Режим доступу: https://gin-gonic.com/docs/introduction/

83. Morrone C. 4 Examples of Multi-Factor Authentication Approaches | Curity [Електронний ресурс] / Christopher Morrone // Curity: Identity Management for APIs, Apps and Websites. – Режим доступу: https://curity.io/resources/learn/approaches-to-mfa/

84. Wrapping a REST API in GraphQL | GraphQL [Електронний ресурс] // GraphQL | A query language for your API. – Режим доступу: https://graphql.org/blog/2016-05-02-rest-api-graphql-wrapper/

85. Node.js – The V8 JavaScript Engine [Електронний ресурс] // Node.js – Run JavaScript Everywhere. – Режим доступу: https://nodejs.org/en/learn/getting-started/the-v8-javascript-engine

86. Documentation - TypeScript for JavaScript Programmers [Електронний ресурс] // TypeScript: JavaScript With Syntax For Types. – Режим доступу: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html

87. Introduction to Apollo Server [Електронний ресурс] // Apollo GraphQL Docs. – Режим доступу: https://www.apollographql.com/docs/apollo-server

88. PostgreSQL: About [Електронний ресурс] // PostgreSQL: The world's most advanced open source database. – Режим доступу: https://www.postgresql.org/about/

ДОДАТОК А

КОД СТВОРЕНОГО ПРОГРАМНОГО РІШЕННЯ

Код auth/schema.ts

```
export const authTypes = `#graphql

type Mutation {
  requestSignUp(email: String!): RequestSignUpResult!

  signUp(email: String!, token: String!, password: String!): AuthResult!

  generateSecret: GenerateOTPResult!

  signIn(email: String!, password: String!): AuthResult!
}

type RequestSignUpResult {
  success: Boolean!
}

type AuthResult {
  authToken: String!
}

type GenerateOTPResult {
  secret: String!
}
`
```

Код auth.ts

```
import { GraphQLError } from "graphql"
import db, { genId } from "../../modules/db/index.js"
import bcrypt from "bcrypt"
import { nanoid } from "nanoid";
import { DateTime } from "luxon"
import { generateJWT } from "modules/auth/auth.js";



const SALT_ROUNDS = 10;

export const authResolvers = {
   Mutation: {
      requestSignUp: async (_, {email}) => {
         await db.signUpRequest.deleteMany({where: {email}});

         const token = nanoid(36);
         await db.signUpRequest.create({
            data: {
               id: genId(),
               email,
               tokenHash: await bcrypt.hash(token, SALT_ROUNDS),
               expiresAt: DateTime.now().plus({minutes: 10}).toJSDate(),

            }
         });

         // Send token
         console.log("TOKEN:", token)
         return {success: true}
```

```
        },

        signUp: async (_, {token, email, password}) => {
            const request =await db.signUpRequest.findUnique({where: {email}});
            if (!request) throw new GraphQLError("Sign up request not found");
                if (request.expiresAt < new Date()) throw new GraphQLError("Sign up
request expired")
            // compare token with token hash
            const validToken = await bcrypt.compare(token, request.tokenHash);
            if (!validToken) throw new GraphQLError("Invalid token");

            // check if user already exists
            const existingUser = await db.user.findUnique({where: {email}})
            if (existingUser) throw new GraphQLError("User already exists")



            // create user and redeem token
            const [user] = await db.$transaction([
                db.user.create({
                    data: {
                        id: genId(),
                        email,
                        secret: null,
                        secret_verified: null,
                        passwordHash: await bcrypt.hash(password, SALT_ROUNDS),
                    }
                }),
                db.signUpRequest.update({
                    where: {id: request.id},
```

```
                data: {redeemedAt: new Date()},
            }),

        ])


        // generate auth token
        const authToken = generateJWT(user.id);
        console.log(authToken);
        return { authToken };
    },

    signIn: async (_, {email, password}) => {
        // check if user exists
        const user = await db.user.findUnique({where: {email}});
        if (!user) throw new GraphQLError("Invalid email and/or password");

        // check pswd
                    const correctPassword = await bcrypt.compare(password,
user.passwordHash);
                if (!correctPassword) throw new GraphQLError("Invalid email and/or
password");

        // gen token
        const authToken = generateJWT(user.id);
        console.log(authToken)
        return { authToken };
    }
  }
}
```

Код mfa/schema.ts

```
export const mfaTypes = `#graphql

  type Mutation {
  generateSecret: GenerateOTPResult!

  verifySecret: verifySecretResult!
}

type GenerateOTPResult {
  secret: String!
  url: String!
}

type verifySecretResult {
  success: Boolean!
}
`
```

Код mfa.ts

```
import { GraphQLError } from "graphql";
import db from "../../modules/db/index.js"
import { generateSecret, verifySecret } from "modules/auth/auth.js";

export const mfaResolvers = {
   Mutation: {
      generateSecret: async (_, __, {user}) => {
         // check if user exists
         const userId = user?.id;
         console.log(userId);
         if (!userId) throw new GraphQLError("User not found");

               const existingUser = await db.user.findUniqueOrThrow({where: {id: userId}});
         if (!existingUser) throw new GraphQLError("User not found");

         // check if secret is already enabled
          if (existingUser.secret_verified) throw new GraphQLError("Secret is already verified");

         // generate secret
         const secret = generateSecret();

         // save secret in db
         await db.user.update({
            where: {id: userId},
            data: {
               secret: secret.base32,
               secret_verified: false
```

```
            }
        });

        return {secret: secret.base32, url: secret.otpauth_url};
    },

    verifySecret: async (_, __, {user, sfToken}) => {
      const userId = user?.id;
      if (!userId) throw new GraphQLError("User not found");
      if (!user.secret) throw new GraphQLError("Please generate secret first");

      // check if already verified
      if (user.secret_verified) throw new GraphQLError("Already verified");

      const isValid = await verifySecret(userId, sfToken);
      if (!isValid) throw new GraphQLError("Wrong OTP");

      await db.user.update({
        where: {id: userId},
        data: {
          secret_verified: true
        }
      })
      return {success: isValid};
    }
  },
};
```

Код user/schema.ts

```
export const userTypes = `#graphql
  type Query {
    userInfo: UserDetails
   }

   type Mutation {
    changeBalance(sum: Float!): Balance
   }

  type UserDetails {
    id: ID!,
    email: String!,
    balance: Float!
  }

  type Balance {
    newBalance: Float,
    oldBalance: Float
  }
  `
```

Код user.ts

```
import { GraphQLError } from "graphql";
import { validateOTP } from "modules/auth/auth";
import db from "modules/db";

export const userResolvers = {
  Query: {
    userInfo: (_, __, {user}) => {
      return user;
    }
  },

  Mutation: {
    changeBalance: async (_, {sum}, {user, sfToken}) => {
      const isValid = await validateOTP(user, sfToken);
      if (!isValid) throw new GraphQLError("OTP is not valid");

      await db.user.update({
        where: {id: user.id},
        data: {balance: sum},
       });

      return {newBalance: sum, oldBalance: user.balance};
    }
  }
};
```

Код gql/index.ts

```
import { authTypes } from './auth/schema'
import {mergeTypeDefs} from '@graphql-tools/merge'
import { userTypes } from './user/schema';
import { mfaTypes } from './mfa/schema';


export * from './auth/auth'
export * from './user/user'
export * from './mfa/mfa'
export const typeDefs = mergeTypeDefs([authTypes, userTypes, mfaTypes]);
```

Код modules/auth/auth.ts

```
import { GraphQLError } from "graphql";
import jwt from "jsonwebtoken"
import db from "modules/db";
import speakeasy from "speakeasy";

export const generateJWT = (userId: string) => {
    return jwt.sign({userId}, SECRET);
}

export const verifyJWT = async (token: string) => {
    const value = jwt.verify(token, SECRET)
    console.log(value)

    const { userId } = jwt.verify(token, SECRET) as {userId: string};
    return await db.user.findUnique({where: { id: userId }});
}

export const generateSecret = () => {
    return speakeasy.generateSecret();
}

export const verifySecret = async (userId, otp) => {
    // get user secret
    const { secret } = await db.user.findUnique({where: { id: userId }});
    if (!secret) throw new GraphQLError("Plese generate a secret first");

    // validate OTP
    const tokenValidates = speakeasy.totp.verify({
        secret: secret,
```

```
        encoding: 'base32',
        token: otp,
        window: 0
      });

    return tokenValidates;
}

export const validateOTP = async (user, otp) => {
    if (!user) throw new GraphQLError("Not authorized");
    if (!user.secret) throw new GraphQLError("Plese generate a secret first");
    if (!user.secret_verified) throw new GraphQLError("Secret not verified");

    // validate OTP
    const tokenValidates = speakeasy.totp.verify({
        secret: user.secret,
        encoding: 'base32',
        token: otp,
        window: 0
      });

    return tokenValidates;
}
```

Код modules/db/index.ts

```
import { PrismaClient } from '@prisma/client';
import { nanoid } from 'nanoid';

const db = new PrismaClient({ log: ['error', 'info', 'query', 'warn'] });
export default db;

export const genId = () => nanoid(16);
```

Код index.ts

```
import { ApolloServer } from "@apollo/server";
import { startStandaloneServer } from "@apollo/server/standalone"
import { authResolvers, mfaResolvers, typeDefs, userResolvers } from "./gql/index.js";
import { verifyJWT } from "modules/auth/auth.js";
import { GraphQLError } from "graphql";

const resolvers = [authResolvers, userResolvers, mfaResolvers];
const server = new ApolloServer({
    typeDefs,
    resolvers,
})
//@ts-ignore
const { url } = await startStandaloneServer(server, {
    context: async ({ req }) => {
        const authToken = req.headers.authorization?.split(' ')[1];
        const sfToken = req.headersDistinct["x-2fa"]?.at(0);
        if (!authToken) return {};
        try {
            const user = await verifyJWT(authToken);
            return {user, sfToken};
        } catch (e) {
                throw new GraphQLError("Invalid token", {extensions: {code: 'NOT_AUTHENTICATED'}})
        };
    }
})

console.log("Server ready at port", url)
```

Код resolvers.ts

```
import { GraphQLError } from "graphql"
import db, { genId } from "./modules/db/index.js"
import bcrypt from "bcrypt"
import { nanoid } from "nanoid";
import { DateTime } from "luxon"
import jwt from "jsonwebtoken"
import invariant from "invariant"

const SALT_ROUNDS = 10;

export const resolvers = {
   Query: {
      games() {
         return "234";
      },
   },
   Mutation: {
      requestSignUp: async (_, {email}) => {
         await db.signUpRequest.deleteMany({where: {email}});

         const token = nanoid(36);
         await db.signUpRequest.create({
            data: {
               id: genId(),
               email,
               tokenHash: await bcrypt.hash(token, SALT_ROUNDS),
               expiresAt: DateTime.now().plus({minutes: 10}).toJSDate(),

            }
```

```
        });

        // Send token
        console.log(token)

        return {success: true}
      },

      signUp: async (_, {token, email, password}) => {
        const request =await db.signUpRequest.findUnique({where: {email}});
        if (!request) throw new GraphQLError("Sign up request not found");
            if (request.expiresAt < new Date()) throw new GraphQLError("Sign up request expired")
        // compare token with token hash
        const validToken = await bcrypt.compare(token, request.tokenHash);
        if (!validToken) throw new GraphQLError("Invalid token");

        // check if user already exists
        const existingUser = await db.user.findUnique({where: {email}})
        if (existingUser) throw new GraphQLError("User already exists")

        // create user and redeem token
        const [user] = await db.$transaction([
          db.user.create({
            data: {
              id: genId(),
              email,
              passwordHash: await bcrypt.hash(password, SALT_ROUNDS),
            }
          }),
```

```
                db.signUpRequest.update({
                  where: {id: request.id},
                  data: {redeemedAt: new Date()},
                }),
              ])



              // generate auth token
              invariant(process.env.JWT_SECRET, "JWT_SECRET not set")
                                    const authToken = jwt.sign({userId: user.id},
"b2efb3ab1e3c2fa5d2aad22a7c0028288e3c4312");
              console.log(authToken);
              return { authToken };
            },
          }
        }
```

ДОДАТОК Б

СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ ЗА ТЕМОЮ РОБОТИ

Staroselskyi A. Strategies for implementing multi-factor authentication in web applications. / Andrii Staroselskyi, Volodymyr Nakonechnyi / VIII Міжнародна науково-практична конференція “Проблеми кібербезпеки інформаційно-телекомунікаційних систем” (PCSITS)” 11 квітня 2025 року, Київ, Україна. С 54-55