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
«____» червня 2023 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань 12 Інформаційні технології
(шифр і назва галузі знань)
спеціальність 125 Кібербезпека
(код і назва спеціальності)
освітній ступень бакалавр
освітня програма Кібербезпека
(назва освітньо-професійної програми)
на тему: «WEB-застосунок на базі архітектури REST з можливістю двофакторної аутентифікації користувача»

Виконавець: студент IV курсу, групи КБ-41

_____ **Андрій СТАРОСЕЛЬСЬКИЙ**
(підпис) (ім'я, прізвище)

	Ім'я, прізвище	**Підпис**
Керівник	Лариса МИРУТЕНКО	

Нормоконтроль	Андрій ФЕСЕНКО	

Київ 2023

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
_____ Сергій ТОЛЮПА
«24» жовтня 2022 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності 125 Кібербезпека
(код і назва спеціальності)

освітньої програми Кібербезпека
(назва освітньої програми)

Студенту **КБ-41** **Старосельському Андрію Олексійовичу**
(група) (прізвище ім'я по-батькові)

Тема кваліфікаційної роботи WEB-застосунок на базі архітектури REST з можливістю двофакторної аутентифікації користувача

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Тема кваліфікаційної роботи затверджена на засіданні кафедри кібербезпеки та захисту інформації протокол №3 від 20.10.2022 р.

2. ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Архітектури web-застосунків, загрози функціонуванню web-застосунків, механізми аутентифікації для web-застосунків, алгоритми генерації одноразових паролів.

3. ЗМІСТ РОЗРАХУНКОВО-ПОЯСНЮВАЛЬНОЇ ЗАПИСКИ

Архітектури web-застосунків та програмного забезпечення, механізми автентифікації для web-застосунків, алгоритми генерації одноразових паролів, загрози безпеці web-застосунків, розробка web-застосунку.

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Практична цінність Реалізація web-застосунку на базі архітектури REST з можливістю використання другого фактору для аутентифікації користувача.

5. ДАТА ВИДАЧІ ЗАВДАННЯ

Дата видачі завдання: 24 жовтня 2022 року

Завдання видав _____ (підпис) Лариса МИРУТЕНКО (ім'я, прізвище)

Завдання прийняв до виконання _____ (підпис) Андрій СТАРОСЕЛЬСЬКИЙ (ім'я, прізвище)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів робіт	Строки виконання робіт (початок-кінець)	Відмітка про виконання
1	Уточнення постановки задачі	24.10.2022 – 20.01.2023	*виконано*
2	Аналіз літератури	28.01.2023 – 10.02.2023	*виконано*
3	Дослідження архітектури web-застосунків	20.02.2023 – 02.03.2023	*виконано*
4	Дослідження принципів архітектури REST	03.03.2023 – 22.03.2023	*виконано*
5	Огляд механізмів аутентифікації для web-застосунків	23.03.2023 – 08.04.2023	*виконано*
6	Дослідження алгоритмів генерації одноразових паролів	09.04.2023 – 14.04.2023	*виконано*
7	Огляд загроз функціонуванню web-застосунків	15.04.2023 – 16.04.2023	*виконано*
8	Побудова архітектури web-застосунку	17.04.2023 – 22.04.2023	*виконано*
9	Реалізація web-застосунку з можливістю двофакторної аутентифікації користувача	23.04.2023 – 14.05.2023	*виконано*
10	Оформлення пояснювальної записки	15.05.2023 – 12.06.2023	*виконано*

Завдання видав _____ (підпис) Лариса МИРУТЕНКО (ім'я, прізвище)

Завдання прийняв до виконання _____ (підпис) Андрій СТАРОСЕЛЬСЬКИЙ (ім'я, прізвище)

Термін подання кваліфікаційної роботи до ЕК 12 червня 2023 року

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел та додатків. Основний текст займає 84 сторінки, включає в себе зміст, вступ, три розділи кваліфікаційної роботи, висновки та список джерел. Крім того, робота містить 2 додатки із загальною кількістю сторінок 6. У пояснювальній записці кваліфікаційної роботи міститься 35 рисунків та 30 літературних джерел.

Метою роботи є розробка web-застосунку на базі архітектури REST з можливістю двофакторної аутентифікації користувача.

Об'єктом дослідження є процес аутентифікації користувача у web-застосунках.

Предметом дослідження є механізм реалізації двофакторної аутентифікації web-застосунку.

Методи дослідження: аналіз відкритих джерел, аналіз методів аутентифікації у web-застосунках, порівняння.

Практичною цінністю є розроблений web-застосунок на базі архітектури REST з можливістю використання другого фактору для аутентифікації користувача.

Ключові слова: двофакторна аутентифікація, web-застосунок, аутентифікація, облікові дані, кібербезпека, захист web-застосунків.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕННЬ

БД	–	База даних
IoT	–	Internet of Things
REST	–	Representational State Transfer
SOAP	–	Simple Object Access Protocol
API	–	Application Program Interface
JWT	–	JSON Web Token
OTP	–	One-Time Password
SQL	–	Structured query language
СУБД	–	Система керування базами даних
IDE	–	Integrated development environment;

ВСТУП

У XXI сторіччі обчислювальна техніка оточує людей у всіх сферах їхнього життя. На ринку представлена величезна кількість смартфонів, ноутбуків, персональних комп’ютерів, ігрових консолей, різноманітних «розумних» пристроїв, таких як холодильники, годинники, телевізори, тощо. Згідно інформації, яку надає компанія Statista, що спеціалізується на ринкових і споживчих даних, кількість одних лише смартфонів, які були під’єднані до мережі стільникових операторів, у 2022 році складала більше 6,5 мільярдів пристроїв [1]. Для порівняння, у 2016 році таких було у два рази менше – 3,6 мільярдів. Як було зазначено, дана статистика розкриває інформацію лише про смартфони, вона не враховує персональні комп’ютери та ноутбуки, історія продажів яких мала свій початок набагато раніше за мобільні пристрої, та пристрої інтернету речей, кількість яких більша за 13 мільярдів [2] із перспективою значного збільшення у найближчі роки.

Сучасна обчислювальна техніка має безліч форм і розмірів, від потужних серверів і ноутбуків до смартфонів та інших різноманітних пристроїв, але впевнено можна стверджувати, що важливою функцією сучасного пристрою для середньостатистичного користувача є можливість підключення до мережі інтернет. Інтернет є одним з найбільших досягнень сучасних технологій, що революціонізувало людський спосіб життя та взаємодії. У 1989 році британський фізик Тім Бернерс-Лі винайшов World Wide Web (WWW), який є одним з найважливіших компонентів інтернету. Він створив систему, що дозволяла зв'язувати гіпертекстові документи з різних комп'ютерів та серверів, надаючи змогу переходити з одного документа на інший за допомогою гіперпосилань. Перші веб-сайти були створені на базі HTML (Hypertext Markup Language) та CSS (Cascading Style Sheets) - стандартних мов для розмітки та оформлення веб-сторінок. Пізніше були розроблені мови програмування для створення динамічних веб-сторінок та web-застосунків, зокрема, PHP, JavaScript, Ruby та інші.

Із появою динамічних та інтерактивних сторінок відбувся значний розвиток інтернету: почали з’являтися веб-сайти та web-застосунки, які люди використовують

у наші дня для споживання контенту, розваг чи роботи. Із збільшенням кількості компонентів веб-сервісів, з’явилася необхідність налагодження їх взаємодії між собою. У цей період почали з’являтися web-API, які і виконували подібні функції. У сучасному інтернеті API є невід’ємною складовою більшої частини web-сервісів. Згідно опитування, проведеного компанією Slashdata у 2021 році, приблизно 90% розробників використовували API у той чи інший спосіб [3]. Велика кількість соціальних мереж, таких як Facebook, TikTok та Instagram надають доступ до своїх API стороннім розробникам, щоб вони могли створювати власні застосунки, які б мали доступ до функціоналу та даних користувачів. Крім соціальних мереж, публічні API мають сервіси із новинами, погодою, дослідницькими даними та інші.

Однак, як і у випадку з будь-якими речами, створеними людиною, web-застосунки мають не лише сильні сторони, а і слабкі. Великою проблемою, із розвитком обчислювальної техніки та інтернету, стали зловмисники. У великій кількості випадків їх ціллю є дані користувачів платформи і, на превеликий жаль, API, зважаючи на їх відкритість, у випадку недосконалості безпеки, є інструментом для компрометації чутливих даних. Особливо чутливою ця проблема є у банківських та фінансових сферах, де циркулює платіжна інформація користувачів. Згідно із дослідженням Salt Security, 94% відсотки опитаних представників різних організацій визнали, що мали проблеми з безпекою API [4]. У аналогічному опитуванні даної організації за 2021 рік, відсоток становив 91, що може свідчити про збільшення інцидентів або визнання проблеми [5].

У згаданому вище опитуванні, про проблеми із вразливостями заявив 41% опитаних, що є найбільшим відсотком та найпоширенішою визнаною проблемою. Однак друге місце за кількістю заяв, займає проблема із аутентифікацією, яка виникла у 40% опитаних. Тож актуальність роботи полягає у розробці web-застосунку, який би надавав користувачам можливість додавання другого фактору для аутентифікації, що дозволило б підвищити безпеку процесу аутентифікації та доступу користувача.

Метою роботи є розробка web-застосунку на базі архітектури REST з можливістю двофакторної аутентифікації користувача.

Для досягнення зазначеної мети кваліфікаційної роботи поставлено наступні завдання:

- дослідити принципи функціонування web-застосунків;
- провести аналіз механізмів захисту web-застосунків;
- побудувати архітектуру web-застосунку із можливістю двофакторної аутентифікації;
- реалізувати web-застосунок із можливістю двофакторної аутентифікації.

Об’єктом дослідження є процес аутентифікації користувача у web-застосунках.

Предметом дослідження є механізм реалізації двофакторної аутентифікації web-застосунку.

Методи дослідження**:** аналіз відкритих джерел, аналіз методів аутентифікації у web-застосунках, порівняння.

Практична цінність роботи полягає в розробці web-застосунку на базі архітектури REST з можливістю використання другого фактору для аутентифікації користувача.

РОЗДІЛ 1
ПРИНЦИПИ ФУНКЦІОНУВАННЯ WEB-ЗАСТОСУНКІВ

1.1 Монолітна та мікросервісна архітектури застосунку

Сфера розробки програмного забезпечення має багату історію, яка охоплює кілька десятиліть, з моменту появи перших комп'ютерів. Програмне забезпечення розвивалася з часом, щоб задовольнити мінливі потреби користувачів, які його використовують. Від перших комп'ютерів до останніх досягнень штучного інтелекту, розробка програмного забезпечення відіграла вирішальну роль у формуванні того, як людина взаємодіє із обчислювальною технікою. Розробка web-додатків є відносно новим напрямком у програмній інженерії. Її історія тісно пов'язана із розвитком інтернет-технологій. Не зважаючи на свою відносну новизну, цей напрямок розробки мав значний вплив на індустрію програмного забезпечення.

Веб-програми — це програми, які дозволяють взаємодіяти із інформацією за допомогою веб-браузера. Доступ до них можна отримати з будь-якої точки світу за наявності підключення до Інтернету. Головною перевагою цього підходу є те, що клієнти не залежать від конкретної операційної системи користувача. Таким чином, web-додатки, певною мірою, відкрили світу поняття відносної крос-платформеності. Завдяки цій універсальній особливості, web-додатки стали дуже популярними в 1990-х і 2000-х роках. Розробникам не потрібно створювати різні версії однієї програми для різних ОС, таких як Microsoft Window, Mac OS чи Linux. Програма створюється лише один раз для будь-якої платформи та може працювати на будь-якій операційній системі, яка дозволяє встановлювати веб-браузер.

На перших етапах розвитку інтернету, web-застосунки мали доволі примітивний функціонал: клієнт, або веб-браузер, отримував код веб-сторінки як статичний документ та відображав його вміст. Працюючи з таким типом сторінки, можливості будь-якої взаємодії були мінімальні. Коли користувач вносив будь-які зміни на веб-сторінці, потрібен був час, щоб оновити цю сторінку, оскільки зміни необхідно було відправити на сервер, дочекатися оновлення сторінки на сервері та

завантажити її знову. Враховуючи значні обмеження у швидкості на початкових етапах розвитку інтернету, такий функціонал вимагав значної кількості часу.

1995 став значущим роком для розвитку інтернету. Компанія Netscape Communications представила JavaScript, мову сценаріїв на стороні клієнта, яка дозволяла розробникам покращувати інтерфейс користувача за допомогою динамічних елементів. JavaScript зробив роботу в інтернеті швидшою і продуктивнішою. Вбудовані сценарії дозволили виконувати різні завдання на конкретних завантажених сторінках «прямо на місці».

У середині 1990-х років веб-браузери стали більш складними, і розробники почали експериментувати зі створенням складніших web-додатків. Одним із перших популярних web-додатків, які змінили життя користувачів, була онлайн-платформа eBay, яка була запущена в 1995 році. Після eBay з'явилися інші популярні веб-сайти, такі як Amazon і Yahoo, які проклали шлях для індустрії електронної комерції.

У 2000-х роках з'явилися соціальні медіа-платформи, такі як Facebook і Twitter, які революціонізували спосіб взаємодії один з одним в Інтернеті. Платформи соціальних мереж дозволяли людям спілкуватися один з одним і обмінюватися інформацією в режимі реального часу, вони проклали шлях для розробки інших типів web-додатків, таких як онлайн-форуми та програми для обміну повідомленнями.

Програми та архітектура програмного забезпечення — це дві взаємопов'язані концепції, які відіграли значну роль у формуванні сучасних інформаційних систем. Із розширенням функціоналу застосунків, виникла необхідність грамотного структурування програм для підвищеннях їх продуктивності, швидкодії та масштабованості. Ідеї архітектури програмного забезпечення почали з'являтися у кінці 1960-х, проте зважаючи на обмежену функціональність комп'ютерних програм та обмеженість можливостей обчислювальної техніки, значної необхідності у специфічних рішеннях не було. Для розробки необхідних рішень, розробникам було достатньо обрати правильні алгоритми та структури даних [6]. Із розширенням можливостей техніки зростала і складність програм, що на певному етапі призвело до підняття проблеми оптимізації коду застосунків. З'явилася необхідність

структурування елементів та коду програми, для ефективної розробки та функціонування.

Архітектура у сфері програмного забезпечення є доволі відносним поняттям. Слово архітектура є похідним від латинського «architector» (творець, будівник), що в свою чергу походить від грецького «αρχιτέκτων» (головний будівник) [7]. Таким чином дане слово передбачає певний зв'язок із будуванням чи побудовою. Архітектура програмного забезпечення передбачає певне структурування коду або елементів програми для побудови робочого застосунку, який виконуватиме необхідні для користувача функції. Але сучасні програми можуть складатися із великої кількості коду, різних елементів, наприклад баз даних, клієнтського інтерфейсу, підпрограм, що дозволяють маніпулювати даними і подібне. Виходячи із цього, важко створити єдину архітектуру, яка б дозволяла вирішити усі проблеми при розробці. На кожному етапі створення програми зазвичай виникають різні архітектурні проблеми для яких необхідно знайти рішення. Архітектура програми дозволяє використовувати різні рішення для кожного рівня абстракції програми. Таким чином важко однозначно виділити категорії програмних архітектур, оскільки вони можуть стосуватися різних аспектів кінцевого продукту чи різних видів програм загалом.

Архітектура програмного забезпечення є важливим фактором під час проектування та створення програм. Одним із підходів до структурування програми є використання монолітної чи мікросервісної архітектури. Монолітна архітектура передбачає створення програми як єдиної автономної одиниці, тоді як мікросервісна архітектура передбачає розбиття програми на набір незалежних, слабко пов'язаних служб. Кожна архітектура має свої сильні та слабкі сторони, і вибір правильної архітектури для конкретної програми залежить від різних факторів, таких як вимоги до масштабованості, складність і ресурси групи розробників. Дані типи архітектур є двома поширеними підходами до створення програмних додатків. Хоча монолітна архітектура була традиційним способом створення програм, архітектура мікросервісів набула популярності в останні роки завдяки своїй здатності забезпечувати масштабованість, стійкість і гнучкість кінцевого продукту. У цьому контексті важливо розуміти відмінності між монолітною та мікросервісною

архітектурою, їхні переваги та недоліки, а також фактори, які впливають на вибір між двома архітектурами.

У монолітній архітектурі всі компоненти та модулі програми міцно пов'язані між собою та взаємодіють через безпосередні виклики функцій або спільні бібліотеки. Монолітний додаток складається з однієї єдиної та нероздільної системи. Зазвичай це рішення включає інтерфейс користувача на стороні клієнта, програму на стороні сервера та базу даних. Всі функції додатка знаходяться в одному місці та керуються звідти. Умовна схема монолітної програми із рівнями наведена на рисунку 1.1. Усі функціональні можливості програми існують в одній кодовій базі.

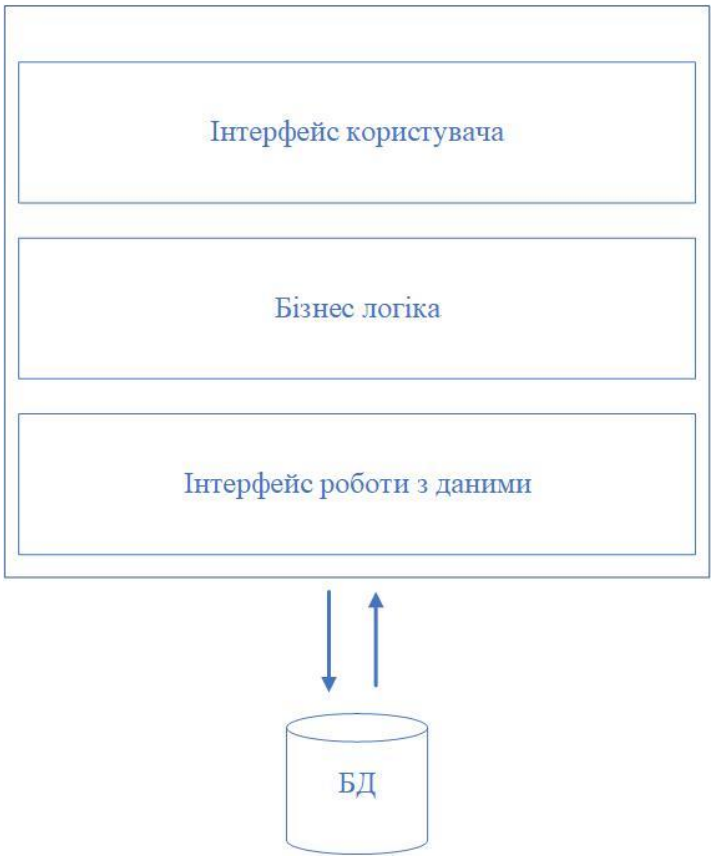

Рисунок 1.1. Схема монолітної архітектури

Монолітна архітектура має переваги у вирішенні наскрізних проблем, налагодженні та тестуванні, розгортанні та розробці. Задачі, такі як журналювання, обробка, кешування та моніторинг продуктивності, легше вирішити в монолітній програмі, оскільки функції програми знаходяться в одному місці. Проводити налагодження та тестування простіше в однорівневій програмі. Розгортання даної

архітектури також є швидшим і легшим, оскільки існує лише один файл або каталог з яким необхідно працювати. Монолітна архітектура — це стандартний спосіб створення додатків, що полегшує розробку додатків кваліфікованим інженерам [8].

Однак, монолітна архітектура має обмеження щодо складності коду, сильно взаємозалежних компонентів, обмеженої масштабованості та способів інтеграції нових технологій. У міру масштабування монолітної програми, її стає надто складно зрозуміти та підтримувати. Сильно взаємозалежні компоненти можуть створити проблеми при впровадженні змін, оскільки будь-яка зміна коду впливає на всю систему. Масштабування обмежене, оскільки компоненти програмного забезпечення не можна розширювати окремо, лише всю програму. Інтеграція нових технологій є проблематичним процесом, оскільки може знадобитися переписування значної кількості коду усієї програми. Крім того монолітна архітектура має обмеження щодо часу запуску та розгортання, може містити складності для нових розробників на проекті, має неефективне горизонтальне масштабування та недостатню надійність, враховуючи, що одна помилка може вивести з ладу всю програму. Зі збільшенням розміру монолітної програми збільшується час її запуску та розгортання. Новим розробникам, які приєднуються до проекту, може бути складно зрозуміти логіку великої монолітної програми, особливо у випадку, коли вона існує протягом довгого періоду часу. Горизонтальне масштабування неможливе, оскільки всю програму потрібно розгортати на кількох серверах, навіть якщо лише одна частина програми зазнає великого навантаження. Нарешті, одна помилка в будь-якому модулі може вивести з ладу всю монолітну програму, зробивши її нефункціональною [9].

У свою чергу мікросервісна архітектура передбачає розбиття програми на незалежні самодостатні служби, які спілкуються один з одним за допомогою протоколів, таких як, наприклад, HTTP. Кожна служба відповідає за певну функцію чи вирішення конкретної задачі, і її можна розробляти та розгортати незалежно. Ця архітектура забезпечує масштабованість, стійкість і гнучкість і підходить для великих і складних програм. Умовна схема архітектури наведена на рисунку 1.2.

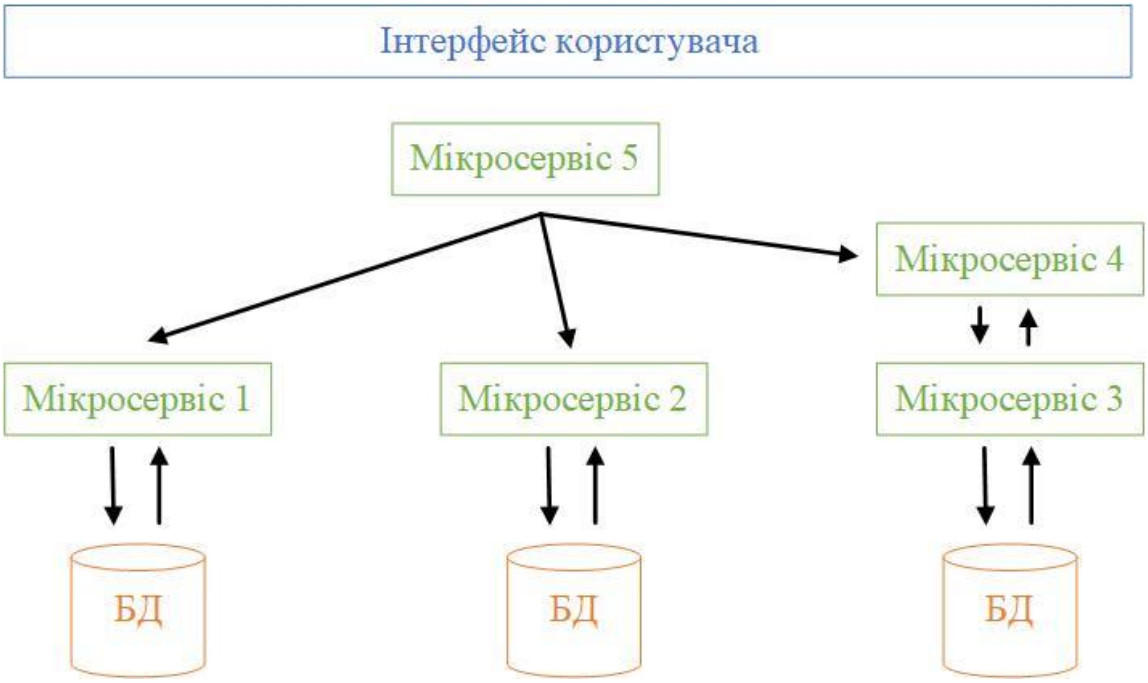

Рисунок 1.2. Схема мікросервісної архітектури

Мікросервіси у даній архітектурі можуть бути як окремими самодостатніми модулями, так і взаємодіяти з іншими мікросервісами для виконання необхідних функцій. Однією із особливостей даного підходу до структурування програм є те, що при правильному налаштуванні та визначенні правил взаємодії, кожен із мікросервісів може бути розроблений із використанням різних мов та технологій програмування.

Існують певні архітектурні принципи, яким повинна відповідати програма побудована з використанням мікросервісів. Принцип єдиної відповідальності, де кожен мікросервіс відповідає лише за свій функціонал є одним із них. Кількість розроблених мікросервісів має дорівнювати кількості необхідних функцій. Інший принцип полягає у тому, що архітектура має бути побудована з урахуванням бізнес-можливостей і дозволяти використовувати технологічні стеки, які найбільше підходять для вирішення бізнес-цілей. Крім того, мікросервіси мають бути розроблені з урахуванням можливих збоїв у роботі і враховуючи особливості архітектури, збій одного мікросервісу не повинен впливати на всю систему. Такий підхід дозоляє забезпечити роботу інших частин програми навіть у випадку відмови одного із модулів [9].

Переваги архітектури мікросервісів полягають у тому, що елементами програми легше керувати, оскільки кожна частина відповідає лише за свій функціонал, і при необхідності проведення оновлення в одному з мікросервісів, повторно розгорнути буде потрібно лише цей модуль. Мікросервіси є автономними частина, а отже, розгортаються незалежно один від одного, тому час їх запуску та розгортання відносно менший. Новим розробникам на проекті легше ввійти в процес роботи, оскільки їм потрібно ознайомитися та зрозуміти лише окремий мікросервіс, який забезпечує функціональність, над якою вони працюватимуть, а не всю систему. Якщо певна частина застосунку стикається з великим навантаженням через надмірне використання з боку користувачів, достатньо масштабувати лише один проблемний мікросервіс.

Недоліки мікросервісної архітектури полягають у тому, що вона складна і вимагає наявності певних кваліфікацій у розробників для налаштування та керування зв’язком між сервісами. Незалежне розгортання мікросервісів є складнішим процесом ніж розгортання програми, побудованої з використанням монолітної архітектури. Підтримка мікросервісів дорога з точки зору використання мережі та ресурсів. Крім того, наявність окремих модулів ускладнює проведення тестів та контролю над роботою системи.

Рішення про вибір між монолітною та мікросервісною архітектурою слід приймати на основі конкретних потреб і цілей проекту. Монолітна архітектура підходить для невеликих програм з меншою функціональністю, тоді як інший варіант більше підходить для великих проектів, які вимагають масштабованості, гнучкості та незалежності сервісів. У той час як монолітна архітектура має такі переваги, як простота розробки, тестування та розгортання, архітектура мікросервісів пропонує такі переваги, як краща масштабованість, відмовостійкість і покращена продуктивність. Однак впровадження архітектури мікросервісів вимагає досвідчених розробників, які мають навички керування складними розподіленими системами та ефективного зв’язку між мікросервісами. Вибір архітектури слід здійснювати з урахуванням потреб проекту, досвіду команди та наявних ресурсів.

1.2 Види архітектури web-застосунків

У розробці web-застосунків, як і у розробці застосунків для мобільних пристрої та інших типів програм, які містять певний функціонал, пов'язаний із мережею інтернет, велике значення відіграє поняття web-API. Абревіатура API з англійської мови перекладається як інтерфейс програмування застосунків. Узагальнено це набір протоколів, інструментів і стандартів для створення програмних додатків. API забезпечують зв’язок між різними програмами або елементами однієї програми, дозволяючи їм обмінюватися даними та функціями. Метою API є надання розробникам стандартизованого способу доступу та маніпулювання даними та функціями, які надає програмне забезпечення шляхом абстрагування імплементації певного функціоналу та надання доступу лише для необхідних для роботи механізмів. Загалом, поняття API використовується у різних сферах інформаційних технологій. До інтерфейсів програмування застосунків можуть відноситися різні бібліотеки мов програмування, функції, які спрощують роботу с операційною системою або так звані web-API, які пов’язані із веб-розробкою і так далі.

Під web-API у більшості випадків розуміється концепція використання HTTP-запитів для надання доступу до даних у певному форматі, зазвичай JSON або XML. Даний вид API почав своє існування невдовзі після появи Інтернету. Однією із перших сфер, де вони використовувалися стала електронна комерція [10]. Лідери ринку, такі як Salesforce, eBay та Amazon, створили API, які дозволяли розробникам інтегрувати елементи їх платформ на власні веб-сайти. Наступним етапом розвитку стала поява соціальних мереж. Компанії-власники створювали API для полегшення роботи зі своїми платформами, отримання даних користувачів та інтеграції у сторонні продукти. Справжній розквіт web-API почався після представлення перших смартфонів. Мобільна еволюція інтернету призвела до становлення API у тому вигляді, у якому вони використовуються і зараз. Смартфони значно розширили можливий функціонал мобільних пристроїв, що, у свою чергу, призвело до розробки мобільних програм на основі API. Ця еволюція вплинула не лише на мобільні телефони, але й відкрила нові можливості використання API для будь-якого

електронного пристрою, який можна під’єднати до Інтернету, революціонізувавши спосіб створення, передачі, зберігання та обміну даними в інтернеті. Крім того, web-API знайшли своє місце і у сфері IoT. Вони дозволяють пристроям спілкуватися один з одним через Інтернет. API дають змогу пристроям інтернету речей обмінюватися даними та взаємодіяти з іншими пристроями та web-додатками. Це дозволяє розробникам створювати інноваційні та інтегровані IoT-рішення, які можуть підвищити функціональність та ефективність широкого спектру галузей, від охорони здоров’я до виробництва та транспорту. Веб-інтерфейси відіграють вирішальну роль у розкритті потенціалу IoT, забезпечуючи інтеграцію та зв’язок між пристроями та системами.

Класифікація API може бути різною: з точки зору користувача вони можуть бути розподілені за функціональністю чи документацією; з точки зору розробника вони можуть бути розподілені за режимом доступності. Загалом, можна виділити три типи доступу до API [11]:

- Внутрішні API, які використовуються певною організацією, її відділом, командою розробників у межах роботи над певним продуктом.
- Зовнішні API, також відомі як публічні чи відкриті, які є загальнодоступними для усіх користувачів.
- Партнерські API, доступ до яких мають лише всередині організації та певні сторонні партнери.

Проте головною особливістю, за якою можна розподілити API, є архітектура. Архітектура є важливим елементом, оскільки вона забезпечує стандартизацію підходу до проектування, розробки та розгортання API, що полегшує розробникам створення та інтеграцію програм, які їх використовують. Це гарантує, що API є масштабованими, надійними та придатними для обслуговування, а також що вони можуть обробляти інтенсивний трафік і підтримувати необхідну кількість клієнтських пристроїв і програм. Добре продумана архітектура API також сприяє безпеці, завдяки використанню стандартних протоколів і механізмів автентифікації, а також забезпечує взаємодію та інтеграцію з іншими системами та службами.

Прийнявши узгоджену архітектуру API, організації можуть скоротити час і витрати на розробку, підвищити продуктивність і покращити взаємодію з користувачем.

Щорічно компанія Postman, яка розроблює інструменти, завдяки яким розробники можуть створювати, тестувати та поширювати власні API, проводить опитування про стан речей у індустрії. Згідно інформації, яка була отримана за 2022 рік, за частотою використання розробниками серед архітектур перше місце займає REST із результатом 89%, друге – webhooks із результатом 35%, третє – SOAP – 34%, а четверте – GraphQL із результатом 28%. Крім даних архітектур до рейтингу потрапили WebSocket, gRPC, MQTT, AMQP, Server-sent event, EDI та EDA, відсоток використання яких зменшується у порядку переліку [12]. Нижче буде розглянуто деякі із цих архітектур.

REST — це найпопулярніший архітектурний стиль. Він має структуру клієнт-сервер, прості й уніфіковані інтерфейси для зв’язку між системами, операції без збереження стану та інші керівні принципи.

Webhooks, також відомі як «зворотні API», засновані на подіях і дозволяють автоматизовано надсилати повідомленням з однієї системи в іншу щоразу, коли відбувається певна подія. Цю концепцію можна використовувати для перевірки змін у даних.

SOAP є більш структурованим і формалізованим архітектурним стилем для API. Однак рішення з архітектурою SOAP можуть працювати повільніше, ніж ті, які побудовані з використанням інших підходів [11]. SOAP API використовують протокол обміну повідомленнями на основі XML із тегами Envelope, Header і Body відповідно до вимог кінцевої точки.

До сильних сторін SOAP можна віднести наступне: вбудований функціонал SOAP для веб-сервісів дозволяє обробляти відповіді незалежно від мови та платформи; сумісність із кількома транспортними протоколами (SOAP може використовувати різні протоколи передачі для різних сценаріїв); вбудована обробка помилок (специфікація SOAP API дозволяє повертати повідомлення Retry XML із кодами помилок і поясненнями); кілька розширень безпеки (SOAP інтегрований з протоколами WS-Security, що робить його придатним для транзакцій корпоративного

рівня. Він забезпечує конфіденційність і цілісність транзакцій, дозволяючи шифрування на рівні повідомлень) [13].

GraphQL, що розшифровується як Graph Query Language, спочатку був розроблений компанією Facebook як інструмент для внутрішнього використання, але пізніше був опублікований як мова для API з відкритим кодом. На відміну від інших типів архітектур, які залежать від того, які кінцеві точки визначені на сервері, GraphQL може відправляти запит на певну інформацію серверу на пряму. У 2015 році технологія стала доступною усім охочім, а згодом керування розробкою та розвитком GraphQL було передано організації GraphQL Foundation, яка займається цим і по цей день [14].

Однією з головних переваг GraphQL є його гнучкість. Він не залежить від мови, тому його можна використовувати з будь-якою мовою програмування та інтегрувати з будь-якою базою даних або API. Це полегшує адаптацію та інтеграцію в існуючі системи. Крім того, оскільки клієнти можуть точно вказати, які дані їм потрібні, розробникам інтерфейсів користувача стає легше працювати незалежно від розробників серверної частини проекту і навпаки.

Потенційною проблемою GraphQL є те, що при неправильному використанні можуть виникнути проблеми з продуктивністю. У випадку, коли клієнт надсилає запит, який має повернути забагато даних, може виникнути навантаження на сервер, що в свою чергу може призвести до сповільнення часу відповіді або виходу із ладу [15].

WebSocket API використовує протокол зв’язку з комп’ютером WebSocket, створюючи повнодуплексний канал зв’язку, використовуючи одне TCP-з’єднання. WebSocket API надають серверам визначений спосіб надсилання інформації та даних клієнтам, навіть якщо клієнт не запитує дані. WebSocket API також дозволяють обмінюватися даними між клієнтами та серверами, зберігаючи з’єднання відкритими, оскільки працює за принципом збереження стану.

gRPC API використовують протокол gRPC, або Remote Procedure Call, створений компанією Google. Він дозволяє дозволяє клієнту звертатися до сервера так само, як до локального об’єкта, що полегшує взаємодію між розподіленими

програмами та системами. Для структурування даних використовується формат protocol buffers [16]. gRPC підтримує різний функціонал, такий як балансування навантаження, трасування, перевірка справності та автентифікація [13].

Server-sent event API, також відомий як SSE, дозволяє клієнту автоматично отримувати оновлення, через HTTP-з'єднання. Ця технологія базується на даних, які надсилаються із сервера.

AMQP API, що означає Advanced Message Queuing Protocol, є протоколом, який відповідає відкритим стандартам і працює на прикладному рівні. AMQP найкраще підходить для проміжного програмного забезпечення, що дозволяє відправникам повідомлень і клієнтам спілкуватися один з одним. AMQP підтримує надійність і безпеку повідомлень завдяки здатності ставити повідомлення в чергу та маршрутизувати їх.

MQTT, що розшифровується як Message Queuing Telemetry Transport, це протокол, який добре підходить для IoT, завдяки своєму легкому підходу до обміну повідомленнями. Пристрої можуть публікувати або підписуватися на повідомлення за допомогою так званого MQTT-брокера, який виступає у якості об'єднуючого вузла.

EDI, або Electronic Data Interchange, існує з 1970-х років. EDI дозволяє компаніям спілкуватися в електронному вигляді, передаючи інформацію, яка зазвичай записується на папері, наприклад квитанції чи рахунки-фактури, або інформацію про замовлення, наприклад на купівлю.

Підсумовуючи, архітектури API необхідні для забезпечення зв'язку та обміну даними між різними системами та програмами. Різні типи архітектур API, як-от REST, Webhooks, SOAP, GraphQL, WebSocket, gRPC, Server-sent events, AMQP, MQTT і EDI, мають власні сильні та слабкі сторони та підходять для різних випадків використання та сценаріїв. Оскільки технології продовжують розвиватися, нові архітектури API з'являються та набувають популярності, що показує згадане вище дослідження компанії Postman, яке відзначає зниження відсотку використання класичної для подібних рішень архітектури REST у порівнянні із попередніми роками [12]. Вибір правильної архітектури має вирішальне значення для досягнення бажаної продуктивності, надійності та масштабованості системи та може значно вплинути на

успіх кінцевого продукту. Для розробників і компаній важливо бути в курсі останніх архітектур API та їхніх можливостей, щоб приймати обґрунтовані рішення та створювати надійні та ефективні системи.

1.3 Принципи архітектури REST

REST є однією з найпопулярніших архітектур API, які використовуються сьогодні, 89% відсотків розробників використовували дану технологію у 2022 році [12]. Це простий, але потужний архітектурний стиль, який набув широкого поширення завдяки своїй масштабованості та гнучкості.

До створення і поширення архітектури REST, для роботи із API використовувався протокол SOAP. Не дивлячись на наявність слова «простий» у перекладі розшифрування даної абревіатури, даний протокол був доволі складний. У розробників часто виникали проблеми із створенням, використанням та виявленням проблем при роботі із ним. У деяких випадках, розмір документації для роботи із подібними API міг складати понад 400 сторінок [17], що звісно не робило їх використання тривіальною задачею.

Концепція REST була вперше представлена в дисертації, написаній Роєм Філдінгом у 2000 році. Філдінг був одним із головних авторів протоколу HTTP, який є основою всесвітньої павутини. У своїй дисертації Філдінг запропонував новий архітектурний стиль для розподілених гіпермедійних систем, який він назвав Representational State Transfer. У своїй основі REST базується на кількох простих принципах. По-перше, він використовує архітектуру клієнт-сервер, де клієнт і сервер є окремими та незалежними один від одного. По-друге, він ґрунтується на моделі зв’язку без стану, де кожен запит від клієнта містить усю інформацію, необхідну серверу для обробки запиту. Нарешті, він використовує єдиний інтерфейс, що означає, що клієнт і сервер використовують однаковий набір методів для спілкування один з одним. Одним із ключових понять REST є ресурс, який є будь-якою інформацією, яку можна ідентифікувати за допомогою URL-адреси. У RESTful API ресурси є основним способом взаємодії клієнтів із сервером. Ресурси можуть бути

будь-якими, від облікового запису користувача до певної частини даних, і кожен ресурс ідентифікується унікальною URL-адресою. Ще одна важлива концепція в REST — представлення, яке є певним форматом або представленням ресурсу. Наприклад, ресурс може бути представлений у форматі JSON, XML або HTML, залежно від потреб клієнта. Повідомлення також можуть включати метадані, які надають додаткову інформацію про ресурс, таку як дата його створення або автор.

З моменту появи в 2000 році REST зазнав кілька покращень. Одна з найбільших змін відбулася з появою HATEOAS (Hypermedia As The Engine Of Application State), що є розширенням архітектури REST, яке дозволяє клієнтам динамічно знаходити ресурси та переміщуватися ними. Іншою важливою подією в еволюції REST стало впровадження специфікації OpenAPI, яка є стандартизованим способом опису RESTful API. Специфікація OpenAPI забезпечує загальну мову для опису ресурсів, операцій і параметрів RESTful API, що полегшує клієнтам розуміння та взаємодію з API.

У даному підрозділі вже згадувалися основні принципи архітектури REST, але для кращого розуміння важливо розглянути їх детальніше. Архітектура REST не визначає конкретних стандартів для створення застосунків, але все ж має певні загальні принципи, використання яких є важливим при створенні RESTful застосунків [18]. Перш за все, як вже було згадано, RESTful застосунки використовують клієнт-серверну архітектуру. Умовна схема даної архітектури зображена на рисунку 1.3.

У даній архітектурі, клієнт, який може бути комп’ютерною програмою або пристроєм, запитує служби або ресурси у сервера. Сервер опрацьовує запит, надає запитані послуги або ресурси та надсилає їх назад клієнту. Ця модель передбачає розподілену обробку, оскільки клієнт і сервер можуть бути розташовані на різних машинах і спілкуватися через мережу. Архітектура клієнт-сервер зазвичай використовується у web-додатках, де клієнтом є веб-браузер або програма, а сервером є веб-сервер. Наступним важливим принципом є використання так званого «уніфікованого інтерфейсу».

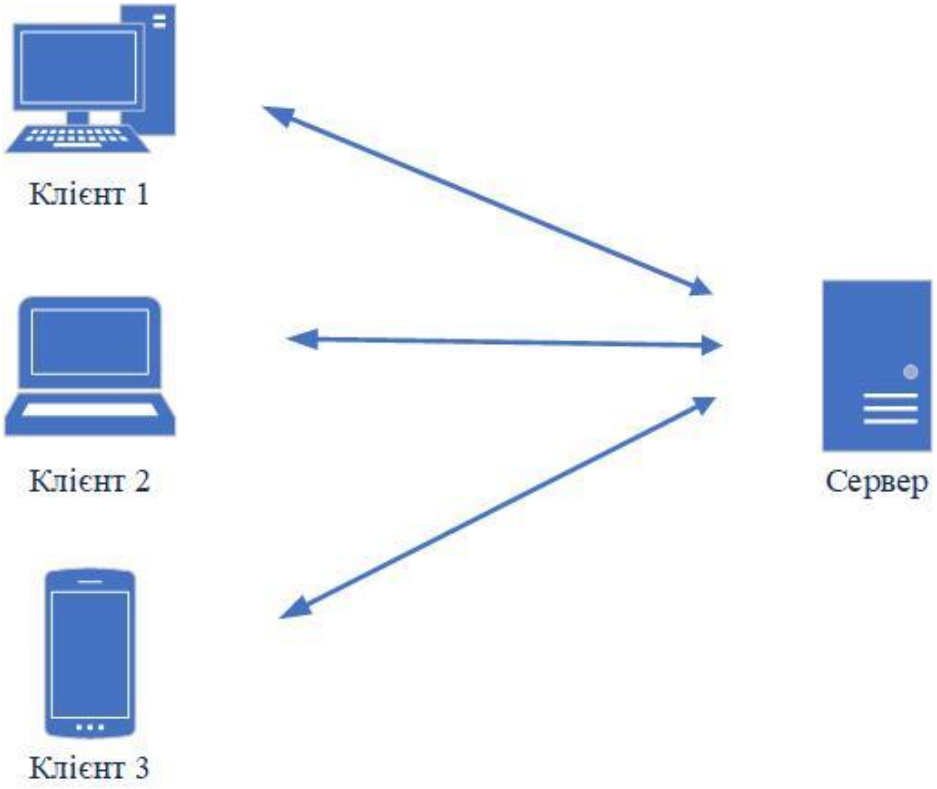

Рисунок 1.3. Схема клієнт-серверної архітектури

Уніфікований інтерфейс забезпечує послідовний і єдиний спосіб взаємодії з сервером і його ресурсами незалежно від клієнтської програми. Цей принцип включає [19]:

- ідентифікацію усіх ресурсів за допомогою URI;
- маніпулювання ресурсами за допомогою представлень, під чим розуміється, що маючи представлення ресурсу та будь-які пов’язані з ним метадані, клієнт повинен мати можливість змінювати або видаляти ресурс;
- повідомлення з самоописом. Представлення ресурсу має бути повернуто клієнту разом з інформацією про те, як його обробляти;
- використання гіпермедіа як механізму стану програми (HATEOAS). Повідомлення між клієнтом і сервером мають містити посилання для відповіді, щоб клієнтська програма могла легко знаходити інші ресурси.

Наступним принципом є підтримка безстанового зв’язку, англійською «stateless». Щоб підпадати під визначення «stateless», сервер повинен мати можливість обробляти запит, не покладаючись на інформацію з попередніх запитів

або не зберігаючи пов’язані з сеансом дані. Таким чином, вся необхідна інформація для виконання запиту повинна бути включена в запит клієнта.

Підтримка кешування – принцип, за яким веб-сервіс має вказувати, чи можна кешувати відповідь, а також тривалість, протягом якої відповідь може кешуватися на стороні клієнта. Кешування дозволяє мінімізувати взаємодію між клієнтом і сервером, тим самим покращуючи загальну продуктивність програми.

Принцип багаторівневості зазначає, що серверна частина може бути представлена різними серверами, які можуть виконувати різні функції. Наприклад, один сервер може містити API, другий може зберігати дані, а третій – автентифікувати запити. Проміжні сервери можуть взаємодіяти з клієнтом без відома головного сервера. Даний підхід може покращити доступність системи за рахунок балансування навантаження та спільних кешів.

Останнім та необов’язковим принципом є використання коду на вимогу (Code on Demand), згідно з яким сервери мають можливість поширювати виконуваний код клієнтам. Типовим прикладом є веб-служби, які відправляють скрипти, наприклад написані на javascript, клієнтам.

Зазначені вище принципи описують загальні ідеї, які мають виконуватися для реалізації сервісу на базі архітектури REST, але існують також і певні ключові елементи, які використовуються для функціонування web-застосунку та його взаємодії із клієнтами. Першим та основним елементом є ресурс. У архітектурі REST під ресурсом розуміється якась форма даних. Крім того, тип представлення ресурсу повинен бути створений для ідентифікації ресурсів у системі. Загальним представленням є ID. Прикладом такої ідентифікації може бути ідентифікатор клієнта, який унікально ідентифікує його в базі даних [18]. Наступним важливим елементом є кінцева точка. Як правило, веб-служба або сервер має певну URL-адресу, це точка входу до ресурсу на сервері. Клієнтська програма може надсилати запити використовуючи ці URL-адреси.

RESTful застосунки побудовані на базі HTTP і використовують його елементи для взаємодії із клієнтами. Такими елементами є методи запитів HTTP, наприклад GET, POST, PUT чи DELETE. Вони використовуються клієнтами для відправлення

запитів на сервер, деякі передбачають у собі наявність певного тіла запиту, інші – ні. Останнім ключовим елементом, який також є частиною HTTP, є статус-коди, такі як 200, 404 і подібні.

Підсумовуючи, принципи архітектури REST забезпечують набір вказівок, які роблять API більш масштабованими, гнучкими та ефективними. Використовуючи ідентифікацію на основі ресурсів, уніфіковані інтерфейси, безстановий зв'язок, кешування та багатошарову систему, розробники можуть створювати API, які легко зрозуміти та використовувати будь-якою клієнтською програмою. Дотримання цих принципів також може покращити загальну продуктивність і масштабованість системи, спрощуючи додавання нових ресурсів і функцій, не впливаючи на існуючу функціональність. Розуміючи та застосовуючи ці принципи, розробники можуть створювати надійні RESTful API, які відповідають потребам їхніх користувачів і клієнтів.

Висновки за розділом 1

У даному розділі було підняте питання архітектури у контексті розробки застосунків та web-застосунків зокрема. Було розглянуто монолітну та мікросервісну архітектури, які використовуються у розробці програмного забезпечення.

Мікросервісна і монолітна архітектури являють собою два протилежні підходи до розробки програмного забезпечення. У монолітній архітектурі вся програма побудована як єдине ціле, де всі компоненти тісно з'єднані між собою. З іншого боку, мікросервісна архітектура — це підхід, при якому додаток розбивається на невеликі незалежні сервіси, які спілкуються один з одним за допомогою API.

У монолітній архітектурі весь код міститься в одній кодовій базі, що полегшує розробку та тестування програми. Однак із збільшенням розміру програми її стає важче підтримувати та масштабувати. Монолітні програми, як правило, менш гнучкі, ніж програми на основі мікросервісів.

Мікросервісна архітектура, з іншого боку, забезпечує більшу гнучкість, масштабованість і стійкість. Оскільки кожна служба є відносно самостійною та не

залежить від інших, розробники можуть вносити зміни в окремі сервіси, не впливаючи на всю програму. Крім того, мікросервісна архітектура дозволяє ефективніше використовувати ресурси, оскільки кожен сервіс можна розгортати незалежно на окремих серверах.

Однією з головних переваг мікросервісної архітектури є те, що вона дозволяє краще ізолювати помилки. У монолітній архітектурі, вихід з ладу одного компонента може призвести до збою у функціонуванні усієї програми. Якщо ж помилка виникає у мікросервісі, за рахунок розподіленості, решта програми може продовжувати функціонувати.

Однак мікросервісна архітектура має власний набір проблем. Однією з головних є збільшення складності управління та координації між елементами програми. Крім того, даний тип архітектури потребує більше зусиль із проектування та тестування, а також збільшення операційних витрат щодо розгортання, моніторингу та обслуговування.

Ще одна головна відмінність між мікросервісною і монолітною архітектурою полягає у підході до керування даними. У монолітній архітектурі всі дані зберігаються в одній базі даних, тоді як при використанні мікросервісів, кожна служба може мати власне сховище даних. Такий підхід може призвести до складності в управлінні даними.

Загалом, вибір між мікросервісами і монолітною архітектурою залежить від конкретних потреб програми. Монолітна архітектура є загалом простіша, її легше розробляти та тестувати, що робить її гарним вибором для невеликих програм. Мікросервіси, з іншого боку, пропонують більшу гнучкість, масштабованість і стійкість, що робить їх гарним вибором для великих і складних програм, які потребують високої доступності та відмовостійкості.

Архітектури API розвивалися протягом багатьох років, щоб задовольнити зростаючі вимоги розробників і бізнесу. Однією з найпопулярніших архітектур, які використовуються сьогодні, є REST.

REST — це легкий, гнучкий і популярний архітектурний стиль, який використовує методи HTTP для виконання операцій CRUD (створення, читання,

оновлення, видалення) над ресурсами. Він орієнтований на ресурси та може використовувати різні формати для обміну даними. REST простий у реалізації і масштабований, що робить його ідеальним для web-додатків.

SOAP — це протокол на основі XML, який забезпечує інтерфейс для обміну даними між веб-службами. Він має складну структуру повідомлень і підтримує обмін повідомленнями як зі, так і без збереження стану. SOAP забезпечує вбудовану обробку помилок і розширення безпеки, що робить його надійним для використання у транзакціях корпоративного рівня.

RPC — це специфікація, яка дозволяє віддалене виконання функції в іншому контексті. RPC має зрозумілу та просту взаємодію, функції, які легко додавати, і високу продуктивність. Однак він може бути перенасичений функціями та мати тісний зв’язок із основною системою.

GraphQL — це мова запитів і середовище виконання для API, що забезпечує гнучкий і ефективний підхід до запитів, оновлення та підписки на дані. Він використовує одну кінцеву точку для обробки всіх запитів, дозволяючи клієнтам запитувати лише ті дані, які їм потрібні. GraphQL пропонує високий ступінь гнучкості та управління версіями, але потребує більше зусиль для реалізації ніж REST.

Таким чином, існують різні архітектури API, і кожна з них має свої переваги та недоліки. REST залишається найпопулярнішою архітектурою завдяки своїй простоті, масштабованості та сумісності з різними мовами програмування та платформами. Однак інші архітектури, такі як GraphQL, SOAP і RPC, також мають свої унікальні функції, які роблять їх корисними для виконання конкретних задач.

Вибір правильної архітектури API залежить від конкретних потреб і вимог програми або системи. Важливо оцінити переваги та недоліки кожної архітектури та вибрати ту, яка відповідає цілям проекту, потребам користувачів і бізнес-цілям.

РОЗДІЛ 2
МЕХАНІЗМИ ЗАХИСТУ WEB-ЗАСТОСУНКІВ

2.1 Огляд механізмів автентифікації для web-застосунків

У повсякденному житті кожна людина стикається із процесом підтвердження особистості. Більша частина населення планети має видані офіційними органами документи, такі як наприклад внутрішній чи закордонний паспорт, водійське посвідчення тощо. Багато компаній мають у офісах охорону чи певні технічні рішення, які відповідають за ідентифікацію та допуск персоналу до місця роботи. Сфера інформаційних технологій не є виключенням. Із розвитком технологій, люди почали зберігати важливу інформацію на обчислювальних пристроях, що в свою чергу призвело до виникнення проблеми забезпечення конфіденційності та цілісності цих даних.

Загалом, процес отримання доступу до певного об’єкту представляє собою три етапи: ідентифікація, аутентифікація та авторизація. Ідентифікація – це процес, який полягає у наданні ідентифікатора особи, предмета чи процесу. Аутентифікація – процес підтвердження відповідності ідентифікатора та суб’єкта, який його надав. Авторизація, у свою ж чергу, відповідає за надання доступу до ресурсів, які доступні суб’єкту, який пройшов аутентифікацію.

Існує три основні форми підтвердження особи: те, що людина знає; те, що людина має; те, чим людина є [20]. Перша форма може бути представлена паролем, секретним питанням, певними видами унікальних кодів і подібним. Друга форма – певними предметами, такими як ключі чи карти. Остання форма зазвичай включає в себе певні біометричні дані, такі як відбитки пальців, скан обличчя чи рогівки ока.

Розглядаючи дані форми із точки зору безпеки, можна стверджувати, що їх надійність буде збільшуватися у відповідному порядку. Найкращим способом захисту безперервно є біометричні дані, оскільки цей тип даних неможливо викрасти, а людина завжди має їх при собі. Предмети, якими володіє людина необхідно носити з собою і мати на увазі, що будь-який предмет може бути викрадений чи загублений. Знання людини є найбільш уразливими. Вони можуть бути скомпрометовані без

фізичної присутності зловмисника через необережність власника або злам інформаційної системи і подібне.

У контексті web-застосунків теоретично можна використовувати усі три підходи, але тут слід врахувати той факт, що для аутентифікації за допомогою предметів чи біометрики, користувачу необхідно мати спеціальне обладнання. Якщо говорити про державні чи банківські системи, де безпека є одним із ключових факторів, можна допустити наявність у користувачів токенів, які вони мають отримати у спеціальних установах, чи біометричних сканерів. Але, якщо розглядати абсолютну більшість web-застосунків у мережі інтернет, то безперечно можна стверджувати, що вони не належать до банківської чи державної сфери, а отже і повинні будувати свої системи із урахуванням наявних у користувачів засобів. Таким чином, знання людини, є ледве не єдиним способом підтвердити її особу.

Для аутентифікації користувача у web-застосунках використовуються різні підходи, основними з яких є: базова аутентифікація, аутентифікація із використанням сесій, дайджест аутентифікація, аутентифікація із використанням токенів та протокол OpenID [21].

2.1.1 Базова аутентифікація

Базова аутентифікація є одним із перших підходів до підтвердження особистості користувача у web-застосунках. Даний метод вбудований у протокол HTTP та полягає у відправленні необхідних даних у заголовку «Authorization» запиту.

Дані мають бути відправлені у вигляді «Authorization: Basic <credentials>», де «credentials» представляють собою закодовані у форматі base64 ім’я користувача та пароль, попередньо зібрані у рядок виду «username:password». Схема базової аутентифікації представлена на рисунку 2.1.

Клієнт робить неавторизований запит на отримання певних ресурсів із обмеженим доступом. Не отримавши даних у заголовку «Authorization», сервер повертає клієнту повідомлення із кодом помилки 401 (не авторизовано) та заголовок «WWW- Authenticate: Basic».

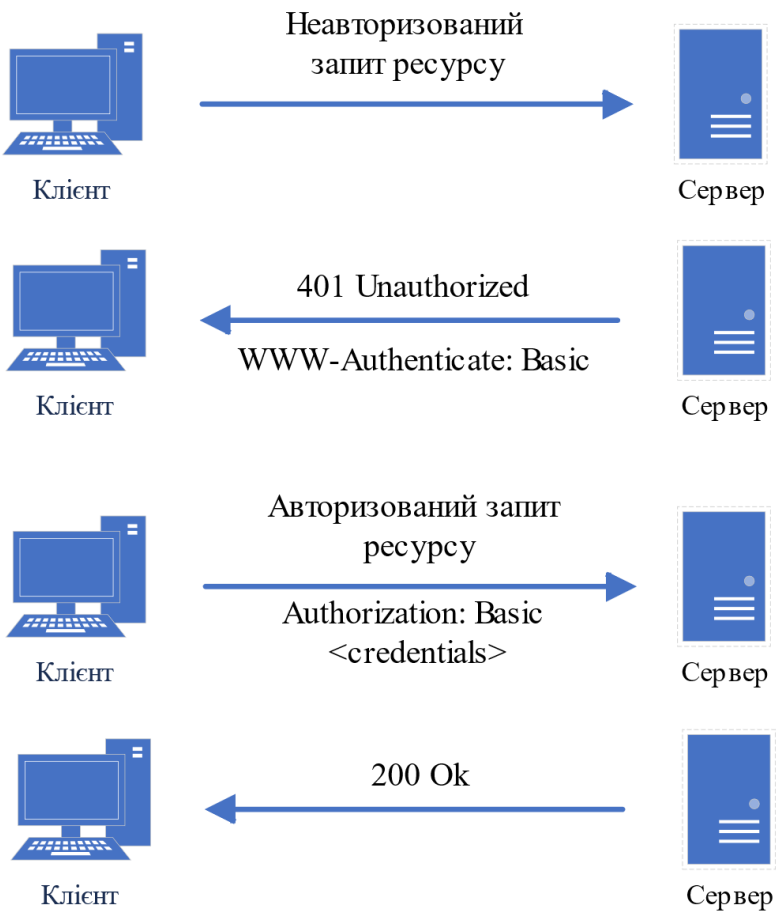

Рисунок 2.1. Схема базової аутентифікації

Після введення даних користувачем, відправляється новий запит із заголовком «Authorization» та вмістом «Basic <credentials>». Отримавши дані, сервер перевіряє їх відповідність та у разі позитивного результату повертає відповідь.

Головною перевагу даного підходу є простота реалізації. Більшість наявних засобів розробки підтримують кодування base64 та дозволяють відправляти дані у заголовках запитів. Крім того, базова аутентифікація інтегрована у протокол HTTP та підтримується, як серверами, так і клієнтами.

Проте недоліки даного методу набагато суттєвіші. Головним із них є безпека. Base64 являє собою стандарт кодування, а не шифрування, тому будь-хто отримавши дані може їх розкодувати та переглянути. Сучасні шифровані з'єднання із використанням SSL/TLS частково вирішують проблему, але постійне відправлення даних через мережу вважається поганим підходом. З цього випливає інший недолік базової автентифікації – необхідність у постійному відправленні даних для

авторизації. Даний підхід працює без збереження стану підключення, дані про сеанс не зберігаються на сервері, тому клієнт повинен кожного разу відправляти їх на сервер. Крім того, даний підхід унеможливлює вихід користувача із системи і хоча різні рішення для вирішення даної проблеми існують, вони не відносяться до базової аутентифікації.

2.1.2 Аутентифікація із використанням сесій

Автентифікація із використанням сесій (або аутентифікація із використанням cookie) є поширеним методом керування автентифікацією користувачів у web-додатках. При використанні даного методу, після аутентифікації користувача, на стороні сервера створюється так звана сесія, яка потім пов’язується з браузером користувача за допомогою файлів cookie.

У даному підході сервер зберігає інформацію користувача та сесії, тому його можна вважати таким, що використовує принцип збереження стану. Кожного разу, коли клієнт відправляє запит, сервер має знайти відповідний файл сесії у сховищі, перевірити його відповідність користувачу, який його відправив та авторизувати дію.

Файли сесії зазвичай вміщують id сесії, інформацію про користувача та браузер, додаткові токени і подібну інформацію. Загалом, насичення файлу може відрізнятися відповідно до вимог застосунку, головна ідея полягає у можливості встановити відповідність між користувачем та id сесії.

Схема процесу аутентифікації із використанням сесій представлена на рисунку 2.2. Клієнт для входу до системи відправляє запит, який містить необхідну інформацію, наприклад логін та пароль. Сервер перевіряє надані дані та у разі успіху створює файл сесії, який містить id сесії та іншу необхідну інформацію. Даний файл зберігається у базі даних. У відповідь на свій запит клієнт отримує файл cookie, який містить id сесії, ідентичний тому, який зберігається у файлі сесії.

Наступні запити клієнт відправляє разом із cookie, який містить id сесії. Отримавши запит, сервер шукає у базі даних файл сесії, який містить наданий клієнтом id сесії.

Якщо пошук проходить вдало і сесія знаходиться, сервер надає доступ до ресурсів, що запитувалися.

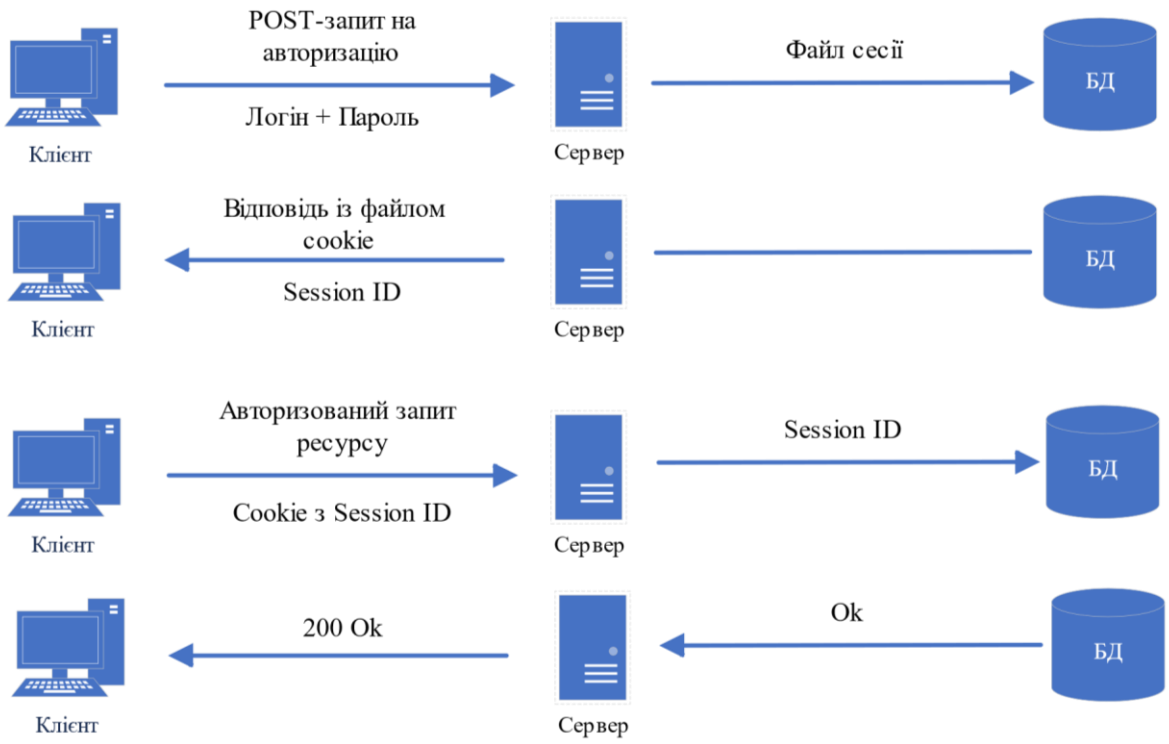

Рисунок 2.2. Схема аутентифікації із використанням сесій

Одна із головних переваг аутентифікації із використанням сесій полягає в тому, що даний підхід дозволяє точно контролювати доступ користувачів до ресурсів. Наприклад, сесія може бути визнана недійсною, якщо користувач вийшов із системи або якщо його сеанс закінчився через бездіяльність. Крім того, користувачу достатньо передати свій логін та пароль лише при першому запиті, надалі після проходження аутентифікації, для авторизації використовується cookie.

Однак даний метод також має деякі недоліки. Одна з головних проблем полягає в тому, що існує можливість викрадення сесії та атак типу міжсайтової підробки запиту. Викрадення сеансу відбувається, коли зловмисник отримує доступ до ідентифікатора сеансу користувача, що дозволяє йому видавати себе за користувача та виконувати дії від його імені. Міжсайтова підробка запиту може бути використана для викрадення ідентифікаторів сеансу шляхом впровадження шкідливого коду на веб-сторінку, яка збирає дані сеансу користувача.

Незважаючи на ці недоліки, автентифікація із використанням сесій залишається популярним методом керування автентифікацією користувачів у web-додатках.

2.1.3 Дайджест аутентифікація

Дайджест аутентифікація являє собою більш захищену базову аутентифікацію. Ідеї даних підходів дуже схожі, за винятком того, що пароль передається у вигляді хешу, створеному із використанням алгоритму MD5.

При створенні хешу використовується нонс – довільне число, яке генерується випадково. У класичній схемі аутентифікації за це відповідає сервер, але можливі і інші варіації для посилення безпеки. Схема дайджест аутентифікації наведена на рисунку 2.3.

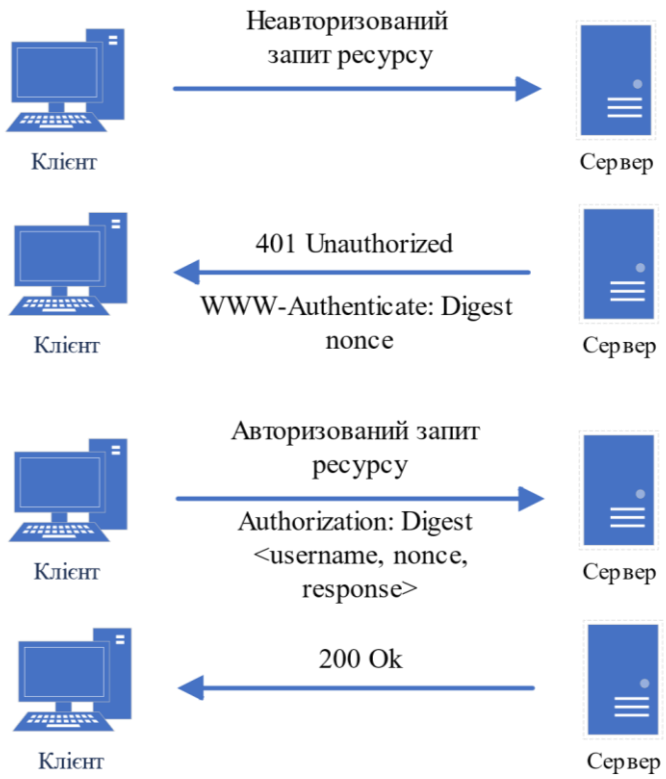

Рисунок 2.3. Схема дайджест аутентифікації

Клієнт робить неавторизований запит на отримання певних ресурсів із обмеженим доступом. У відповідь сервер повертає клієнту повідомлення із кодом

помилки 401 (не авторизовано) та заголовок «WWW- Authenticate: Digest», а також згенерований нонс.

Клієнт визначеним способом генерую хеш із використанням алгоритму MD5. Для генерації використовується пароль, та інша інформацію, включаючи наданий сервером нонс. Після цього клієнт відправляє запит із заголовком Authorization, в який додає ім’я користувача, нонс та створений хеш.

Отримавши хеш та супутні дані, сервер за визначеним алгоритмом генерує хеш із використанням наданого клієнтом нонсом та паролем користувача, який отримується із бази даних. Якщо хеші співпадають, то аутентифікація вважається успішною і доступ до ресурсів надається.

Переваги даного підходу полягають у більшій надійності у порівнянні із базовою аутентифікацію, крім того таке рішення легко інтегрувати у системи, враховуючи підтримку у протоколі HTTP,

Але більшість недоліків базової аутентифікації зберігаються, необхідне постійне відправлення даних та відсутній механізм виходу користувача із системи.

2.1.4 Аутентифікація із використанням токенів

Аутентифікація із використанням токенів — це сучасний спосіб аутентифікації користувачів у web-додатках. Замість того, щоб надсилати облікові дані з кожним запитом на сервер, користувач аутентифікується один раз, а потім отримує токен. Надалі цей токен надсилається з кожним наступним запитом, і сервер використовує його для перевірки того, що користувач усе ще автентифікований.

Загалом, до токену можна відносити як, наприклад, фізичні пристрої, так і цифрові токени. Вони можуть використовуватися у різних ситуаціях для різних задач. Крім того, цифрові токени можуть використовуватися для аутентифікації із різними підходами, наприклад зі збереженням стану чи без нього. При використанні даного методу, кожен запит має містити заголовок «Authorization» із вмістом «Bearer <значення токену>».

У великій кількості сучасних web-застосунків використовується так званий JWT. Особливістю даного токена є те, що він містить підпис, який робиться у момент його генерації та дозволяє перевірити відсутність модифікацій у корисному навантаженні чи інших.

JWT складається з трьох елементів:

- Заголовку (включає тип токена та алгоритм хешування, який використовується для підпису).
- Корисного навантаження (включає необхідну серверу інформацію про користувача, час функціонування токену та подібне).
- Підпису (хеш, який використовується для підтвердження відсутності модифікації токену, складається із закодованих у base64 значень заголовку та корисного навантаження, а також секрету, який відомий лише серверу, який генерує та валідує токени). Приклад розшифровки токена приведений на рисунку 2.4.

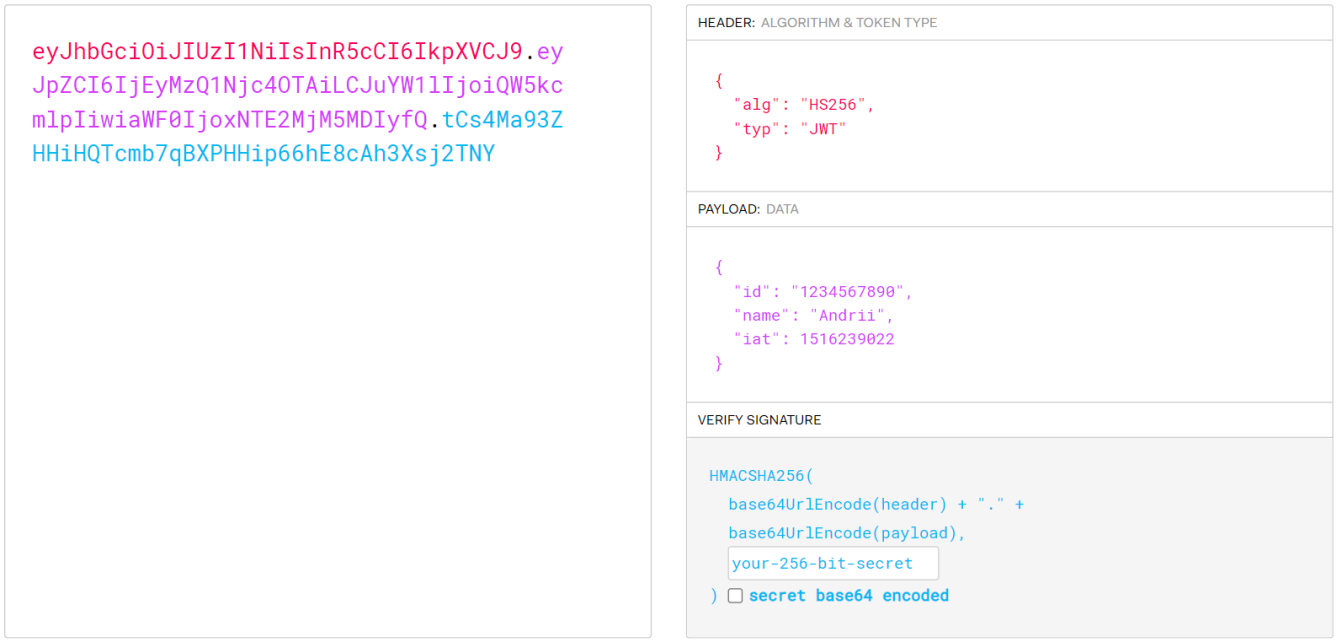

Рисунок 2.4. JWT у закодованому та розкодованому вигляді

Усі три частини кодуються у base64 та розділяються знаком крапки. Важливо відмітити, що токен є лише закодованим, а не зашифрованим, тому його вміст може бути переглянутий, включаючи всі значення крім секрету, оскільки для підпису

використовуються алгоритми хешування. Схема аутентифікації із використанням JWT зображена на рисунку 2.5.

Клієнт відправляє запит із даними необхідними для аутентифікації на сервер. Сервер перевіряє дані та генерує JWT, який містить необхідну йому для роботи інформацію та підпис, створений із використанням секрету сервера. При наступному запиті, клієнт відправляє отриманий JWT разом з іншим корисним навантаженням. Сервер розкодовує JWT, бере з нього значення заголовку та корисного навантаження, генерує новий хеш із відомим йому секретом та перевіряє чи дорівнює він значенню підпису. У разі співпадіння, надається доступ до ресурсу.

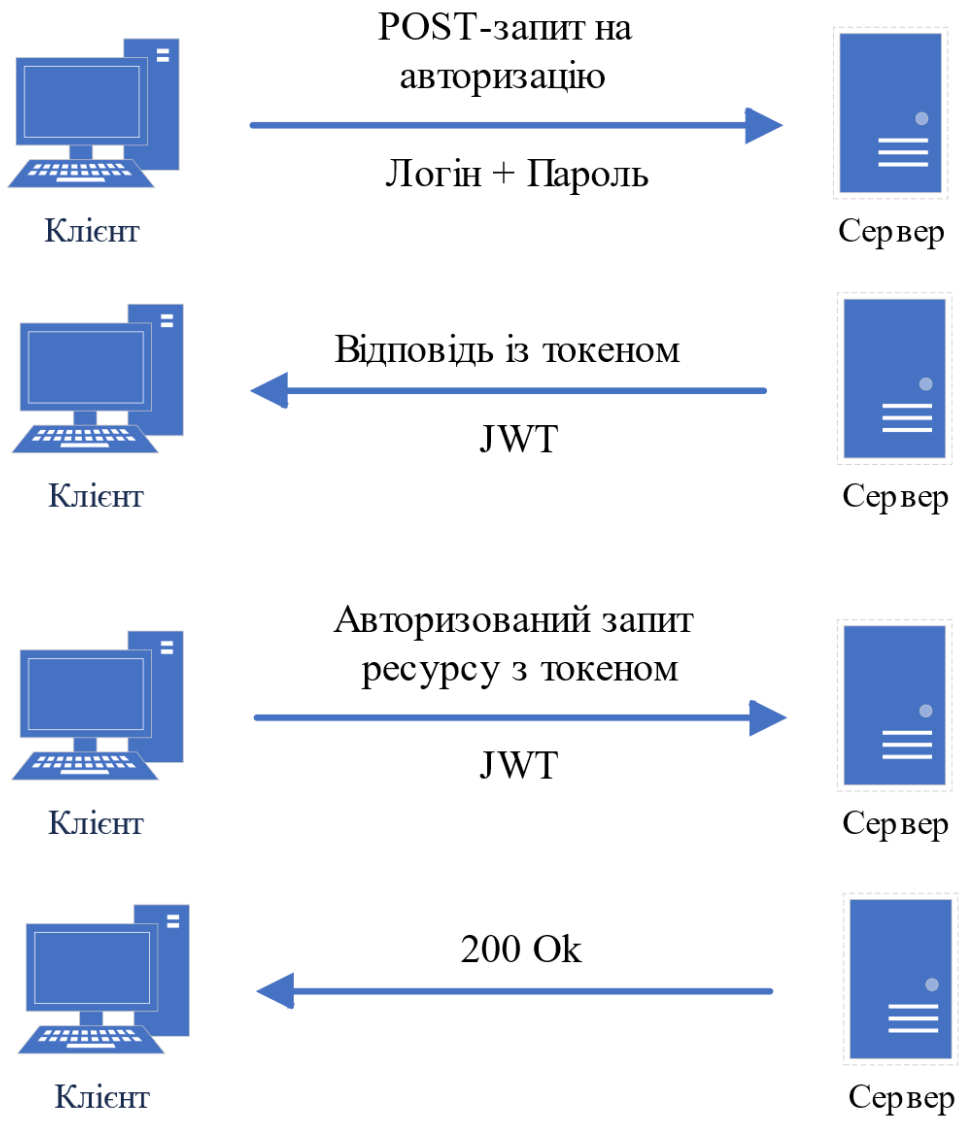

Рисунок 2.5. Схема аутентифікації із використанням JWT

Автентифікація із використанням токенів зазвичай використовується в сценаріях єдиного входу (SSO), коли користувач входить у систему один раз і може отримати доступ до кількох ресурсів без повторного входу. Даний підхід також широко використовується в мобільних додатках, де сесії нелегко підтримувати.

Однією з головних переваг даного підходу до аутентифікації є те, що він усуває потребу в сеансах, які можуть створювати навантаження у web-застосунках із високим трафіком. Токени можна зберігати на стороні клієнта, зменшуючи навантаження на сервер. Крім того, оскільки токени є автономними, їх можна використовувати на кількох серверах або доменах без необхідності централізованого керування сеансами.

Однак автентифікація із використанням токенів також має деякі потенційні недоліки. Якщо токен викрадено, зловмисник потенційно може використовувати його, щоб видати себе за користувача, доки термін дії токена не закінчиться. Крім того, оскільки токени часто зберігаються на стороні клієнта, вони можуть бути вразливими до міжсайтового скриптингу.

Загалом автентифікація із використанням токенів — це гнучкий і масштабований спосіб автентифікації користувачів у web-додатках, але його слід використовувати з обережністю та з відповідними заходами безпеки.

2.1.5 Використання протоколу OpenID для аутентифікації

OAuth (Open Authorization) і OpenID — це протоколи, які використовуються для авторизації та автентифікації відповідно. Вони часто працюють у зв’язці, але служать різним цілям.

OAuth — це стандарт авторизації, який дозволяє програмам сторонніх розробників отримувати доступ до ресурсів від імені власника. Це означає, що користувач може надати програмі дозвіл на доступ до своїх даних на сервері, не повідомляючи свої облікові дані, наприклад ім’я користувача та пароль. Замість цього програма отримує токен, який відкриває їй доступ до необхідних ресурсів. Цей токен може бути відкликаний користувачем у будь-який момент.

OpenID — це протокол, який дозволяє користувачам проходити автентифікацію, використовуючи єдиний обліковий запис від надійного стороннього постачальника ідентифікаційної інформації. Він надає web-додаткам стандартизований спосіб автентифікації користувачів і дозволяє використовувати наявні облікові записи соціальних мереж або електронної пошти для входу у систему на різні веб-сайти та програми.

Аутентифікація зазвичай відбувається шляхом натискання спеціальної кнопки на веб-сторінці чи у застосунку, яка перенаправляє користувача на сторінку для входу в систему сервісу, в якому він має обліковий запис. Після аутентифікації користувач повертається на ресурс.

Перевагою даного підходу є універсальність та безпека. Маючи лише один обліковий запис, користувач може використовувати різні сервіси без необхідності створювати акаунти на кожному з них. Крім того, у разі інциденту безпеки та витоку даних на сторонньому сервісі, облікові дані не потраплять до рук зловмисників.

Мінусом даного підходу є необхідність мати обліковий запис у системі провайдера OpenID. До того ж, дозволи, які надаються з використанням OAuth можуть бути використані недобросовісними розробниками для отриманням більшої кількості даних користувачів, ніж необхідно для функціонування сервісу. Крім того, у разі наявності проблем у провайдерів OpenID, користувачі не зможуть отримати доступ до системи, хоча слід зауважити, що надавачі подібних послуг зазвичай – великі технологічні компанії, які мають достатньо ресурсів для інвестування у безпеку.

Використання OAuth і OpenID є поширеною практикою на значній кількості веб-ресурсів. Багато великих компаній, включаючи Google, Apple і Microsoft, впровадили ці протоколи та надали іншим застосункам можливість інтегрувати їх рішення. Вони забезпечують безпечний і зручний підхід для доступу до ресурсів у мережі без шкоди обліковим даним користувача чи конфіденційності. Подібний метод аутентифікації є зручним для користувача.

2.2 Використання одноразових паролів у якості другого фактору

Мета другого фактору — забезпечити додатковий рівень безпеки на додаток до традиційної комбінації імені користувача та пароля, яка може бути скомпрометована зловмисниками.

При двофакторній аутентифікації користувач надає два різні типи факторів: те, що він знає, наприклад пароль, і те, що він має, наприклад фізичний пристрій. Цим фізичним пристроєм може бути телефон, планшет або спеціальний апаратний токен.

Коли користувач намагається увійти у систему, він повинен спочатку ввести своє ім’я користувача та пароль, як зазвичай. Потім йому буде запропоновано надати другий фактор автентифікації. Це може бути одноразовий код, надісланий на телефон, або згенерований апаратним токеном.

Останніми роками впровадження другого фактору стає все більш важливим, оскільки витоки і викрадення особистих даних стають все більш поширеними. Використання двофакторної аутентифікації допомагає захистити облікові записи від несанкціонованого доступу та гарантує, що лише авторизовані користувачі можуть отримати доступ до конфіденційної інформації.

Існує кілька різних підходів до впровадження другого фактору. Такими можуть бути: одноразові паролі, отримані через SMS або згенеровані у спеціальних програмах, мобільні застосунки та апаратні токени.

У контексті веб-сервісів популярним рішенням є саме одноразові паролі, що пов’язано із легкістю їх отримання користувачем. Наприклад, вони можуть бути відправленні повідомленням на мобільний пристрій, використовуючи спеціальні сервіси. Крім того, існує велика кількість мобільних застосунків, які дозволяють генерувати одноразові паролі прямо на пристрої користувача. Великі компанії можуть дозволити використовувати мобільний пристрій в якості другого фактору. Зазвичай це реалізовується за допомогою мобільного застосунку, який пов’язаний із певним сервісом, в якому користувач може підтвердити спробу авторизації або отримати одноразовий пароль.

Існує багато алгоритмів для генерації одноразових паролів, але основними є HOTP (HMAC-based One-Time Password) та TOTP (Time-Based One-Time Password) [22].

Алгоритм HOTP зосереджений навколо секретного ключа, також відомого як «сід», обмін яким здійснюється між токеном і сервером лише один раз під час ініціалізації. Після цього обміну секретний ключ надійно зберігається як токеном, так і сервером і більше не поширюється мережею. Крім того алгоритм HOTP використовує лічильник на основі подій. Значення лічильника збільшується щоразу, коли користувач натискає кнопку на токені. З іншого боку, лічильник сервера збільшується після кожної успішної автентифікації. Коди HOTP генеруються за допомогою алгоритму одноразового пароля на основі HMAC (Hash-Based Message Authentication Code).

HMAC є криптографічним методом, який включає криптографічну хеш-функцію, як правило, SHA-1, і набір параметрів, включаючи секретний ключ і лічильник. Схема генерації HOTP зображена на рисунку 2.6.

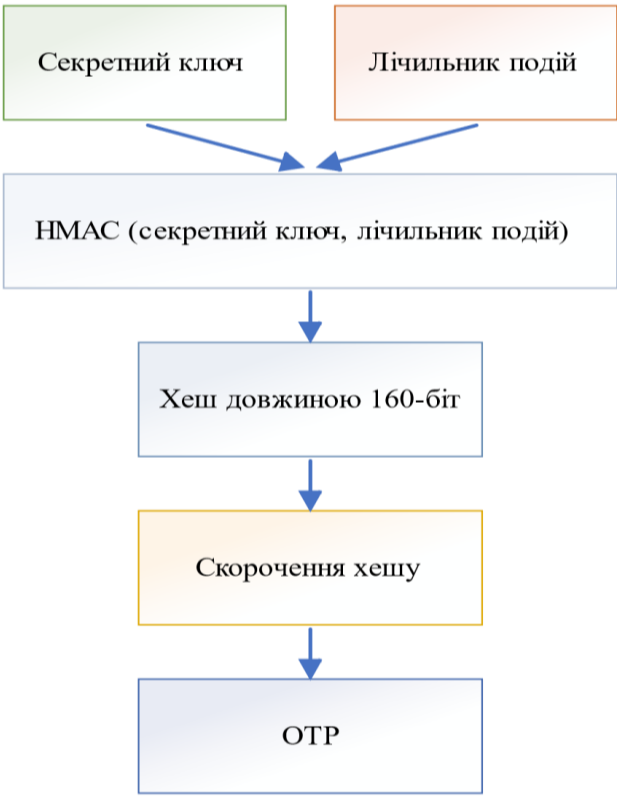

Рисунок 2.6. Алгоритм генерації OTP з використанням алгоритму HOTP

Вихід HMAC – це 160-бітний хеш, який є занадто довгим для практичного використання. Алгоритм HOTP скорочує хеш-значення та перетворює його на зрозуміле для людини ціле число за допомогою операції модуля. Остаточним виходом є рядок цифр, який користувач може легко прочитати та ввести на сторінці входу.

Коли користувач хоче пройти автентифікацію за допомогою свого токена HOTP, він вводить значення, що відображається на токені, у текстове поле на сторінці входу. Потім сервер генерує власний OTP і порівнює його з OTP користувача. Якщо вони збігаються, сервер надає доступ користувачеві, а потім автоматично генерує нове значення одноразового паролю. Користувач повинен натиснути кнопку на маркері HOTP після успішної автентифікації, щоб оновити значення лічильника.

Одноразовий пароль на основі часу (TOTP) — це алгоритм OTP, який використовує спільний секретний ключ і лічильник часу. TOTP побудовано на основі алгоритму HOTP із заміною типу лічильника – із подій на часу. Щоб створити код TOTP, поточний час Unix ділиться на значення часового кроку (довжина «життя» одноразового паролю), яке зазвичай становить 30 секунд, для обчислення лічильника часу. Подальший процес генерації коду TOTP такий самий, як і для HOTP (рис. 2.7).

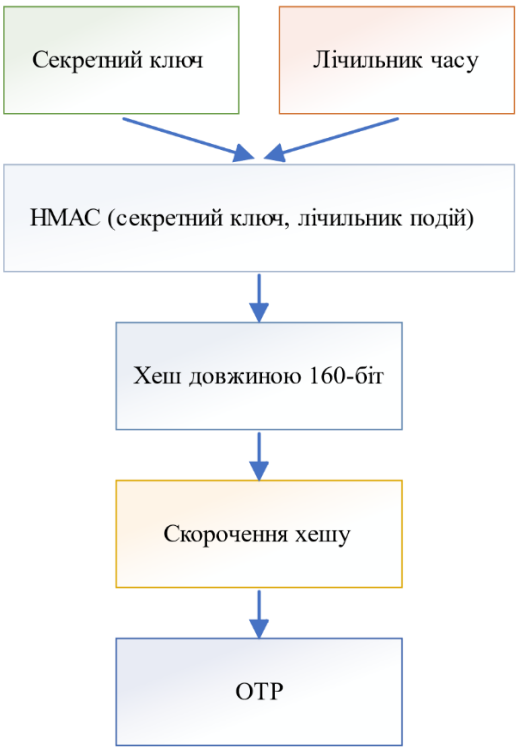

Рисунок 2.7. Алгоритм генерації OTP з використанням алгоритму TOTP

Спільний секретний ключ і лічильник часу хешуються за допомогою HMAC для отримання хеш-значення, яке скорочується та перетворюється на зрозуміле для людини ціле число за допомогою операції модуля. Остаточним виходом є рядок цифр, який служить кодом TOTP. Коли користувач хоче пройти автентифікацію за допомогою TOTP, він вводить код TOTP, який відображається на його пристрої, у текстове поле на сторінці входу.

Сервер генерує власний код TOTP, використовуючи той самий спільний секретний ключ і лічильник поточного часу, і порівнює його з кодом TOTP користувача. Якщо два коди збігаються, сервер надає доступ користувачеві. Алгоритм TOTP гарантує, що код дійсний лише протягом певного періоду часу, зазвичай 30 секунд, що додає додатковий рівень безпеки процесу автентифікації.

Головною проблемою при використанні одноразового пароля на базі HOTP є можлива вірогідність розсинхронізації лічильників клієнта та сервера. Якщо користувач оновить пароль багато разів і при цьому не буде авторизуватися у системі, його лічильник «піде» набагато далі, ніж лічильник сервера. Вирішенням даної проблеми може бути розширення «вікна» прийому паролів сервером, тобто мають прийматися декілька попередніх та наступних паролів відповідно до поточного стану лічильника. Проте такий підхід збільшує ризики компрометації OTP та не вирішує проблеми великої різниці між значеннями лічильників. Крім того, HOTP не має обмеження у часі «життя». Якщо зловмисник зможе побачити пароль і використати його до того, як користувач виконає вхід у систему, дані можуть бути скомпрометованими.

TOTP вирішує обидві дані проблеми шляхом використання лічильника часу. Даний тип одноразових паролів прив’язаний до часу Unix, який є універсальним для пристроїв, що прибирає проблему розсинхронізації. До того ж, подібні паролі мають визначений термін життя та оновлюються автоматично без потреби у будь-яких діях зі сторони користувача.

Загалом, одноразові паролі є гарним рішенням для використання в якості другого фактору. Це універсальний підхід, який не потребує додаткових апаратних засобів і нескладний у реалізації. Але існують і певні недоліки, серед яких важкість

відновлення доступу у разі проблем із пристроєм, який відповідає за генерацію OTP та необхідність надійного зберігання ключа як на стороні клієнта, так і на стороні сервера.

2.3 Загрози безпеці web-застосунків

Web-сервіси стали невід'ємною частиною сучасної архітектури програмного забезпечення. Однак, як і будь-який програмний додаток, веб-сервіси вразливі до різних загроз безпеці, які можуть призвести до витоку або втрати даних і несанкціонованого доступу. Ці загрози створюють значні ризики для безпеки та цілісності веб-сервісів, а також для користувачів, які мають до них доступ, та їх даних. Основними атаками, які несуть ризик для API є: ін’єкції, XSS, DDOS, MITM та викрадення акаунтів [23].

Однією з найпоширеніших загроз безпеці веб-сервісів є так звані ін’єкції. Ін’єкційні атаки відбуваються, коли зловмисник впроваджує шкідливий код у веб-сервіс, використовуючи вразливі місця в механізмах перевірки вхідних даних програми. Зловмисник може використати цей код для виконання несанкціонованих дій, таких як викрадення конфіденційних даних, видалення даних або отримання контролю над системою.

Іншою значною загрозою безпеці веб-сервісів є атаки міжсайтових сценаріїв (XSS). XSS-атаки відбуваються, коли зловмисник впроваджує шкідливий код у веб-сервіс, який потім виконується браузером жертви. Цей код можна використовувати для викрадення конфіденційної інформації, як-от облікових даних або даних кредитної картки, або для виконання несанкціонованих дій від імені жертви.

Ще однією поширеною загрозою для веб-сервісів є атаки типу «відмова в обслуговуванні» (DoS). DoS-атаки відбуваються, коли зловмисник перевантажує веб-сервіс великим обсягом трафіку або запитів, в результаті чого сервіс перевантажується та стає недоступним для користувачів. DoS-атаки можуть призвести до значних фінансових втрат, а також завдати шкоди репутації постраждалої організації.

Атаки типу «людина посередині» (MITM) становлять серйозну загрозу безпеці веб-сервісів, які передбачають передачу конфіденційних даних. Атаки MITM відбуваються, коли зловмисник перехоплює зв’язок між двома сторонами та змінює дані, що передаються. Цей тип атаки може призвести до викрадення конфіденційної інформації, наприклад облікових даних або фінансових даних, і може призвести до значних фінансових втрат.

На додаток до цих загроз, веб-сервіси також уразливі до викрадення сесії, паролів та інших типів порушень безпеки. Щоб пом’якшити ці загрози, розробники веб-сервісів повинні впровадити відповідні заходи безпеки, такі як перевірка введених даних, шифрування, контроль доступу та механізми автентифікації.

У згаданому в першому розділі дослідженні Salt Security зазначається, що серед респондентів найбільше занепокоєння викликають: так звані «зомбі» API (54%), викрадення облікових записів (43%), DoS-атаки (31%), випадкове розголошення чутливих даних (27%), витоки даних (25%), «тіньові» API (20%) [4].

Неприбуткова організація OWASP, яка спеціалізується на дослідженнях та поширенні матеріалів щодо безпеки web-застосунків, випускає звіт під назвою «OWASP API Security Top 10», в якому розглядає найбільш вразливі місця API та способи їх ліквідації. Цікавим фактом є те, що згідно вже згаданого опитування Salt Security, лише 54% опитаних зазначили, що їх організація фокусуються на проблемах зазначених у звіті OWASP, хоча 66% усіх атак спрямовані на зазначені там вразливості або використовують їх комбінації [4]. OWASP API Security Top 10 визначає десять основних загроз, як будуть розглянуті далі [24].

Порушена авторизація на рівні об’єкта: API часто надають кінцеві точки, які мають справу з ідентифікаторами об’єктів, що може призвести до проблеми з авторизацією на рівні об’єкта. Перевірки авторизації на рівні об’єкта слід розглядати для кожного методу, який використовує дані, надані користувачем, для доступу до джерела даних. Загроза авторизації на рівні об’єкта виникає, через відсутність належної системи авторизації. Це може призвести до отримання зловмисником доступу до конфіденційної інформації або виконання дій, на які він не має права. Одним із способів пом’якшити цю загрозу є впровадження відповідної системи

авторизації, яка залежить від політики та ієрархії користувачів. Ця система повинна мати можливість контролю доступу на рівні об'єкта. Також рекомендується використовувати довільні та непередбачувані GUID для ідентифікаторів записів замість послідовних номерів. Це ускладнює можливості вгадування наступного ідентифікатора і отримання доступ до інформації, яку він не повинен отримати.

Проблеми з аутентифікацією користувачів: автентифікації є складним елементом системи, і якщо вона реалізована неправильно, можуть виникнути вразливості, якими можуть скористатися зловмисники. Важливо реалізувати належні механізми автентифікації для розпізнавання клієнта або користувача, який отримує доступ до системи. Щоб зменшити ці загрози, важливо дотримуватися стандартів автентифікації у сферах створення токенів, політики паролів і зберігання.

Надмірне розголошення даних: API часто надають клієнтам атрибути об’єктів, не враховуючи їх чутливість. Ця практика є ризикованою, оскільки розробники очікують, що клієнти фільтруватимуть дані, перш ніж представляти їх користувачеві. Однак такий підхід може призвести до витоку даних, оскільки конфіденційні дані можуть витікати неавторизованим сторонам. Щоб зменшити цю загрозу, розробники повинні мати чітке уявлення про дані, які вони зберігають або повертають клієнтам.

Відсутність обмежень на доступ до ресурсів: API не завжди встановлюють обмеження на розмір або кількість ресурсів, які клієнти можуть запитувати, що призводить до кількох ризиків безпеки. Зловмисники можуть використати цю слабкість, запитуючи велику кількість ресурсів, спричиняючи погіршення продуктивності веб-сервера та потенційно призводячи до атаки типу «відмова в обслуговуванні». Крім того, необмежені запити ресурсів можуть призвести до проблем автентифікації, наприклад до атак грубою силою. Щоб зменшити ці ризики, важливо вжити відповідних заходів. Наприклад, API можуть обмежувати частоту викликів, які може здійснювати клієнт, щоб переконатися, що запити ресурсів не перевантажують сервер.

Проблеми із авторизацією на рівні функцій: API можуть мати проблеми з авторизацією через складні правила контролю доступу, що включають різні ієрархії, групи та ролі. Ці правила можуть стирати межу між адміністративними та

звичайними операціями, даючи зловмисникам можливість скористатися цими недоліками та отримати доступ до ресурсів інших користувачів або адміністративних функцій. Для виявлення цих проблем, важливо чітко визначати ролі кожної функції.

Масове призначення: дана вразливість виникає, коли надані клієнтом дані пов'язуються з моделями даних без достатньої фільтрації атрибутів. Зловмисники можуть змінювати властивості об'єктів без дозволу, вгадуючи властивості об'єктів, досліджуючи інші кінцеві точки API, читаючи документацію або додаючи додаткові атрибути об'єктів для запиту необхідних даних. Щоб послабити загрозу, розробники можуть реалізувати функції, які безпосередньо не перетворюють вхідні дані клієнта у внутрішні об'єкти чи змінні, включають лише дозволені атрибути, які клієнт повинен оновити, і визначають схеми для правильної обробки корисних навантажень вхідних даних.

Неправильні конфігурації безпеки: дані вразливості можуть виникати через незахищені конфігурації, такі як відкрите хмарне сховище, неправильно налаштовані HTTP заголовки, непотрібні HTTP методи, дозволений обмін ресурсами між джерелами (CORS) і докладні повідомлення про помилки, що містять конфіденційну інформацію. Для подолання цих проблем, важливо впроваджувати перевірені заходи безпеки та використовувати політики та процеси для забезпечення безпеки конфігурації.

Ін'єкції: атаки, такі як SQL або NoSQL ін'єкції чи ін'єкція команди, відбуваються, коли ненадійні дані передаються серверу як частина команди чи запиту. Це може дозволити зловмисникам виконувати неавторизовані команди або отримувати доступ до конфіденційних даних. Щоб запобігти ін'єкціям, необхідно перевіряти всі введені клієнтом дані, обмежити кількість повернутих даних і чітко визначте типи даних для рядкових параметрів.

Неправильне керування ресурсами: точна документація має вирішальне значення для API, оскільки вони зазвичай мають більше кінцевих точок, ніж звичайні веб-програми. Застарілі версії API та відкриті кінцеві точки – це дві проблеми, які можна вирішити шляхом належної інвентаризації розгорнутих версій API та хостів. Щоб забезпечити належний захист, усі активи, включно з API, слід інвентаризувати.

Крім того, активи слід класифікувати на основі їх чутливості та критичності для бізнесу.

Неправильно налаштоване логування та моніторинг є серйозними проблемами безпеки для API. Зловмисники можуть використовувати вразливості, отримувати доступ і переходити до інших систем, щоб змінювати, викрадати або видаляти дані. Важливо постійно реєструвати дії у системі, контролювати інфраструктуру, мережу та API, а також безпечно зберігати логи в централізованому місці, де вживаються необхідні заходи для контролю доступу.

Загалом загрози безпеці, з якими стикаються веб-сервіси, численні та різноманітні, і вони вимагають комплексного підходу до безпеки, який передбачає не лише впровадження відповідних заходів безпеки, але й постійний моніторинг та аналіз системи для виявлення потенційних загроз і реагування на них своєчасно та ефективно. Застосовуючи проактивний підхід до безпеки, розробники веб-сервісів можуть допомогти забезпечити безпеку та цілісність своїх систем і даних, які вони обробляють і передають.

Висновки за розділом 2

У даному розділі були розглянуті методи аутентифікації у web-застосунках, елементи другого фактору, які можна використати у web-застосунках, а також можливі загрози їх функціонуванню. Автентифікація — це процес підтвердження особи користувача або клієнта, який намагається отримати доступ до ресурсу. Щоб забезпечити доступ до ресурсів лише авторизованим особам або організаціям, у web-застосунках використовуються різні методи автентифікації.

Існують різні типи механізмів автентифікації, які можна використовувати для захисту web-додатків і API. Найпростішим є базова автентифікація, яка передбачає надсилання імені користувача та пароля у вигляді відкритого тексту з кожним запитом. Цей підхід не рекомендується використовувати через слабку безпеку та постійні відправлення облікових даних користувача на сервер.

Автентифікація із використанням сесій передбачає створення файлу сесії користувача після його автентифікації та збереження цього файлу на сервері. Після генерації сесії, сервер повертає її ідентифікатор користувачу у файлі кукі. У подальшому особа користувача перевіряється під час кожного наступного запиту за допомогою ідентифікатора сесії. Цей підхід є більш безпечним, ніж базова автентифікація, але вимагає додаткових ресурсів на стороні сервера.

Дайджест-автентифікація схожа на базову автентифікацію, але замість надсилання пароля у вигляді відкритого тексту використовує криптографічний хеш. Цей підхід є більш безпечним, ніж базова автентифікація, але все одно вимагає постійного відправлення облікових даних користувача на сервер.

Автентифікація з використанням токенів передбачає використання токенів (часто JWT), які зазвичай генеруються сервером і надсилаються клієнту після успішної автентифікації. Потім клієнт надсилає токен з кожним наступним запитом, який сервер, в свою чергу, перевіряє. Цей підхід широко використовується і вважається доволі безпечним.

OpenID – це протокол автентифікації, який дозволяє користувачам проходити аутентифікацію в кількох web-додатках за допомогою одного набору облікових даних. Він використовує комбінацію різних технологій, включаючи OAuth, щоб забезпечити безпечний і стандартизований механізм автентифікації та авторизації.

HOTP і TOTP — два алгоритми, які використовуються для генерації одноразових паролів. HOTP генерує OTP, хешуючи спільний секретний ключ зі значенням лічильника, тоді як TOTP — хешуючи спільний секретний ключ із поточним часом.

HOTP — це статичний алгоритм, тобто значення лічильника має бути синхронізовано між клієнтом і сервером. З іншого боку, TOTP — це динамічний алгоритм, який змінює пароль кожні заздалегідь визначений проміжок часу, як правило, 30 секунд.

Обидва алгоритми зазвичай використовуються в системах двофакторної автентифікації, де користувачі повинні ввести одноразовий пароль на додаток до свого імені користувача та пароля, щоб отримати доступ до системи або програми.

Загрози API – це вразливі місця та ризики безпеки, які можуть поставити під загрозу конфіденційність, цілісність і доступність даних і систем, які покладаються на API. Ці загрози включають ін’єкційні атаки, порушену автентифікацію та контроль доступу, незахищене зберігання та передачу конфіденційних даних, слабкий моніторинг і логування, неправильні конфігурації безпеки, а також недостатні обмеження на кількість запитів. Щоб зменшити загрози, важливо впроваджувати належні засоби контролю безпеки, такі як використання механізмів шифрування та автентифікації, застосування політик контролю доступу, ведення логів і моніторингу, а також регулярне тестування на вразливості.

РОЗДІЛ 3
РОЗРОБКА WEB-ЗАСТОСУНКУ НА БАЗІ АРХІТЕКТУРИ REST З МОЖЛИВІСТЮ ДВОФАКТОРНОЇ АВТЕНТИФІКАЦІЇ КОРИСТУВАЧА

3.1 Архітектура web-застосунку

Як вже було зазначено, додавання другого фактору для аутентифікації значно підвищує рівень безпеки облікового запису користувача та його даних. Додатковий рівень безпеки однозначно необхідний для систем, в яких циркулює конфіденційна інформація користувачів, наприклад банківського чи державного сектору. Не зважаючи на це, впровадження двофакторної аутентифікації у застосунку будь-якої області функціонування є непоганою практикою. Користувач може бажати підвищеної безпеки свого облікового запису у соціальних мережах, де відбуваються приватні розмови, чи сервісах потокової передачі відео, за які користувач сплачує кошти із власного гаманця.

Основний функціонал застосунку, який буде розроблений у даному розділі, полягатиме у можливості користувача створювати пост, який міститиме заголовок та основний текст. Даний пост може бути вподобаний іншими користувачами.

Операції над постом включатимуть:

- створення;
- оновлення;
- отримання;
- видалення.

Умовна схема запитів представлена на рисунку 3.1. Пост можна буде створити, оновити, отримати список усіх постів від сервера, отримати конкретний пост за id. Звісно, пост зможе бути видаленим.

Створення буде відбуватися шляхом відправлення POST-запиту, у тілі якого міститимуться усі необхідні поля для створення посту. Отримання постів буде відбуватися відправленням GET-запитів із вказуванням кількості постів та фільтрів, або id посту, у випадку із запитом на отримання конкретного запису. Оновлення буде відбуватися за рахунок PUT-запиту, із зазначенням id в якості параметру запиту та

тілом, яке міститиме оновлену інформацію. Видалення буде відбуватися шляхом відправлення DELETE-запиту із id необхідного посту.

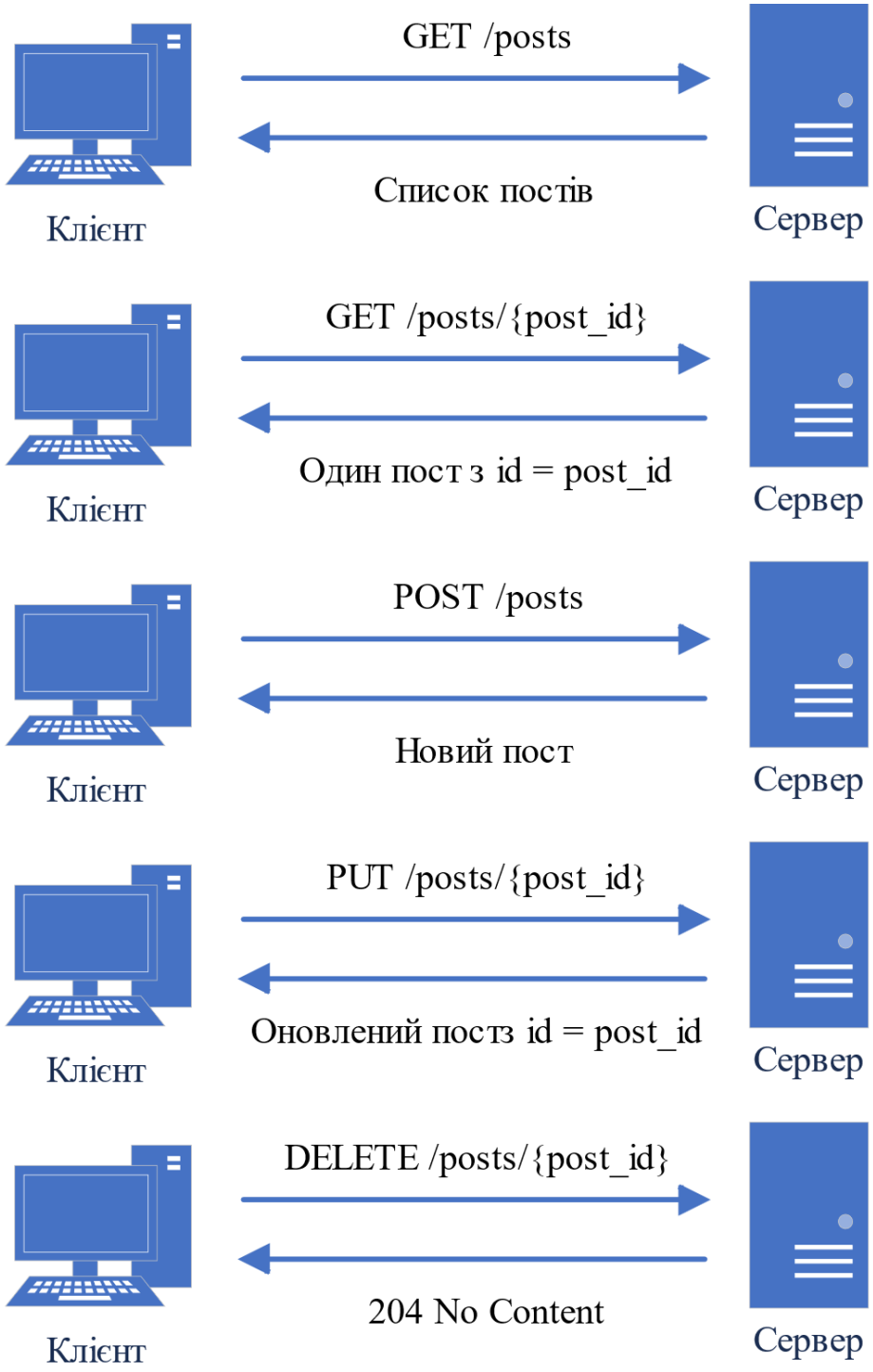

Рисунок 3.1. Операції над постом

Операції будуть здійснюватися від імені користувача, тож необхідною частиною застосунку є кінцева точка, яка дозволить створювати обліковий запис. Схема запитів, пов’язаних з користувачем, представлена на рисунку 3.2.

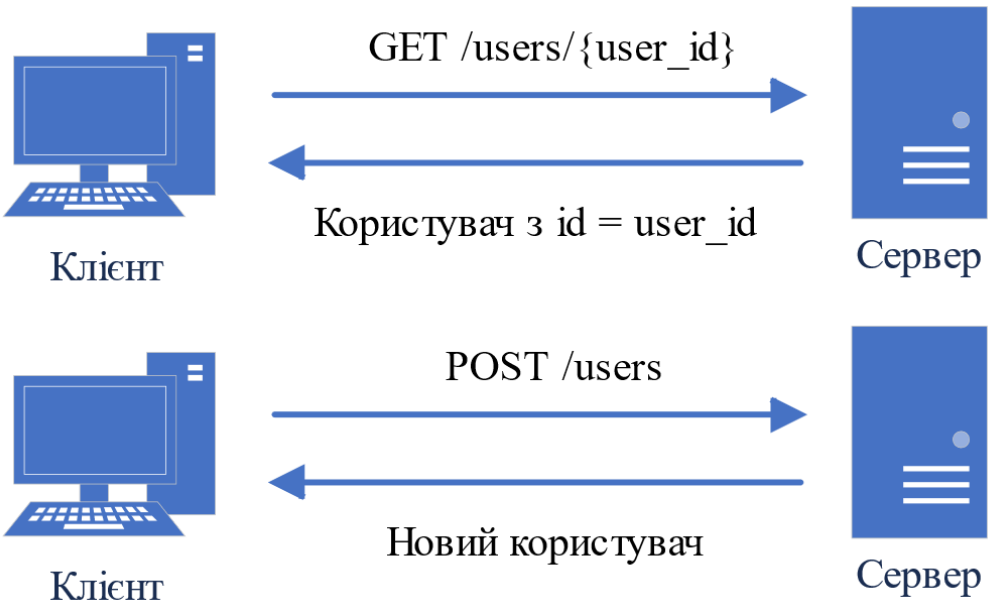

Рисунок 3.2. Операції над користувачем

Користувача можна буде створити (POST-запит з обліковими даними) та отримати інформацію про нього за допомогою id.

Крім того буде створена кінцева точка «/vote», яка міститиме одну операцію, що дозволятиме голосувати за певний пост (рис. 3.3).

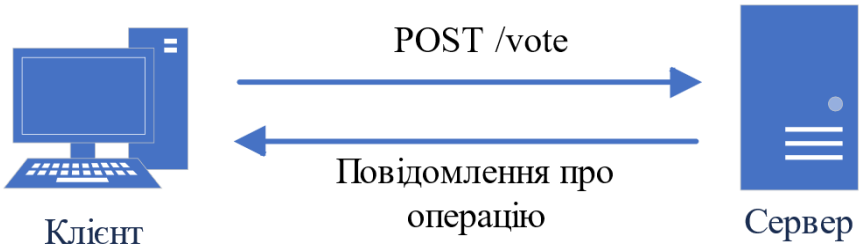

Рисунок 3.3. Операція голосування

Операція голосування буде виконуватися за допомогою POST-запиту, який міститиме у своєму тілі id посту та «напрямок» голосу – вверх або вниз.

Усі дії відбуватимуться від імені певного користувача, а отже необхідно реалізувати систему автентифікації та авторизації. Аутентифікація у системі буде реалізовано за рахунок використання веб-токенів, а саме JWT. Схема аутентифікації наведена на рисунку 3.4.

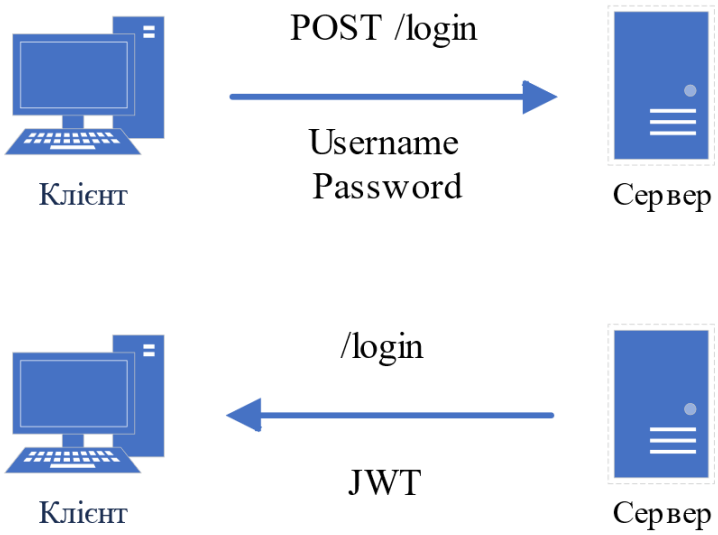

Рисунок 3.4. Схема автентифікації

Клієнт відправляє POST-запит до кінцевої точки login, який містить ім'я користувача та пароль. Сервер перевіряє облікові дані та повертає JWT, який містить певну інформацію, наприклад id користувача, термін дії і так далі.

Як вже було зазначено у другому розділі, існують різні елементи, які можуть бути використані в якості другого фактору. Їх можна використовувати для підвищення безпеки облікового запису користувача. Одним із найпопулярніших є TOTP або одноразовий пароль на основі часу. Більшість популярних служб, таких як соціальні мережі чи служби електронної пошти, використовують цей тип OTP у якості другого фактору, оскільки користувачі легко можуть зберегти його в програмі на своєму телефоні чи інших пристроях. У порівнянні з HOTP, користувачам не потрібно оновлювати код, оскільки це робиться автоматично за допомогою таймера.

Щоб реалізувати двофакторну автентифікацію, необхідно створити нову кінцеву точку (/otp/generate), яка дозволить користувачеві згенерувати новий секрет OTP. Схема запитів і відповідей для ввімкнення двофакторної аутентифікації зображена на рисунку 3.5.

Клієнт надсилає запит до кінцевої точки «/otp/generate», надаючи JWT, який був згенерований сервером під час попереднього входу та містить ідентифікатор користувача. Сервер зчитує ідентифікатор користувача з JWT і генерує новий секрет,

який потім поєднується із записом цього користувача в базі даних. Генерацію секрету буде здійснена за допомогою спеціальних модулів мови програмування.

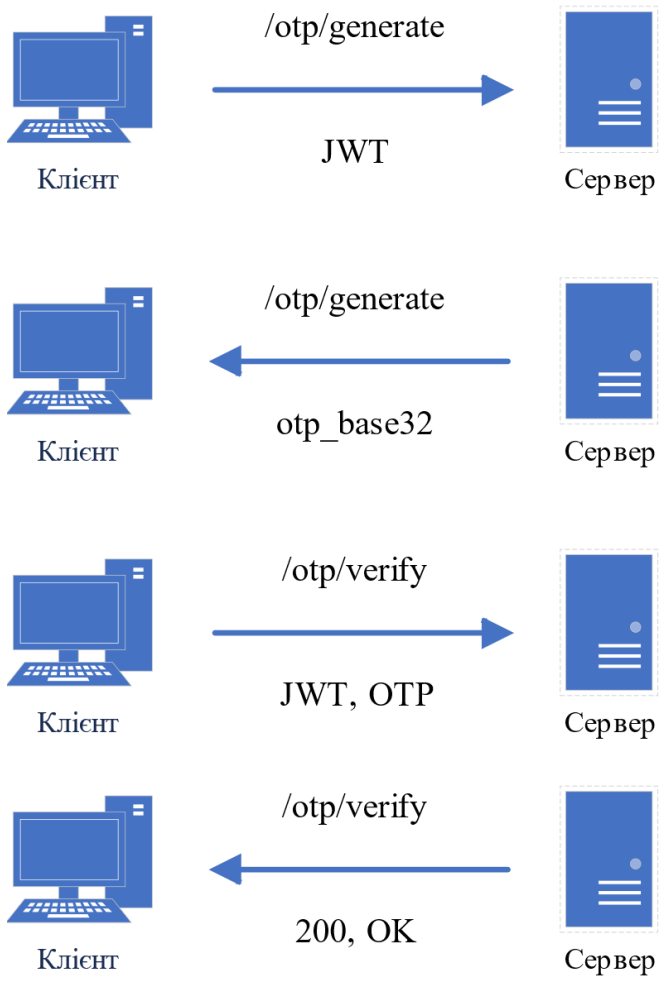

Рисунок 3.5. Схема увімкнення другого фактору

Коли секрет збережено, сервер надсилає відповідь клієнту з корисним навантаженням, що містить цей секрет. Щоб перевірити, чи OTP працює належним чином, необхідно створити нову кінцеву точку для його підтвердження. Цією кінцевою точкою може бути, наприклад, «/otp/verify». Корисним навантаженням будуть JWT і OTP, які можна розмістити в заголовку або в тілі запиту. Якщо все гаразд, перевірку додатковим фактором успішно ввімкнено для облікового запису користувача, що, в свою чергу, має бути позначено в базі даних.

Під час наступного входу, клієнту буде запропоновано надати облікові дані та OTP (рис. 3.6).

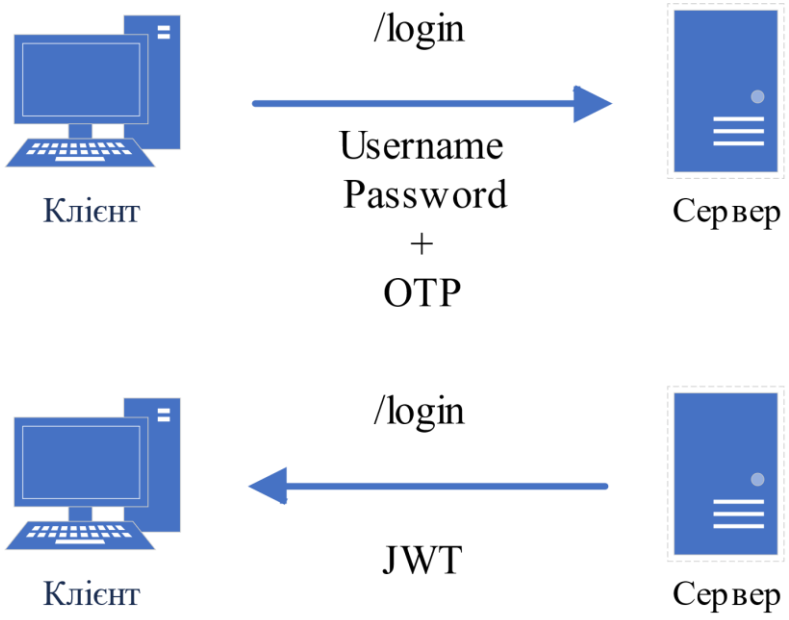

Рисунок 3.6. Схема автентифікації з увімкненим 2-м фактором

Коли клієнт, для облікового запису якого увімкнено двофакторну аутентифікацію, надсилає запит, для входу в систему, із корисним навантаженням, що містить облікові дані та OTP, сервер повинен перевірити облікові дані, отримати секрет користувача із бази даних, згенерувати OTP і порівняти його з наданим. Якщо OTP однакові, автентифікація пройшла успішно, і JWT потрібно повернути клієнту.

3.2 Вибір технологій для розробки web-застосунку

Обираючи мову програмування, на якій буде виконуватися розробка web-застосунка, необхідно звернутися до тих, які традиційно використовуються для створення «серверних» частин продукту. Такими є Python, Java, JavaScript, PHP, C#, Ruby, Golang і так далі. Важливо розуміти, що окрім мови, важливим елементом при виборі технології є фреймворк. Для однієї мови може існувати декілька різних бібліотек, які дозволяють реалізувати бажані функції, але у кожної з них є свої особливості.

Не існує єдиної «найкращої» мови для розробки web-застосунків, оскільки різні мови мають різні сильні та слабкі сторони залежно від конкретного випадку використання та вимог програми. Зрештою, вибір мови для розробки API залежить

від таких факторів, як цільова платформа, вимоги до продуктивності, досвід розробника, а також особливі функції та функції, необхідні для API. Для створення застосунку у даній кваліфікацій роботі буде використовуватися Python.

Python — популярна мова програмування високого рівня, відома своєю простотою, легкістю використання та універсальністю. Його створив наприкінці 1980-х Гвідо ван Россум, який працював у голландському національному дослідницькому інституті CWI. Він назвав його на честь комедійної групи «Monty Python», фанатом якої він був. Синтаксис Python був створений таким чином, щоб бути читабельним і лаконічним, що робить його хорошим вибором як для початківців, так і для досвідчених програмістів. З тих пір дана мова програмування стала однією з найпоширеніших у світі, із застосуваннями в різних сферах, від веб-розробки до машинного навчання.

Однією з найбільших переваг Python є його відносна простота. Код Python легко читати та писати, що дозволяє розробникам писати чисто та лаконічно. Python також має велику й активну спільноту, яка створила багато бібліотек і фреймворків, які полегшують роботу з мовою. Python також відомий своєю універсальністю. Його можна використовувати для широкого спектру додатків, від простих скриптів до складних програм для аналізу даних і алгоритмів машинного навчання. Він також використовується для веб-розробки, для якої застосовують фреймворки, такі як Django, Flask та FastAPI.

Слабкі сторони Python включають його відносно повільну швидкість порівняно з мовами нижчого рівня або деякими компільованими мовами високого рівня. Хоча дана особливість не є значною проблемою для більшості програм, це може бути проблемою при роботі з дуже великими наборами даних або складними алгоритмами, які вимагають високої продуктивності обчислень. Інша потенційна слабкість полягає в тому, що динамічна типізація Python та робота за принципом інтерпретації може ускладнити виявлення помилок під час розробки.

Для розробки web-застосунків за допомогою Python, як правило, використовуються три основні фреймворки, які вже були згадані вище – це Django, Flask та FastAPI.

Flask — це відносно легкий фреймворк, який вважається простим і легким для вивчення. Він часто використовується для малих і середніх програм і має мінімалістичний підхід, який дозволяє розробникам швидко створювати програми. Flask відомий своєю гнучкістю та здатністю легко інтегруватися з іншими бібліотеками. Ця гнучкість також може бути слабкою стороною, оскільки вимагає від розробника приймати більше рішень щодо того, як структурувати свою програму.

Django, з іншого боку, — це фреймворк, який має багато вбудованих функцій та інструментів, що робить його популярним вибором для великих і складних web-додатків. Django відомий своєю надійністю та безпекою, що робить його чудовим фреймворком для проектів, які обробляють конфіденційну інформацію. Однак наявність великої кількісті вбудованих функцій Django іноді ускладнюють налаштування або інтеграцію з іншими бібліотеками та вимагають значних навичок від розробника.

FastAPI — це нова бібліотека, яка швидко набирає популярності завдяки своїй швидкості та продуктивності. FastAPI також відомий своєю інтуїтивно зрозумілою документацією, що генерується автоматично по мірі написання коду. Через відносну новизну, FastAPI має меншу спільноту та кількість ресурсів для підтримки.

Для розробки web-застосунку у даній кваліфікаційній роботі, було прийнято рішення використати фреймворк FastAPI, враховуючи його легкість та новизну.

Розробники даного фреймворку відзначають його наступні особливості [25]:

- Швидкість: продуктивність на рівні із NodeJS та Go.
- Прискорення написання коду на 200-300%.
- Зменшення кількості людських помилок на 40%.
- Інтуїтивність: підтримка редакторів та авто-заповнення.
- Легкість: створений для легкого навчання та швидкого читання документації.
- Короткий: зменшення повторів коду.
- Наповненість: у результаті роботи — повністю готовий застосунок.
- Заснований на відкритих стандартах: OpenAPI та JSON Schema.

Як вже було зазначено, FastAPI — це сучасний швидкий фреймворк для створення API за допомогою Python 3.7+ на основі функції підказок і анотацій типів.

Він дозволяє створювати високопродуктивні асинхронні API, використовуючи потужність фреймворку Starlette ASGI, який побудований на основі популярної бібліотеки asyncio.

Однією з головних сильних сторін FastAPI є його автоматична перевірка даних і генерація документації на основі підказок типу, наданих розробником. FastAPI автоматично генерує документи OpenAPI та JSON Schema, що дозволяє дуже легко зрозуміти API без необхідності читати документацію.

FastAPI також забезпечує підтримку веб-сокетів, фонових завдань і ін’єкції залежностей. Він є чудовим вибором для розробки мікросервісів, а завдяки швидкості та продуктивності він добре підходить для створення API з високим трафіком і малою затримкою. З точки зору недоліків, FastAPI є відносно новою бібліотекою, яка все ще розвивається, що може призвести до певних труднощі при виникненні нестандартних проблем.

Наступним важливим елементом для реалізації web-застосунку є база даних. База даних — це організований набір даних, що зберігаються та доступні в електронному вигляді. Це критично важливий компонент розробки сучасного програмного забезпечення, який використовується для ефективного зберігання, організації та керування великими обсягами даних. Бази даних можна класифікувати різними способами, наприклад на основі їхніх моделей даних. У розробці застосунків, поширеними типами є реляційні, NoSQL та об’єктно-орієнтовані бази даних.

Реляційні бази даних є доволі поширеним типом, вони засновані на реляційній моделі даних. Ця модель даних організовує дані в одну або кілька таблиць, кожна з яких складається з набору рядків і стовпців. Крім того таблиці можуть бути пов’язані між собою. Реляційні бази даних використовують мову структурованих запитів (SQL) для керування даними.

NoSQL бази даних — це новий тип баз даних, який виник через обмеження реляційних баз даних, що проявляються у обробці неструктурованих даних. Вони можуть зберігати та керувати даними в різних форматах, включаючи документи, графи, пари ключ-значення і так далі. Бази даних NoSQL мають високу масштабованість, гнучкість і можуть ефективно обробляти великі обсяги даних.

Об’єктно-орієнтовані бази даних зберігають дані у вигляді об’єктів, вони використовуються для керування складними структурами даних. Об’єктно-орієнтовані бази даних є дуже гнучкими та добре підходять для додатків, які потребують складних зв’язків між даними та їх взаємодії.

Існує велика кількість СУБД, які використовуються для керування базами даних. Деякі з них є комерційними та вимагають оплати за користування, а інші – розповсюджуються безкоштовно. Кожна СУБД має свої сильні та слабкі сторони, а вибір СУБД залежить від конкретних потреб програми.

Для розробки web-застосунку у даній роботі, було обрано PostgreSQL. Це об’єктно-реляційна СУБД з відкритим кодом, яка підтримується спільнотою. В основі даної системи лежить 35 років розробки та покращень, а її функціонал цілком задовольняє потреби web-застосунку, що буде розроблений [26].

Однією з головних сильних сторін PostgreSQL є її гнучкість і масштабованість. Вона може працювати з широким діапазоном робочих навантажень, від невеликих програм до великих систем корпоративного рівня. Вона підтримує велику кількість типів даних, включаючи числові, рядкові дані та дані дати/часу, а також більш розширені типи даних, такі як масиви, JSON і XML. Ця гнучкість дозволяє розробникам створювати налаштовані рішення для баз даних, які можуть відповідати конкретним потребам їхніх додатків.

Ще однією важливою перевагою PostgreSQL є її розширюваність. Вона містить надійну систему розширення, яка дозволяє розробникам додавати спеціальні функції до системи баз даних. Сюди входить усе: від простих користувацьких типів даних до складних повнофункціональних плагінів.

Незважаючи на численні переваги, PostgreSQL не позбавлений недоліків. Однією з головних проблем є його відносно велика складність у вивченні. Це дуже складна система, для повного освоєння якої розробникам може знадобитися певний час. Крім того, незважаючи на широкі можливості налаштування, надмірна кількість функцій може ускладнити керування системою, особливо для невеликих команд розробників без великого досвіду роботи з базами даних. На додаток, PostgreSQL

може працювати повільніше за своїх конкурентів та не підтримуватися певними програмами [27].

У якості клієнтської частини, для розробки та демонстрації результату, буде використовуватися Postman. Це популярна платформа для колективної розробки, тестування та документування API. Вона забезпечує зручний інтерфейс, який дозволяє розробникам легко робити HTTP-запити та переглядати відповіді в графічному форматі. Postman підтримує широкий діапазон типів запитів, включаючи GET, POST, PUT, DELETE тощо, і дозволяє налаштовувати заголовки, параметри та тіло запиту [28].

Окрім надсилання запитів, Postman також надає функції для тестування та налагодження API, наприклад налаштування автоматичних тестів, перевірку відповідей на помилки та створення документації для API. Postman також містить функції для спільної роботи, такі як обмін запитами та наборами тестів з членами команди та контроль версій.

Таким чином, фінальний стек технологій, які будуть використовуватися представлений:

- мовою програмування Python;
- фреймворком FastAPI;
- СУБД PostgreSQL;
- застосунком Postman.

Написання коду буде виконуватися у IDE PyCharm. Це популярне IDE, спеціально розроблене для розробників Python. PyCharm розроблений компанією JetBrains і доступний як у професійній версії, так і в версії спільноти. IDE підтримує широкий спектр веб-фреймворків Python, таких як Django, Flask, Pyramid і FastAPI, і включає такі функції, як інтелектуальне завершення коду, налагодження, тестування та контроль версій. PyCharm також пропонує функції для підвищення продуктивності розробника, такі як перевірка коду, інструменти рефакторингу та шаблони коду [29].

PyCharm відомий своїми потужними можливостями налагодження, що дозволяє розробникам легко знаходити та виправляти помилки у своєму коді. На

додаток до основних функцій, PyCharm пропонує різні плагіни, які розширюють функціональні можливості IDE.

3.3 Опис програмної реалізації web-застосунку

Програма починає свою роботу із файлу «main.py». Код у даному файлі визначає програму FastAPI з кількома маршрутизаторами для різних кінцевих точок API: «post», «user», «auth», «vote» та «otp». Він також додає посередника для спільного використання ресурсів між джерелами (CORS), щоб дозволити запити з будь-якого джерела, з використанням будь-якого методу та заголовку [30]. У кінці файлу, визначається коренева кінцева точка, яка повертає відповідь JSON із «Hello World!!!» повідомленням. Блок коду наведений на рисунку 3.7.

```python
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from .routers import post, user, auth, vote, otp


app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

app.include_router(post.router)
app.include_router(user.router)
app.include_router(auth.router)
app.include_router(vote.router)
app.include_router(otp.router)


@app.get(path="/")
def root():
    return {"message": "Hello World!!!"}
```

Рисунок 3.7. Код файлу main.py

Маршрутизатори відповідають за додавання кінцевих точок, які визначають типи запитів та корисне навантаження, яке має бути відправлено. Перша операція, яку має зробити користувач – реєстрація. Вона відбувається за допомогою кінцевої точки «/users», яка описана у файлі «user.py». Код визначає маршрутизатор за допомогою FastAPI «APIRouter» для обробки пов'язаних з користувачем кінцевих точок. У даному файлі визначено дві кінцеві точки, функціонал яких описується у відповідних функціях.

Кінцева точка «/users» використовується для створення нового користувача. Вона приймає об'єкт «schemas.UserCreate» в тілі запиту, який потім перевіряється на відповідність. При правильно відправленому тілі запиту, програма дістає пароль, отриманий від користувача, який потім хешується та додається до бази даних. Якщо користувач із такою електронною адресою вже зареєстрований, то сервер повертає клієнту «HTTPException» із кодом 400. У разі успішної реєстрації, сервер повертає інформацію про створеного користувача. Код наведено на рисунках 3.8 та 3.9.

```python
router = APIRouter(
    prefix="/users",
    tags=["Users"]
)


@router.post(path="/", status_code=status.HTTP_201_CREATED, response_model=schemas.UserOut)
def create_user(user: schemas.UserCreate, db: Session = Depends(get_db)):

    hashed_password = utils.create_hash(user.password)
    user.password = hashed_password

    new_user = models.User(**user.dict())
    db.add(new_user)
    try:
        db.commit()
    except exc.IntegrityError:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
                            detail="User with this email already exists")

    db.refresh(new_user)

    return new_user
```

Рисунок 3.8. Функція створення користувача

```python
@router.get("/{user_id}", response_model=schemas.UserOut)
def get_user(user_id: int, db: Session = Depends(get_db)):
    user = db.query(models.User).filter(models.User.user_id == user_id).first()

    if not user:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail=f"User with id {user_id} does not exist")

    return user
```

Рисунок 3.9. Функція отримання інформації користувача

Основна ідея даної роботи полягає у додаванні можливості використання другого фактору для підвищення надійності захисту облікового запису. Як вже було зазначено у підрозділі 3.1, для ввімкнення другого фактору буде використовуватися кінцева точка «/otp», функціонал якої описаний у файлі «otp.py». Код у даному файлі визначає маршрутизатор для обробки пов'язаних з OTP кінцевих точок. Маршрутизатор ініціалізується із префіксом «/otp» і тегом «OTP». У ньому визначено дві функції кінцевої точки.

Перша функція кінцевої точки використовується для генерації нового OTP. Для цього потрібна автентифікація за допомогою JWT, який можна отримати з кінцевої точки «/login», визначеної в іншому маршрутизаторі. Після автентифікації ця кінцева точка генерує новий секрет OTP у кодуванні base32 і URL-адресу для зручного додавання до застосунків, які генерують одноразові паролі. Потім програма зберігає секрет у базі даних, пов'язуючи його з поточним користувачем. Сервер у відповідь повертає клієнту секрет, закодований у base32, URL-адресу автентифікації OTP та URL-адресу кінцевої точки «перевірки», де користувач може ввести свій OTP, щоб перевірити працездатність та увімкнути двофакторну аутентифікацію. Блок коду наведений на рисунку 3.10.

Друга функція кінцевої точки використовується для перевірки OTP, введеного користувачем. Ця кінцева точка також вимагає автентифікації за допомогою JWT-токена. Після автентифікації, секрет користувача отримується з бази даних за допомогою ідентифікатора користувача, який міститься у токені. Сервер генерує OTP із отриманими значеннями. Одноразовий пароль, введений користувачем, перевіряється на відповідність OTP, який був згенерований сервером, і якщо вони

однакові, функція двофакторної аутентифікації вмикається для даного облікового запису, про що робиться помітка у базі даних.

```python
@router.post(path="/generate", status_code=status.HTTP_201_CREATED)
def generate_otp(current_user = Depends(get_current_user), db: Session = Depends(get_db)):
    otp_base32 = pyotp.random_base32()
    user_query = db.query(models.User).filter(models.User.user_id == current_user.user_id)
    user = user_query.first()
    otp_auth_url = pyotp.totp.TOTP(otp_base32).provisioning_uri(name=user.email, issuer_name="FASTAPI")
    setattr(user, "otp_base32", otp_base32)

    if user.otp_enabled:
        setattr(user, "otp_enabled", False)
    db.commit()

    return {
        "base32": otp_base32,
        "otpauth_url": otp_auth_url,
        "to_verify": "/otp/verify"
    }
```

Рисунок 3.10. Функція генерації секрету TOTP

У разі успіху, сервер повертає відповідь, яка містить повідомлення про успіх. Якщо OTP не збігається або користувач не існує, функція повертає «HTTPException» із кодом 400 і повідомленням про помилку. Блок коду на рисунку 3.11.

```python
@router.post(path="/verify")
def verify_otp(payload: schemas.OTPValidate, current_user = Depends(get_current_user), db: Session = Depends(get_db)):
    message = "Token is invalid or user doesn't exist"
    user_query = db.query(models.User).filter(models.User.user_id == current_user.user_id)
    user = user_query.first()
    current_otp = pyotp.TOTP(user.otp_base32)

    if not current_otp.verify(payload.otp):
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail=message)

    user.otp_enabled = True
    db.commit()

    return {
        "otp_enabled": True,
        "user_id": user.user_id,
        "user_email": user.email
    }
```

Рисунок 3.11. Функція попередньої перевірки OTP

За вхід до облікового запису відповідає кінцева точка «/login», функціонал якої описаний у файлі «auth.py». Код у даному файлі визначає маршрутизатор, який використовується для обробки кінцевих точок API, пов’язаних з автентифікацією користувачів. Маршрутизатор має одну кінцеву точку під назвою «/login», яка обробляє запит HTTP POST з обліковими даними користувача як вхідними параметрами. Кінцева точка повертає відповідь у форматі JSON із JWT, який можна використовувати для автентифікації.

Спочатку виконується запит до бази даних, щоб знайти користувача з вказаною електронною поштою. Якщо користувач не існує, виникає «HTTPException» із кодом «403 Forbidden» і повідомленням про помилку.

Якщо для облікового запису ввімкнена двофакторна аутентифікація, програма перевіряє, чи присутній у запиті заголовок X-Auth-2FA. Якщо його немає, виникає «HTTPException» із кодом «206 Partial Content» і повідомленням про помилку. Якщо заголовок присутній, відбувається перевірка OTP, який був наданий у заголовку, з одноразовим паролем, згенерованим із секретного ключа користувача. Якщо OTP не збігаються, виникає «HTTPException» із кодом «400 Bad Request» і повідомленням про помилку.

Якщо користувач існує та OTP (якщо ввімкнено) успішно перевірений, кінцева точка перевіряє пароль користувача, викликаючи функцію «verify_pwd», визначену в модулі «utils». Якщо пароль недійсний, сервер викликає «HTTPException» із кодом «403 Forbidden» і повідомленням про помилку.

Якщо облікові дані користувача дійсні, системою генерується токен за допомогою функції «create_access_token», визначеної в модулі «oauth2». У відповідь користувачу повертається JWT разом із типом токена. У подальшому токен використовується для автентифікації користувача при наступних запитах. Описаний блок коду представлений на рисунку 3.12.

Код, розміщений у файлі «oauth2.py», визначає функції та змінні, пов’язані з автентифікацією за допомогою JWT у програмі FastAPI. «Oauth2_scheme» — це екземпляр класу «OAuth2PasswordBearer» із FastAPI, який використовується для автентифікації користувачів і отримання токенів. «SECRET_KEY», «ALGORITHM» і

«ACCESS_TOKEN_EXPIRE_MINUTES» — це змінні, які використовуються для створення та перевірки JWT.

```python
@router.post(path="/login", response_model=schemas.Token)
def login(user_credentials: OAuth2PasswordRequestForm = Depends(),
          db: Session = Depends(database.get_db),
          x_auth_2fa: str | None = Header(default=None)):
    user = db.query(models.User).filter(models.User.email == user_credentials.username).first()
    if not user:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="Invalid credentials")

    if user.otp_enabled:
        if x_auth_2fa is None:
            raise HTTPException(status_code=status.HTTP_206_PARTIAL_CONTENT, detail="X-Auth-2FA header is missing")

        totp = pyotp.TOTP(user.otp_base32)
        if not totp.verify(x_auth_2fa):
            raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Invalid credentials")

    if not utils.verify_pwd(user_credentials.password, user.password):
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="Invalid credentials")

    access_token = oauth2.create_access_token(data={"user_id": user.user_id})
    return {"access_token": access_token, "token_type": "bearer"}
```

Рисунок 3.12. Функція входу у систему

«Create_access_token» — це функція, яка приймає словник із даними, які будуть записані у токен, та створює JWT, який містить ці дані разом із часом закінчення терміну дії, який визначено сервером, код наведений на рисунку 3.13.

```python
def create_access_token(data: dict):

    to_encode = data.copy()
    expire = datetime.utcnow() + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    to_encode.update({"exp": expire})

    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)

    return encoded_jwt
```

Рисунок 3.13. Функція генерації токена

«Verify_access_token» — це функція, яка приймає JWT і перевіряє його, декодуючи його та перевіряючи час закінчення терміну дії. Якщо JWT дійсний,

функція повертає об’єкт «TokenData», що містить ідентифікатор користувача. Якщо JWT недійсний, виникає виняток. Код наведений на рисунку 3.14:

```
def verify_access_token(token: str, credentials_exception):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        user_id: str = payload.get("user_id")

        if not user_id:
            raise credentials_exception
        token_data = schemas.TokenData(id=user_id)
    except JWTError:
        raise credentials_exception

    return token_data
```

Рисунок 3.14. Функція перевірки токена

«Get_current_user» — це функція, яка приймає токен та сеанс бази даних і використовує «verify_access_token» для отримання ідентифікатора користувача із JWT. Потім функція отримує користувача з бази даних за допомогою цього ідентифікатора та повертає його. Якщо токен недійсний, виникає виняток. Код наведений на рисунку 3.15:

```
def get_current_user(token: str = Depends(oauth2_scheme), db: Session = Depends(database.get_db)):
    credentials_exception = HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
                                          detail="Could not validate credentials",
                                          headers={"WWW_Authenticate": "Bearer"})

    token_data = verify_access_token(token, credentials_exception)

    user = db.query(models.User).filter(models.User.user_id == token_data.id).first()

    return user
```

Рисунок 3.15. Функція отримання поточного користувача

Фінальна файлова структура програми наведена на рисунку 3.16.

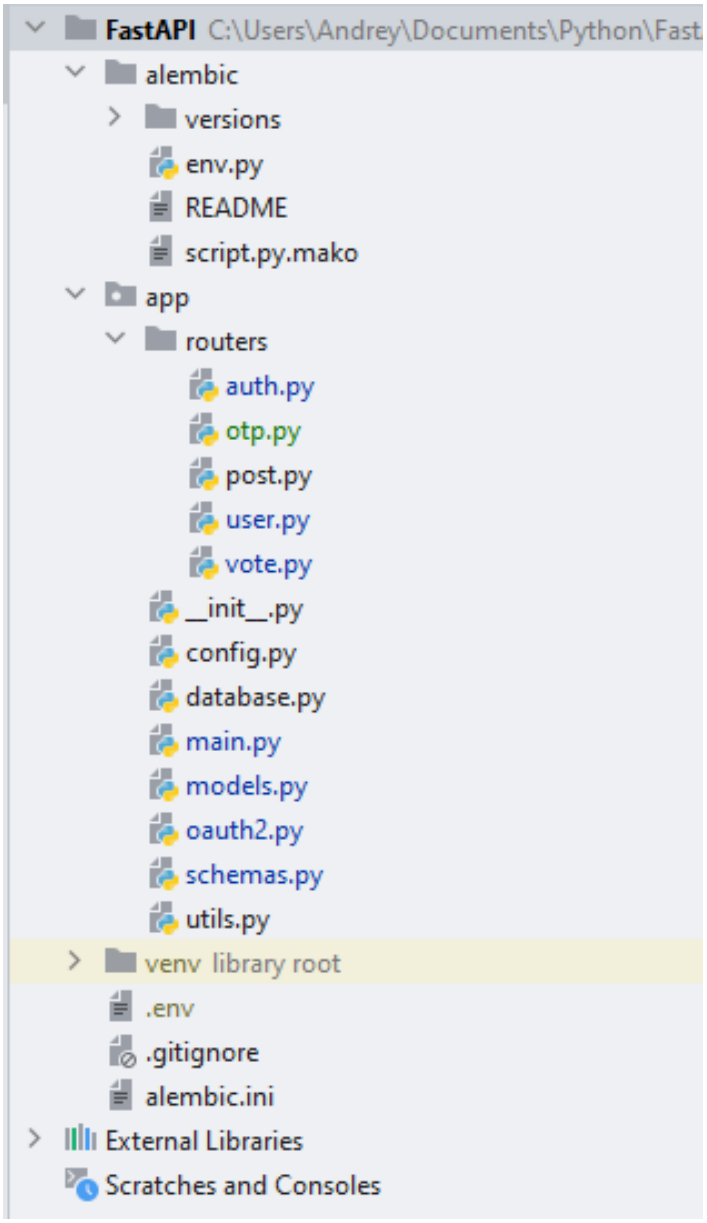

Рисунок 3.16. Файлова струкутра програми

Увійшовши у систему, користувач може використовувати інші функції програми, які мають відношення до постів, такі як створення, видалення, голосування тощо. Код даних частин програми описаний у інших файлах та може бути переглянутий у додатку. Аутентифікація користувача у всіх операціях відбувається за допомогою JWT. Його генерація, перевірка та використання у операціях були описані вище у даному розділі.

3.4 Тестовий приклад

Для демонстрації роботи застосунку буде використовуватися програмне забезпечення Postman та браузер Firefox.

При переході на адресу, на якій працює веб-сервер, за допомогою браузера, відкривається сторінка із привітанням у форматі JSON (рис. 3.17). Таким чином, можна стверджувати, що коренева кінцева точка працює як треба.

Рисунок 3.17. Результат запиту до кореневої кінцевої точки

Як вже зазначалося раніше, однією із особливостей фреймворку FastAPI, який використовувався при розробці, є автоматична генерація документації. При чому генерація відбувається із використанням двох різних інструментів (Swagger UI та ReDoc). Дані інструменти не тільки відображають кінцеві точки, а і надають можливості відправляти запити прямо із веб-браузера. Приклад згенерованої документації наведено на рисунку 3.18.

При відправленні запитів з даної сторінки, можна легко експериментувати з різними типами вхідних даних, отримувати відповіді прямо на даній сторінці у режимі реального часу та, таким чином, тестувати застосунок.

Дана функція є дуже зручною як з точки зору розробника, так і з точки зору користувача, якому не потрібно використовувати сторонні програми для відправлення запитів.

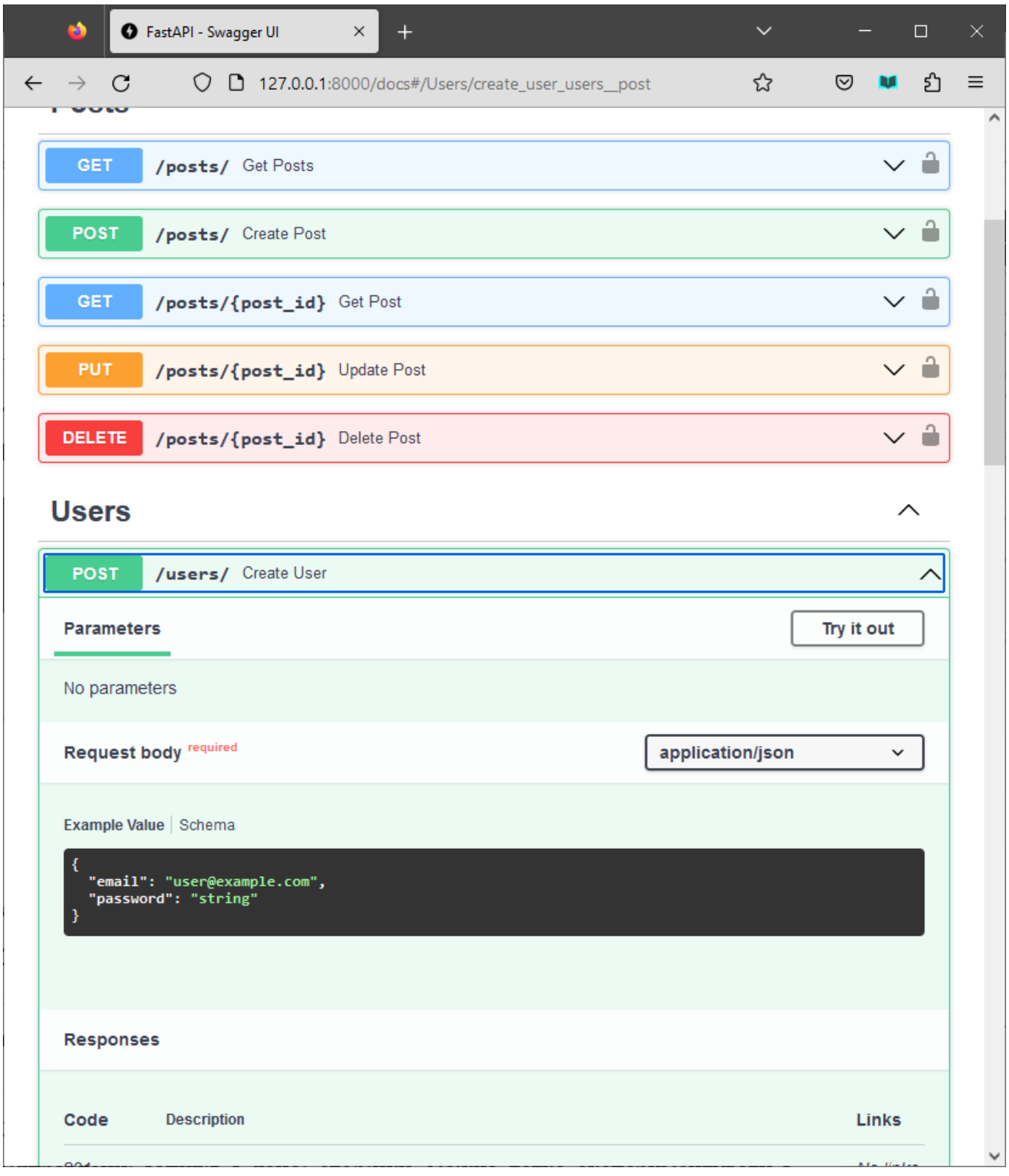

Рисунок 3.18 Документація, згенеровано за допомогою Swagger UI

Першим етапом роботи із застосунком є реєстрація користувача. Вона відбувається за допомогою POST запиту до кінцевої точки «/users». Тіло запиту має містити електронну пошту користувача, що буде використана у якості логіна та

пароль (рис. 3.19). У відповідь сервер надсилає повідомлення, яке містить id користувача, електронну адресу та дату створення облікового запису.

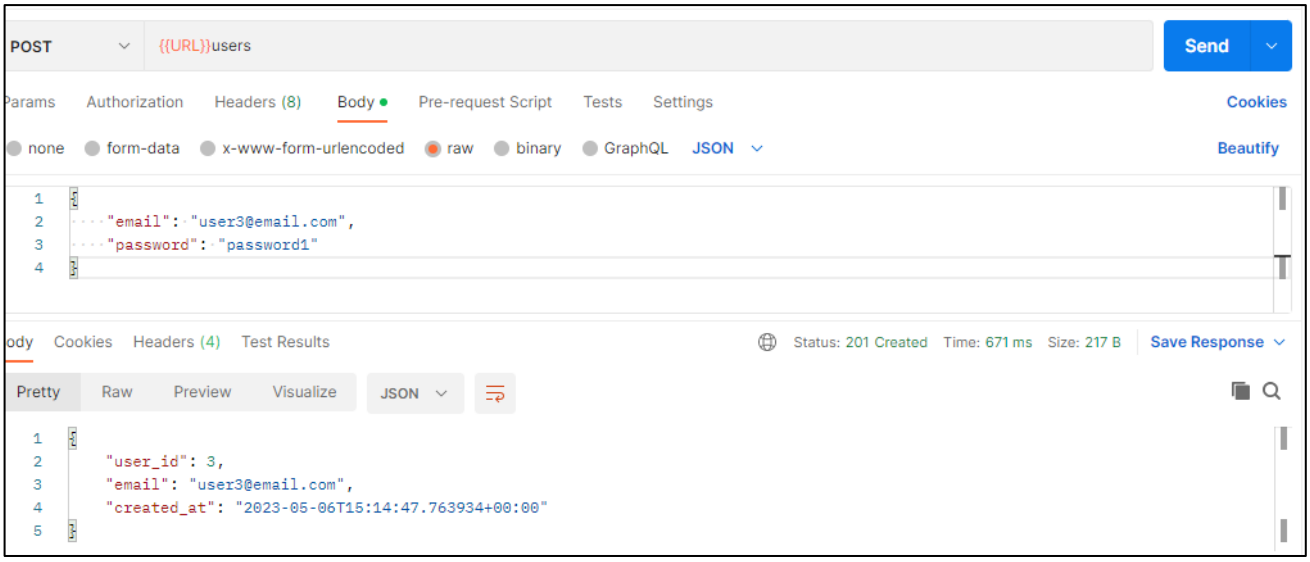

Рисунок 3.19. Реєстрація нового користувача

Після створення облікового запису, користувач може зайти у систему зі своїми обліковими даними. Це можна зробити за допомогою кінцевої точки «/login» (рис. 3.20). У тілі запиту мають міститися облікові дані користувача.

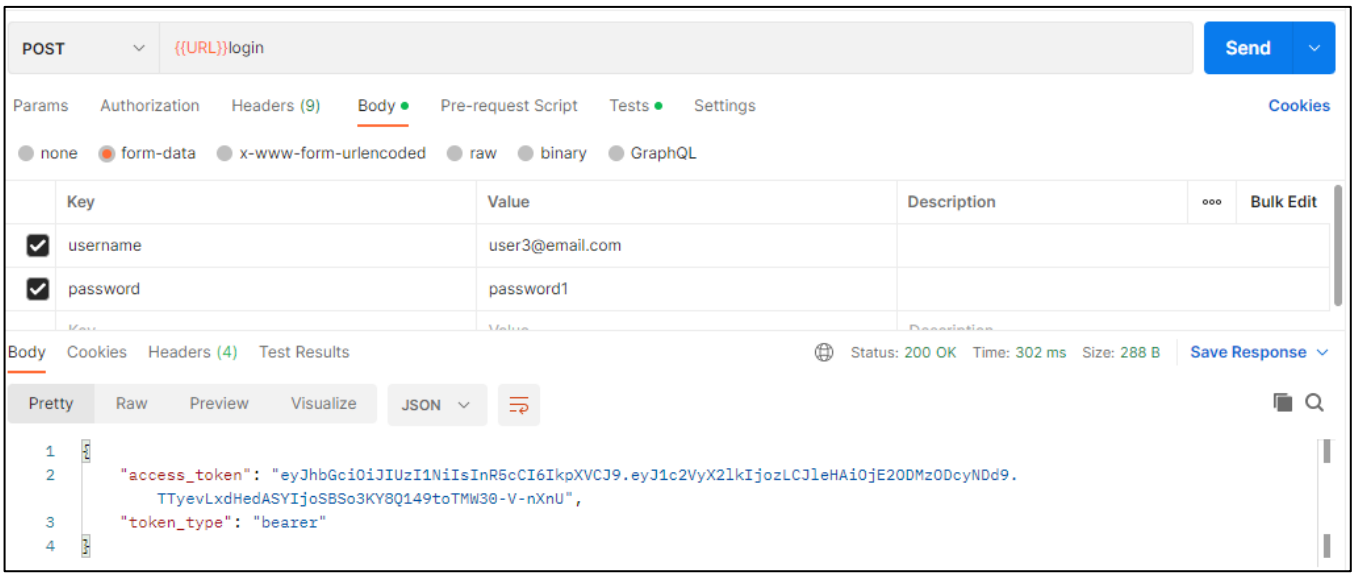

Рисунок 3.20. Вхід до системи

У відповідь сервер повертає JWT, який буде використовуватися для подальшої аутентифікації. Вміст токену зображений на рисунку 3.21. Основним елементом є

корисне навантаження, яке містить id користувача та час закінчення терміну дії токена.

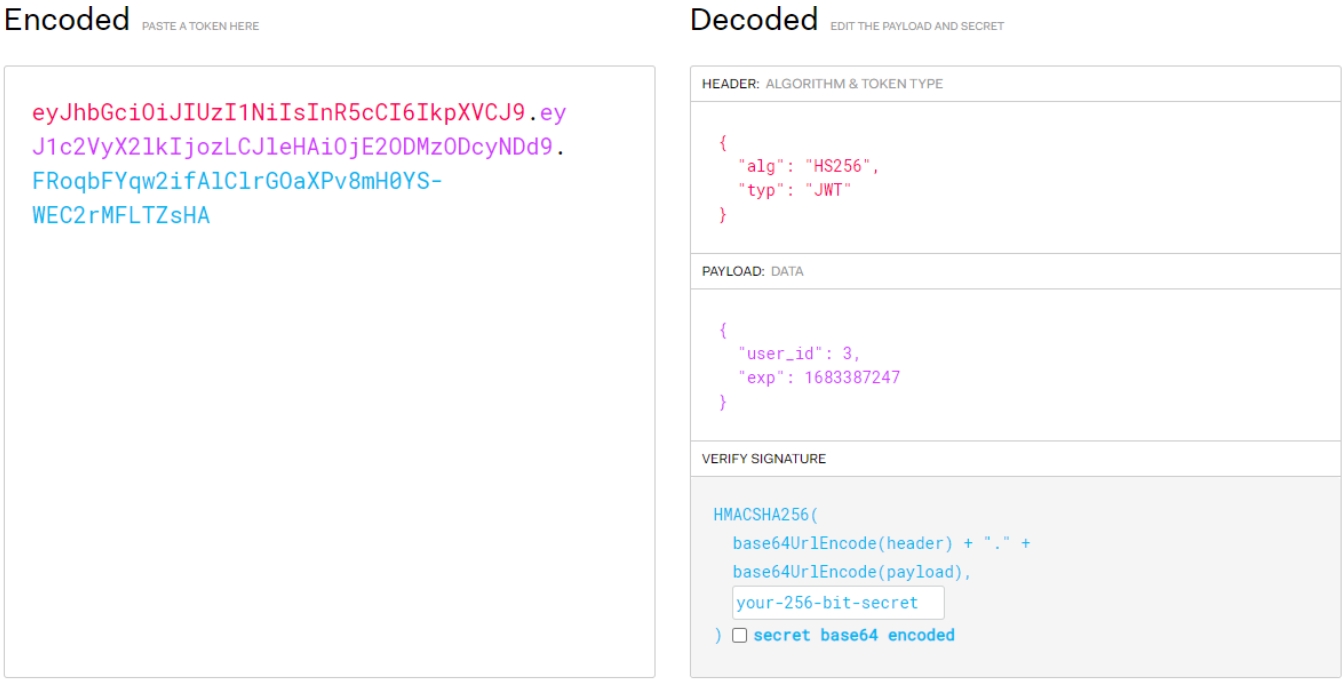

Рисунок 3.21. Вміст JWT

Після аутентифікації, користувач може створювати певні пости. Крім цього користувач може модифікувати та видаляти власні пости. Якщо вхід у систему було виконано, користувач може переглядати загальний список постів, використовуючи кінцеву точку «/posts». Існує можливість обмеження кількості отриманих постів за допомогою параметру «limit». Інші параметри запиту дозволяють пропустити певну кількість записів та виконати пошук за змістом. Відправлення GET-запиту до кінцевої точки «/posts» із зазначенням id посту, дозволяє отримати лише пост із вказаним id.

Для демонстрації можливості створення постів, було відправлено POST-запит до кінцевої точки «/posts». Запит має містити заголовок «Authorization» зі значенням JWT та тіло зі значеннями назви посту, змісту та опціонально булеве значення, яке показує, чи опублікований пост. У відповідь сервер повертає повідомлення із кодом 201, яке містить вміст посту, id, дату створення та дату створення (рис.3.22).

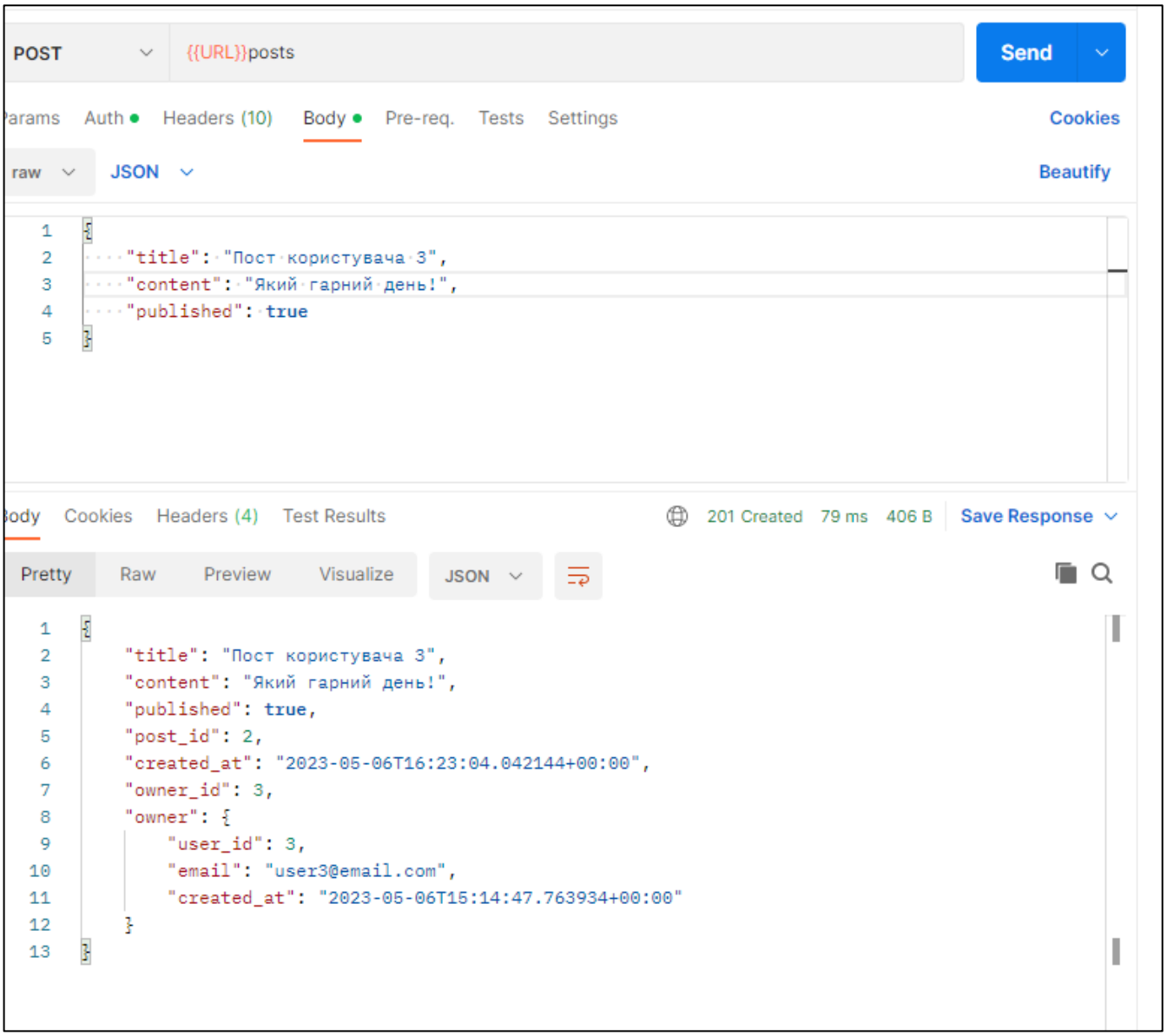

Рисунок 3.22. Створення нового посту

Увімкнення додаткового захисту облікового запису шляхом додавання другого фактору у вигляді TOTP відбувається наступним чином: користувач відправляє POST-запит до кінцевої точки «/otp/generate». У відповідь сервер повертає повідомлення із згенерованим секретом (рис. 3.23). Крім секрету, повідомлення містить посилання, яке може бути використано для генерації QR-коду. Даний код, в свою чергу, може бути використаний для додавання інформації, потрібної для створення паролів, до застосунків для двофакторної автентифікації на смартфонах чи комп’ютерах.

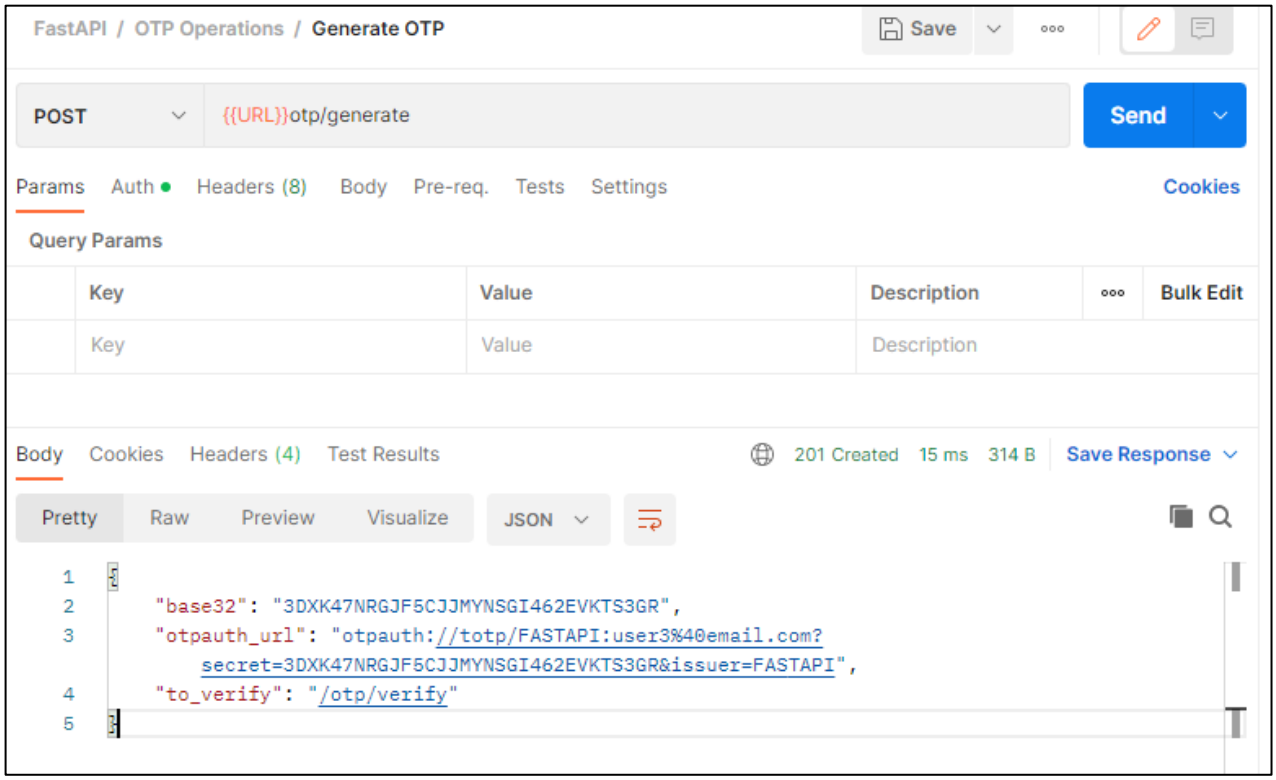

Рисунок 3.23. Запит для генерації секрету

Для увімкнення двофакторної аутентифікації, користувачу необхідно відправити запит, який містить актуальний OTP до кінцевої точки «/otp/verify» (рис. 3.24).

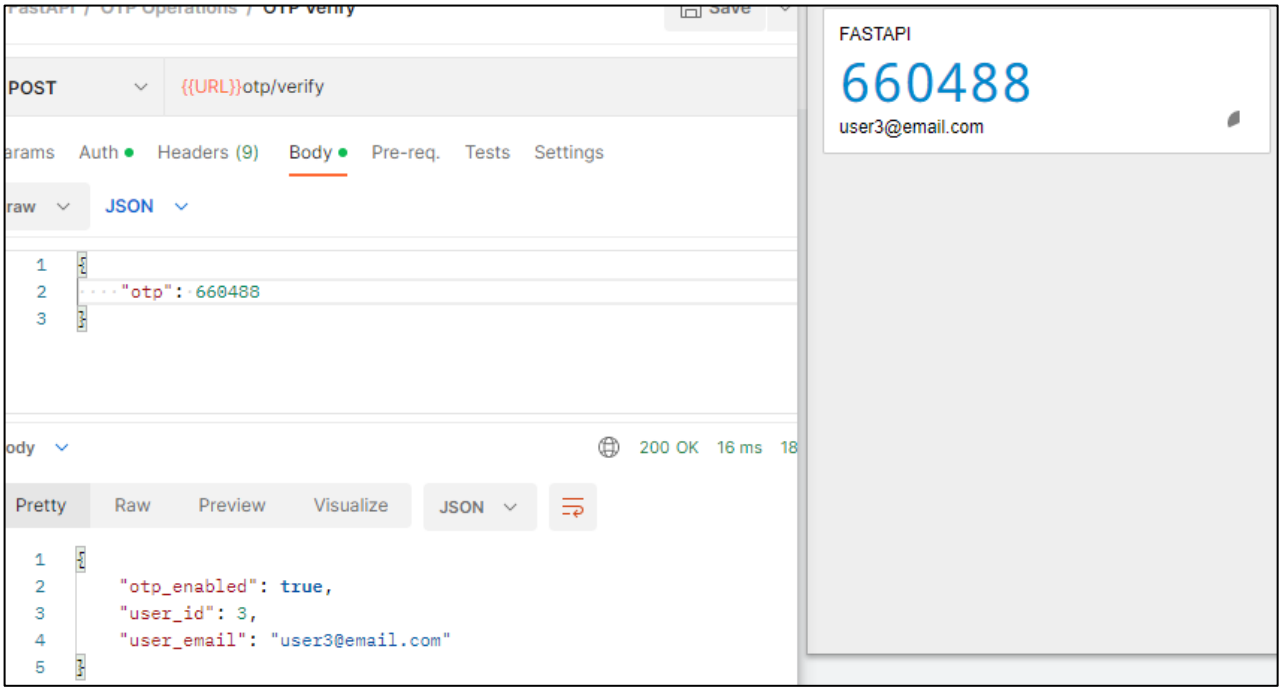

Рисунок 3.24. Підтвердження додавання другого фактору

При наступному вході у систему, запит, крім облікових даних користувача, має містити заголовок «X-Auth-2FA» із поточним значенням TOTP, Якщо двофакторна аутентифікація ввімкнена для облікового запису користувача, а заголовок не зазначений у запиті, сервер поверне повідомлення про помилку. Якщо двофакторна аутентифікація вимкнена, вхід відбуватиметься лише за обліковими даними. Якщо актуальний одноразовий пароль та облікові дані були надані у запиті – аутентифікація буде успішною. Приклад входу у систему із увімкненою двофакторною аутентифікацією зображений на рисунку 3.25.

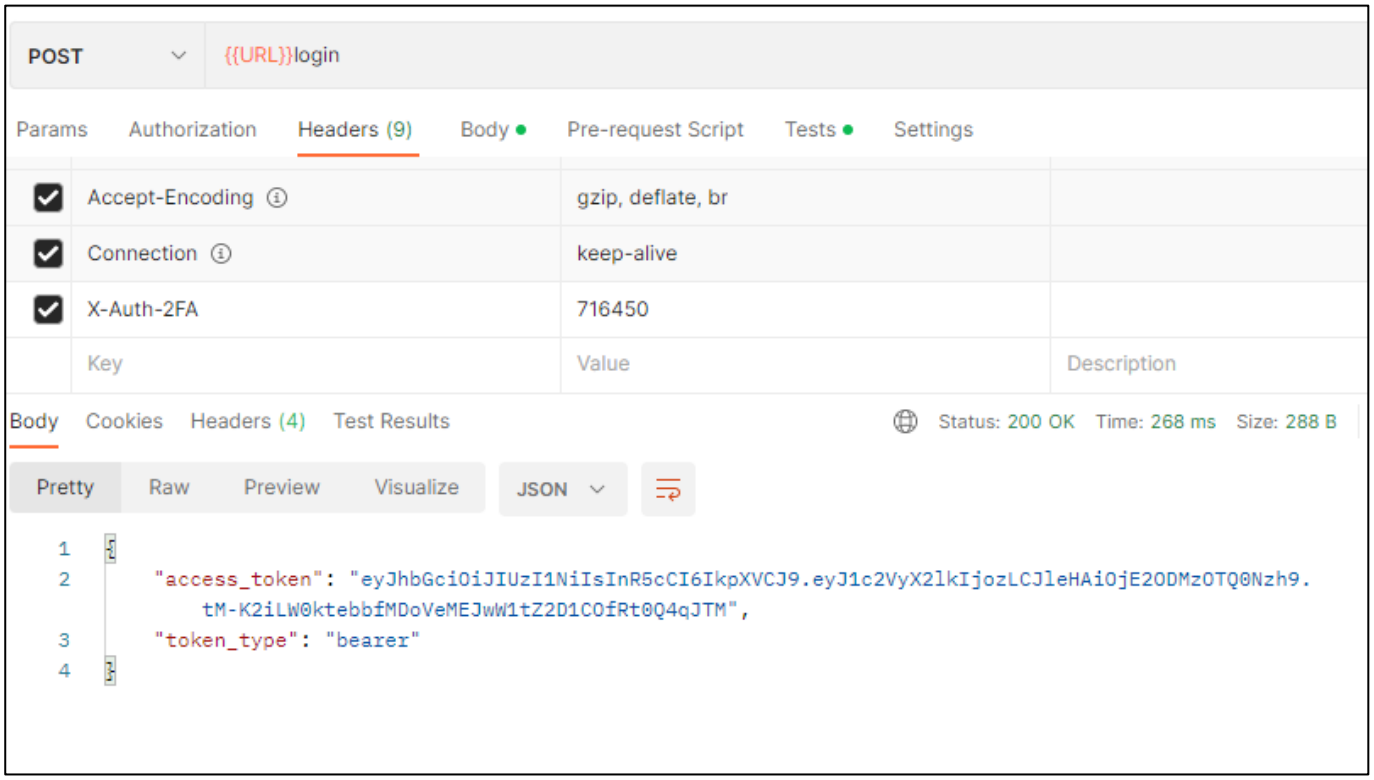

Рисунок 3.25. Вхід у систему із увімкненою двофакторною аутентифікацією

Тепер користувач може користуватися іншим функціями застосунку, а його обліковий запис захищений додатковим фактором, у якості якого використовується одноразовий пароль, а саме TOTP. Для генерації OTP може бути використаний програмний застосунок на смартфоні чи комп’ютері або токен. У відкритому доступі наявна велика кількість необхідних рішень.

Висновки за розділом 3

У даному розділі була представлена архітектура web-застосунку, який було розроблено у даній кваліфікаційній роботі, описані технології, які використовувалися для його створення, певні елементи коду програми та продемонстрована робота застосунку.

Створений web-додаток дозволяє користувачам реєструватися у системі за допомогою електронної пошти та пароля. Після реєстрації, користувач може створювати, модифікувати, видаляти та переглядати пости за допомогою кінцевої точки «/posts». Крім цього, користувач може голосувати за пости за допомогою кінцевої точки «/vote».

Для аутентифікації у застосунку використовується JWT, який містить у своєму корисному навантажені ідентифікатор користувача та час закінчення терміну дії. JWT надається клієнту після успішної аутентифікації

Основним елементом роботи є впровадження можливості увімкнення двофакторної аутентифікації. Для цього використовуються кінцеві точки «/otp/generate» та «/otp/verify». Перша відповідає за генерацію секрету для TOTP, а друга – за підтвердження працездатності та остаточне ввімкнення двофакторної аутентифікації. Після цього, для входу у систему, необхідно додати заголовок «X-Auth-2FA» із одноразовим паролем.

Для створення застосунку було використано наступні технології: Python, FastAPI, PostgreSQL та Postman.

Python — це мова програмування високого рівня, яка широко використовується у веб-розробці, наукових обчисленнях, аналізі даних, штучному інтелекті тощо. Вона відома своєю читабельністю та простотою використання, а також великою та активною спільнотою розробників.

FastAPI — це сучасний швидкий веб-фреймворк для створення API за допомогою Python. Його розроблено таким чином, щоб він був простим у використанні, швидким при розробці та високопродуктивним. Він поставляється з

вбудованою підтримкою сучасних функцій Python, а також може бути налаштований завдяки багатьом стороннім бібліотекам і плагінам.

PostgreSQL — це потужна об’єктно-реляційна СУБД із відкритим вихідним кодом, відома своєю надійністю, масштабованістю та стійкістю. Вона широко використовується в багатьох сферах, включаючи веб-розробку, аналітику даних та машинне навчання.

Postman — це популярний інструмент для тестування API, який забезпечує інтуїтивно зрозумілий і простий у використанні інтерфейс для надсилання HTTP-запитів, перегляду відповідей і налагодження викликів API.

Для демонстрації роботи API було створено обліковий запис користувача, додано новий пост від його імені, згенеровано OTP-секрет та увімкнено двофакторну аутентифікацію. Після цього було продемонстровано вхід до системи із використанням основних облікових даних та одноразового пароля. Таким чином, архітектура web-застосунку була описана, а працездатність продемонстрована.

ВИСНОВКИ

У кваліфікаційній роботі було розроблено web-застосунок на базі архітектури REST з можливістю двофакторної аутентифікації користувача та продемонстровано його функціонал.

У першій частині кваліфікаційної роботи було досліджено принципи функціонування web-застосунків. Дослідження було зроблено за рахунок огляду монолітної та мікросервісної архітектури побудови програмного забезпечення, а також різних видів архітектур для створення web-застосунків. Крім цього, окремо було розглянуто принципи архітектури REST та проведено порівняння даної архітектури з іншими технологіями.

У другій частині кваліфікаційної роботи було проведено аналіз механізмів захисту web-застосунків. Були розглянуті механізми автентифікації для web-застосунків, такі як: базова аутентифікація, аутентифікація із використанням сесій, дайджест аутентифікація, аутентифікація із використанням токенів і використання протоколу OpenID для аутентифікації. Крім того були розглянуті алгоритми генерації одноразових паролів TOTP та HOTP у контексті можливості їхнього використання у якості другого фактору. Окрема увага була приділена загрозам функціонуванню web-застосунків.

У третій частині, на основі опрацьованої у попередніх двох частинах інформації, було розроблено архітектуру web-застосунку, описано кінцеві точки та їх функціонал. Було зроблено огляд технологій та програмних засобів, які використовувалися при розробці, такі як Python, FastAPI, PostgreSQL та Postman. Було описано програмний код ключових елементів застосунку та продемонстровано його функціонал, шляхом створення нового користувача, входу у систему з використанням облікових даних та увімкнення двофакторної аутентифікації для облікового запису.

Виходячи із поставленої мети кваліфікаційної роботи були виконані наступні завдання:

- досліджено принципи функціонування web-застосунків;
- проведено аналіз механізмів захисту web-застосунків;

- побудовано архітектуру web-застосунку із можливістю двофакторної аутентифікації;
- реалізовано web-застосунок із можливістю двофакторної аутентифікації.

Всі завдання було виконано в повному обсязі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Number of smartphone mobile network subscriptions worldwide from 2016 to 2022, with forecasts from 2023 to 2028 [Електронний ресурс]. – Режим доступу: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

2. Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030 [Електронний ресурс]. – Режим доступу: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

3. APIs Have Taken Over Software Development [Електронний ресурс]. – Режим доступу: https://nordicapis.com/apis-have-taken-over-software-development/

4. API Security Trends [Електронний ресурс]. – Режим доступу: https://salt.security/api-security-trends

5. 91% of organizations had an API security incident last year [Електронний ресурс]. – Режим доступу: https://www.securitymagazine.com/articles/94509-of-organizations-had-api-security-incident-last-year

6. Software architecture [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Software_architecture

7. АРХІТЕКТУРА — ЕТИМОЛОГІЯ [Електронний ресурс]. – Режим доступу: https://goroh.pp.ua/%D0%95%D1%82%D0%B8%D0%BC%D0%BE%D0%BB%D0%BE%D0%B3%D1%96%D1%8F/%D0%B0%D1%80%D1%85%D1%96%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0

8. Microservices vs monolith: Which architecture is the best choice for your business? [Електронний ресурс]. – Режим доступу: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/

9. Monolithic vs Microservices architecture [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/monolithic-vs-microservices-architecture/

10. Intro to APIs: History of APIs [Електронний ресурс]. – Режим доступу: https://blog.postman.com/intro-to-apis-history-of-apis/

11. Intro to APIs: What Is an API? [Електронний ресурс]. – Режим доступу: https://blog.postman.com/intro-to-apis-what-is-an-api/

12. 2022 State of the API Report [Електронний ресурс]. – Режим доступу: https://www.postman.com/state-of-api/api-technologies/#api-technologies

13. Comparing API Architectural Styles: SOAP vs REST vs GraphQL vs RPC [Електронний ресурс]. – Режим доступу: https://www.altexsoft.com/blog/soap-vs-rest-vs-graphql-vs-rpc/

14. FAQ | GraphQL [Електронний ресурс]. – Режим доступу: https://graphql.org/faq/

15. Rundown on Top APIs: GraphQL vs. REST vs. gRPC vs. SOAP [Електронний ресурс]. – Режим доступу: https://www.velvetech.com/blog/graphql-vs-rest-choose-api/

16. gRPC vs. REST: Getting Started With the Best API Protocol [Електронний ресурс]. – Режим доступу: https://www.toptal.com/grpc/grpc-vs-rest-api

17. The History of REST APIs [Електронний ресурс]. – Режим доступу: https://blog.readme.com/the-history-of-rest-apis/

18. REST [Електронний ресурс]. – Режим доступу: https://www.aloi.io/api/rest/

19. Uniform Interface [Електронний ресурс]. – Режим доступу: https://restapilinks.com/uniform-interface/

20. What Is Identification, Authentication and Authorization? [Електронний ресурс]. – Режим доступу: https://www.notarize.com/blog/what-is-identification-authentication-and-authorization

21. Web Authentication Methods Compared [Електронний ресурс]. – Режим доступу: https://testdriven.io/blog/web-authentication-methods/

22. HOTP vs TOTP: What’s the Difference? [Електронний ресурс]. – Режим доступу: https://rublon.com/blog/hotp-totp-difference/

23. Guide to Basic API Security Best Practices [Електронний ресурс]. – Режим доступу: https://www.twilio.com/blog/basic-api-security-guide

24. A Quick Look at The OWASP API Security Top 10 [Електронний ресурс]. – Режим доступу: https://www.twilio.com/blog/owasp-api-security-top-10-overview

25. FastAPI [Електронний ресурс]. – Режим доступу: https://fastapi.tiangolo.com/

26. PostgreSQL: The World's Most Advanced Open Source Relational Database [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/

27. What is PostgreSQL? Introduction, Advantages & Disadvantages [Електронний ресурс]. – Режим доступу: https://www.guru99.com/introduction-postgresql.html

28. Знайомтесь, Postman - must have для QA [Електронний ресурс]. – Режим доступу: https://qagroup.com.ua/publications/znajomtes-postman-must-have-dlia-qa/

29. What is PyCharm? Features, Advantages & Disadvantages [Електронний ресурс]. – Режим доступу: https://hackr.io/blog/what-is-pycharm

30. CORS (Cross-Origin Resource Sharing) [Електронний ресурс]. – Режим доступу: https://fastapi.tiangolo.com/tutorial/cors/

ДОДАТОК А

Вміст файлу post.py

```
from fastapi import Response, status, HTTPException, Depends, APIRouter
from .. import models, schemas, oauth2
from sqlalchemy.orm import Session
from sqlalchemy import func
from ..database import get_db
from typing import List

router = APIRouter(prefix="/posts", tags=["Posts"])


@router.get(path="/", response_model=List[schemas.PostOut])
def get_posts(db: Session = Depends(get_db),
              current_user: models.User = Depends(oauth2.get_current_user),
              limit: int = 10,
              skip: int = 0,
              search: None | str = ""):
    posts = db.query(models.Post,
func.count(models.Vote.post_id).label("votes"))\
        .join(models.Vote, models.Vote.post_id == models.Post.post_id,
isouter=True)\

.group_by(models.Post.post_id).filter(models.Post.title.contains(search)).lim
it(limit).offset(skip).all()

    return posts


@router.post(path="/", status_code=status.HTTP_201_CREATED,
response_model=schemas.Post)
def create_post(post: schemas.PostCreate,                    # Extract request
body, convert to Post
                db: Session = Depends(get_db),
                current_user: models.User =
Depends(oauth2.get_current_user)):

    new_post = models.Post(owner_id = current_user.user_id, **post.dict())
    db.add(new_post)
    db.commit()
    db.refresh(new_post)

    return new_post


@router.get(path="/{post_id}", response_model=schemas.PostOut)
def get_post(post_id: int,
```

```
            db: Session = Depends(get_db),
            current_user: models.User = Depends(oauth2.get_current_user)):
    post = db.query(models.Post,
func.count(models.Vote.post_id).label("votes"))\
        .join(models.Vote, models.Vote.post_id == models.Post.post_id,
isouter=True)\
        .group_by(models.Post.post_id).filter(models.Post.post_id ==
post_id).first()

    if not post:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"post with id {post_id} was not found")

    return post


@router.delete(path="/{post_id}")
def delete_post(post_id: int,
                db: Session = Depends(get_db),
                current_user: models.User =
Depends(oauth2.get_current_user)):
    post_query = db.query(models.Post).filter(models.Post.post_id == post_id)
    post = post_query.first()

    if not post:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"post with id {post_id} does not exist")

    if not post.owner_id == current_user.user_id:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
detail="Not authorized to perform requested action")

    post_query.delete(synchronize_session=False)
    db.commit()

    return Response(status_code=status.HTTP_204_NO_CONTENT)


@router.put(path="/{post_id}", response_model=schemas.Post)
def update_post(post_id: int,
                post: schemas.PostCreate,
                db: Session = Depends(get_db),
                current_user: models.User =
Depends(oauth2.get_current_user)):
    post_query = db.query(models.Post).filter(models.Post.post_id == post_id)
    updated_post = post_query.first()

    if not updated_post:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail=f"post with id {post_id} does not exist")
```

```
    if not updated_post.owner_id == current_user.user_id:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
detail="Not authorized to perform requested action")

    post_query.update(post.dict(), synchronize_session=False)
    db.commit()

    return post_query.first()
```

Вміст файлу vote.py

```
from fastapi import status, HTTPException, Depends, APIRouter
from .. import models, schemas, oauth2
from sqlalchemy.orm import Session
from ..database import get_db

router = APIRouter(prefix="/vote", tags=["Vote"])


@router.post(path="/", status_code=status.HTTP_201_CREATED)
def vote(vote: schemas.Vote,
         db: Session = Depends(get_db),
         current_user: models.User = Depends(oauth2.get_current_user)):
    post = db.query(models.Post).filter(models.Post.post_id ==
vote.post_id).first()
    if not post:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail=f"post with id {vote.post_id} does not exist")

    vote_query = db.query(models.Vote).filter(models.Vote.post_id ==
vote.post_id, models.Vote.user_id == current_user.user_id)
    found_vote = vote_query.first()

    if vote.direction == schemas.VoteDirection.up:
        if found_vote:
            raise HTTPException(status_code=status.HTTP_409_CONFLICT,
                                detail=f"user {current_user.user_id} has
already voted on post {vote.post_id}")
        new_vote = models.Vote(post_id=vote.post_id,
user_id=current_user.user_id)
        db.add(new_vote)
        db.commit()
        return {"message": "successfully added vote"}
    else:
        if not found_vote:
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="vote does not exist")
        vote_query.delete(synchronize_session=False)
        db.commit()
        return {"message": "successfully deleted vote"}
```

Вміст файлу models.py

```
from sqlalchemy import Column, Integer, String, Boolean, ForeignKey
from sqlalchemy.sql.expression import text
from sqlalchemy.sql.sqltypes import TIMESTAMP
from sqlalchemy.orm import relationship

from .database import Base


class Post(Base):
    __tablename__ = "posts"

    post_id = Column(Integer, primary_key=True, nullable=False)
    title = Column(String, nullable=False)
    content = Column(String, nullable=False)
    published = Column(Boolean, server_default='TRUE', nullable=False)
    created_at = Column(TIMESTAMP(timezone=True), nullable=False,
server_default=text("now()"))
    owner_id = Column(Integer, ForeignKey("users.user_id",
ondelete="CASCADE"), nullable=False)
    owner = relationship("User")


class User(Base):
    __tablename__ = "users"

    user_id = Column(Integer, primary_key=True, nullable=False)
    email = Column(String, nullable=False, unique=True)
    password = Column(String, nullable=False)
    otp_enabled = Column(Boolean, server_default="FALSE", nullable=False)
    otp_base32 = Column(String, nullable=True)
    created_at = Column(TIMESTAMP(timezone=True), nullable=False,
server_default=text("now()"))


class Vote(Base):
    __tablename__ = "votes"
    user_id = Column(Integer, ForeignKey("users.user_id",
ondelete="CASCADE"), nullable=False, primary_key=True)
    post_id = Column(Integer, ForeignKey("posts.post_id",
ondelete="CASCADE"), nullable=False, primary_key=True)
```

Вміст файлу utils.py

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
```

```
def create_hash(password: str):
    return pwd_context.hash(password)


def verify_pwd(plain_password: str, hashed_password: str):
    return pwd_context.verify(plain_password, hashed_password)
```

ДОДАТОК Б
СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ ЗА ТЕМОЮ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Тези наукових конференцій

Staroselskyi A. Implementation of two-factor authentication in RESTful web-services. / Andrii Staroselskyi, Larysa Myrutenko / VI Міжнародна науково-практична конференція “Проблеми кібербезпеки інформаційно-телекомунікаційних систем” (PCSITS)” 27 квітня 2023 року, Київ, Україна, стр. 45-46.