Міністерство освіти і науки України

Луцький національний технічний університет
(повне найменування закладу вищої освіти)

Факультет комп'ютерних та інформаційних технологій
(повне найменування факультету)

Кафедра комп'ютерної інженерії та кібербезпеки
(повне найменування кафедри)

КВАЛІФІКАЦІЙНА РОБОТА ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

ВЕБ-СЕРВЕР З ФУНКЦІОНАЛОМ ПРИВАТНОСТІ ТА СИСТЕМОЮ АВТОРИЗАЦІЇ ІНТЕГРОВАНОЇ З ЗАХИЩЕНИМ НОСІЄМ

A WEB SERVER WITH PRIVACY FUNCTIONALITY AND AN AUTHORIZATION SYSTEM INTEGRATED WITH A SECURE MEDIUM

спеціальність 123 Комп'ютерна інженерія
(шифр і назва спеціальності)

освітня програма Комп'ютерна інженерія
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи КІ-42
Нагурний Владислав Володимирович

(підпис)

Керівник:
к.т.н., доцент
Костючко Сергій Миколайович

(підпис)

Кваліфікаційну роботу
допущено до захисту
« 12 » червня 2024 р.
Гарант освітньої програми:
к.т.н., доцент
Лавренчук Світлана Василівна

(підпис)

Луцьк – 2024 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерної інженерії та кібербезпеки*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *123 Комп'ютерна інженерія*
Освітня програма: *«Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ проф. Н.Чернящук
« 10 » 01 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Нагурному Владиславу Володимировичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Веб-сервер з функціоналом приватності та системою авторизації інтегрованої з захищеним носієм*

Керівник роботи *к.т.н., доцент Костючко Сергій Миколайович*

затверджені наказом закладу вищої освіти від «30» грудня 2023 року № 459/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи 11.06.2024р.

3. Вихідні дані до роботи *Джерелом розробки є науково-технічна література та публікації в періодичних виданнях з даного питання, опубліковані зарубіжні та вітчизняні роботи в даній області та різні інтернет-ресурси технічного спрямування*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Вступ
Вивчення джерел і методик
Вибір технологій
Розробка системи
Висновки

5. Перелік графічного (ілюстративного) матеріалу:

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналітична частина	*Костючко С.М., доцент*		
Проектна частина	*Костючко С.М., доцент*		
Практична частина	*Костючко С.М., доцент*		
Нормоконтроль	*Багнюк Н.В., доцент*		
Гарант ОП	*Лавренчук С.В., доцент*		
Показник запозичень тексту	***___%***		
Академічна доброчесність	*Міскевич О.І.,асистент*		

7. Дата видачі завдання 10.01.2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	*Розділ 1. Вивчення джерел і методик*	до 15.02.2024 р.	Виконано
2.	*Розділ 2. Вибір технологій*	до 15.03.2024 р.	Виконано
3.	*Розділ 3. Розробка системи*	до 04.05.2024 р.	Виконано
4.	*Висновки*	до 07.05.2025 р.	Виконано
5.	*Формування списку використаних джерел*	до 10.05.2024 р.	Виконано
6.	*Формування додатків*	до 15.05.2024 р.	Виконано
7.	*Оформлення ілюстративного матеріалу*	до 20.05.2024 р.	Виконано
8.	*Нормоконтроль*	до 01.06.2024 р.	Виконано
9.	*Інструментальна перевірка на академічний плагіат*	до 04.06.2024 р	Виконано
10.	*Представлення кваліфікаційної роботи бакалавра до захисту*	до 11.06.2024 р.	Виконано

Здобувач вищої освіти _____ Нагурний В.В.
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ Костючко С.М.
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Нагурний В. В. Веб-сервер з функціоналом приватності та системою авторизації інтегрованої з захищеним носієм. Рукопис.

Кваліфікаційна робота бакалавра ОП «Комп'ютерна інженерія» спеціальності 123 Комп'ютерна інженерія. Луцький національний технічний університет. Луцьк, 2024.

Кваліфікаційна робота складається з вступу, трьох розділів, висновків, списку використаних джерел.

У першому розділі кваліфікаційної роботи розглянуто сучасний стан проблеми, задачі та питання які необхідно розв'язати у роботі.

У другому розділі були показані переваги вибраного напряму роботи у порівнянні з іншими можливими вирішеннями поставлених завдань.

У третьому розділі було розроблено систему, описано розробку функціональних рішень.

Об'єкт дослідження: веб-сервери та їх інфраструктура.

Предмет дослідження: механізми авторизації і забезпечення приватності в контексті веб-серверів, інтеграція цих механізмів з фізичними захищеними носіями.

Мета роботи: розробка та реалізація веб-сервера, що містить вдосконалені функції приватності та систему авторизації, яка інтегрована з захищеним носієм для підвищення безпеки доступу.

Ключові слова: Токен, веб-ресурс, фреймворк, автентифікація, контроль доступу.

ANNOTATION

Nagurnyi V. V. A web server with privacy functionality and an authorization system integrated with a secure medium. Manuscript.

Bachelor's qualification work of the EP «Computer Engineering» specialty 123 Computer Engineering. Lutsk National Technical University. Lutsk, 2024.

The qualification work consists of an introduction, three sections, conclusions, and a list of used sources.

In the first section of the qualification work, the current state of problems, tasks and questions that must be solved in the work is considered.

The second section showed the advantages of the chosen line of work in comparison with other possible solutions to the tasks.

In the third chapter, the system was developed, the development of functional solutions was described.

Research object: web servers and their infrastructure.

The subject of research: mechanisms of authorization and ensuring privacy in the context of web servers, integration of these mechanisms with physical protected media.

The purpose of the work: development and implementation of a web server containing advanced privacy features and an authorization system that is integrated with a secure medium to increase access security.

Keywords: Token, web resource, framework, authentication, access control.

ЗМІСТ

ВСТУП

Актуальність теми веб-сервера з функціоналом приватності та системою авторизації, інтегрованої з захищеним носієм, обумовлена зростаючими загрозами кібербезпеки та потребою в більш високому рівні захисту особистих даних у цифровому світі. У сучасних умовах, коли майже всі аспекти особистого та професійного життя людей переплетені з інтернетом, веб-сервери стають первинними цілями для кібератак. Це стосується не тільки великих корпорацій, але й малих та середніх підприємств, які часто не мають достатніх ресурсів для забезпечення належного рівня кіберзахисту. Тому розробка веб-серверів, які містять вбудовані засоби для підвищення приватності та безпеки, є важливою для забезпечення конфіденційності та інтегритету користувацьких даних.

Інтеграція системи авторизації з захищеними носіями пропонує новий рівень захисту, здатний значно знизити ризики несанкціонованого доступу та витоку інформації. Використання фізичних захищених носіїв, як метод аутентифікації, дозволяє впровадити багаторівневий захист, що є особливо критичним для секторів, де вимоги до безпеки інформації є високими, наприклад, у фінансових установах, урядових організаціях, та медичних закладах. Таким чином, тема розробки захищеного веб-сервера з удосконаленою системою авторизації стає дедалі більш актуальною і потребує додаткових досліджень та розробок в цій області.

Об’єкт дослідження: веб-сервери та їх інфраструктура.

Предмет дослідження: механізми авторизації і забезпечення приватності в контексті веб-серверів, інтеграція цих механізмів з фізичними захищеними носіями.

Мета роботи: розробка та реалізація веб-сервера, що містить вдосконалені функції приватності та систему авторизації, яка інтегрована з захищеним носієм для підвищення безпеки доступу.

В роботі були поставлені такі завдання:

– Аналіз існуючих рішень веб-серверів з функціоналом приватності та системами авторизації. Дослідити наявні методи та технології, виявити їх недоліки та переваги.

– Вивчення принципів роботи захищених носіїв і їх інтеграція з системами авторизації. Оцінити можливості різних типів захищених носіїв та способи їх інтеграції у веб-сервера.

– Розробка архітектури веб-сервера. Проектування структури веб-сервера, яка оптимально використовує функціонал приватності та інтегровану систему авторизації.

– Імплементація системи авторизації на базі захищеного носія. Програмування та налаштування системи авторизації, яка використовує фізичні захищені носії для ідентифікації користувачів.

– Тестування системи з точки зору безпеки та приватності. Проведення різноманітних тестів для оцінки надійності та ефективності імплементованих механізмів.

РОЗДІЛ 1
ВИВЧЕННЯ ДЖЕРЕЛ І МЕТОДИК

1.1 Огляд сучасних клієнтно-серверних архітектур

Архітектура клієнт-сервер описує обчислювальну модель, у якій один комп’ютер, який називається клієнтом, запитує ресурс від іншого комп’ютера, який називається сервером, через мережеве з’єднання. Сервер отримує запит, обробляє його і відповідає клієнту. У цій моделі може існувати один або кілька клієнтських комп’ютерів, які запитують ресурси або служби від одного або кількох серверів, які працюють разом для обслуговування запиту. Сервер зазвичай має базу даних, де він зберігає свої дані та запускає програми, які дозволяють йому отримувати та обробляти запити. Існують стандартизовані протоколи, які дозволяють клієнту і серверу спілкуватися. Вони включають протокол передачі гіпертексту (HTTP), протокол передачі файлів (FTP) і простий протокол передачі пошти (SMTP). Ця архітектура забезпечує міжпроцесний зв’язок між клієнтом і сервером, за допомогою якого вони можуть обмінюватися даними. На цій моделі працює багато програм, включаючи обмін електронною поштою, системи баз даних та Інтернет.

Запити надходять від клієнта, і вони надсилаються на сервер через канал зв'язку, як мережа. Клієнтами можуть бути прості комп’ютери, які запускають клієнтські програми, такі як веб-браузери, мобільні програми або будь-які інші програми, які можуть запитувати послугу або ресурс. Такі комп’ютери не потребують багато конфігурацій або складних програм, оскільки вони не обслуговують жодного запиту. З іншого боку, сервери – це складні комп’ютери з високою обчислювальною потужністю на основі програм, на яких вони працюють. Сервер отримує типові запити, обробляє їх і відповідає клієнту. Для створення сервера потрібні складні привілеї, оскільки вони повинні запускати багато програм, включаючи механізми безпеки, які перевіряють і автентифікують запити користувачів, перш ніж дозволити їх обробку.

Архітектура клієнт-сервер має деякі відмінні архітектурні конструкції, які відрізняють її від інших. У першому випадку і клієнтському, і серверному комп’ютерам потрібен протокол, через який вони можуть обмінюватися інформацією. Ці комп’ютери взаємодіють безпосередньо з протоколом транспортного рівня. У моделі OSI існують рівні, через які проходить інформація, коли вона переміщується від одного комп’ютера до іншого, отже, протокол стає в нагоді, щоб гарантувати, що кожен із цих рівнів може розуміти та передавати дані під час проходження через нього. Транспортний рівень використовує протоколи нижнього рівня для надсилання та отримання повідомлень. Іншою характеристикою є здатність одного сервера обслуговувати кілька запитів одночасно, причому серверу потрібні різні програми, які можуть обробляти ці запити. Іншою важливою характеристикою цієї архітектури є її масштабованість як вертикально, так і горизонтально, отже, до архітектури можна підключати все більше і більше серверів для підтримки робочого навантаження. У той же час, можливості сервера можуть бути збільшені, як оперативна пам'ять і процесор. В архітектурі клієнт-сервер клієнтські та серверні комп’ютери можуть працювати на різнорідних апаратних і програмних ресурсах.

З моменту створення та впровадження клієнт-серверна архітектура внесла кілька втручань та інновацій, деякі з яких дозволили спростити співпрацю та спільне використання ресурсів. Однією з видатних особливостей цієї архітектури є її здатність централізовано зберігати дані з віддаленим доступом з клієнтських комп’ютерів, географічно розподілених по всьому світу. В ідеалі це означає, що доступ до ресурсів сервера можна отримати з кількох місць. Цей дистрибутив забезпечує масштабованість, оскільки більше ресурсів можна легко підключити до архітектури та використовувати з будь-якого місця. Це збільшило обчислювальну потужність архітектури та забезпечило скорочення або повну відсутність часу під час обслуговування запитів. Така модель полегшує та здешевлює підтримку ресурсів у цій архітектурі, оскільки ними можна керувати централізовано. Балансування навантаження можливе, оскільки архітектура дозволяє легко підключати додаткові резервні сервери та сервери-репліки, які

розподіляють робоче навантаження між собою та водночас пропонують простіший варіант відновлення у випадку збою одного сервера. Усе це спрямовано на те, щоб зробити можливим спільне використання ресурсів незалежно від фізичного місцезнаходження користувача.

З моменту створення та використання клієнт-серверна архітектура застосовувалася в кількох конструкціях і програмах із використанням різноманітних переваг. Його перевагою є недоліки, які потенційно можуть перешкодити його реалізації, якщо його не розглянути та не керувати належним чином. Одним із ключових недоліків цієї архітектури є забезпечення безпеки будь-якого спільного ресурсу. Відповідно до його конструкції вся інформація зберігається централізовано, отже, будь-яка проблема з мережевим з'єднанням між клієнтом і сервером може посилити впливають або перешкоджають зручності використання цієї моделі. Будь-яка технічна проблема з сервером унеможливить підключення клієнтських комп'ютерів і доступ до будь-яких ресурсів із сервера. Атаки типу «людина посередині» та атаки «відмова в обслуговуванні» (DoS) є одними з найпоширеніших атак, які потенційно можуть вплинути на впровадження цієї архітектури. DoS робить ресурси недоступними, тоді як «Людина посередині» змінює інформацію під час її пересилання між клієнтом і сервером. Серед інших атак – підробка пакетів. Початкове впровадження цієї архітектури є дуже дорогим, оскільки передбачає придбання та налаштування потужних серверів, налаштування механізмів безпеки, таких як придбання та встановлення брандмауерів, а також налаштування мережі, серед іншого.

Архітектура клієнт-сервер може бути реалізована у двох формах, тобто архітектура тонкого клієнта та архітектура товстого клієнт-сервера. В архітектурі тонкого клієнта проект передбачає використання клієнтських комп'ютерів, які запускаються для доступу до ресурсів із сервера. Ці клієнтські комп'ютери не обробляють інформацію зі свого боку, оскільки вони повністю покладаються на мережі для доступу та запиту ресурсу. Цей дизайн здебільшого використовується там, де задіяна конфіденційна інформація та де клієнтські

комп’ютери не можуть запускати деякі програми або навіть зберігати дані. З іншого боку, товстий клієнт передбачає використання клієнтських комп’ютерів, які можуть обробляти деякі запити локально, оскільки вони запускають програми, які дозволяють їм обробляти ці запити. Наприклад, мобільні програми можуть запускати деякі програми незалежно, не запитуючи жодних ресурсів у сервера. Однак періодично вони можуть підключатися до сервера, якщо потрібні деякі віддалені ресурси.

1.2 Аналіз методик аутентифікації та авторизації

Повна система веб-автентифікації важлива для впорядкованої роботи веб-служб. Як перша лінія захисту, це передумова безпеки служби веб-додатків. По суті, автентифікація – це процес перевірки облікових даних. Метою автентифікації є ідентифікація дозволеного користувача. Він може гарантувати, що користувач, який використовує веб-сервіс із цифровою ідентифікацією, є законним власником цифрової ідентифікації. Ключовими елементами системи веб-автентифікації є облікові дані та стратегія автентифікації. Люди використовують облікові дані, щоб підтвердити свою особу, облікові дані повинні мати певну надійність, щоб протистояти атакам на вгадування, підробку та відтворення. Верхня межа придатності системи автентифікації визначається обліковими даними. Однак при розробці системи автентифікації ми повинні повністю враховувати вимоги конкретних прикладних сценаріїв. Стратегія автентифікації – це технічне рішення, яке запускається після всебічного розгляду питань безпеки, атрибутів облікових даних, вартості впровадження та взаємодії з користувачем. Він реалізує обмеження безпеки щодо облікових даних автентифікації. Нижня межа зручності використання системи аутентифікації визначається стратегією аутентифікації. У безперервній чорно-білій грі побудова системи веб-автентифікації стає досконалішою, різноманітна облікова інформація, повна стратегія автентифікації та надійна мережева передача дають можливість людям прагнути до кращого досвіду веб-автентифікації. Ця стаття

присвячена двом ключовим елементам веб-автентифікації: облікові дані та стратегія автентифікації. У другій частині будуть представлені основні облікові дані, які використовуються в веб-автентифікації: пароль (включаючи статичний пароль і динамічний пароль), біометрія, сеанс, веб-токен, цифровий сертифікат тощо, підсумовані їхні характеристики, переваги та недоліки, а також обговорено захист облікових даних. Третя частина представить стратегії автентифікації, вибере важливі схеми автентифікації, починаючи з Web 1.0, підсумує їх принцип роботи та проаналізує їх логічну безпеку. Четверта частина підсумовує весь документ і висуває деякі погляди на тенденцію розвитку веб-автентифікації.

У веб-системі автентифікації облікові дані є різновидом інформації, яка втілює легітимність особи. Коли користувач, який заявляє про цифрову ідентифікацію, запитує веб-ресурс, веб-автентифікація вимагає від користувача надати певні облікові дані, які можуть підтвердити легітимність його або її особи, і передає зібрані облікові дані на наступні кроки для перевірки. Розробники веб-додатків повинні враховувати вимоги під час розробки процесу сертифікації, вибрати правильні облікові дані та зробити веб-послуги більш безпечними та зручними.

Пароль є найпоширенішим обліковим даним у веб-автентифікації з часів Web 1.0. Хоча безпека пароля нижча, ніж біометрія, цифрові сертифікати та інші облікові дані, він має такі переваги, як простота впровадження, низька вартість і легкість у розгортанні. Пароль залишатиметься основним обліковим даним веб-автентифікації протягом певного періоду часу.

Статичний пароль – це секретна інформація, що складається з символів, яка не має обмежень за часом і може бути змінена та використана повторно. В даний час більшість веб-сайтів, включаючи нещодавно створені сайти, все ще вибирають текстовий пароль як один із своїх основних облікових даних для автентифікації [1]. Текстовий пароль – це рядок, який встановлюється користувачем відповідно до звичок використання, запам'ятовується та зберігається користувачем. Однак людський мозок може запам’ятати лише приблизно від 5 до 7 паролів [2], і користувач часто використовує кілька веб-

служб із паролями як обліковими даними для автентифікації, у поєднанні з недостатньою увагою користувача до інформаційної безпеки в щоденних справах [3], що призводить до активна генерація та використання вразливих команд з низькою інформаційною ентропією [4]. Зважаючи на дедалі серйозніші проблеми безпеки, спричинені дефектами текстових паролів, дослідник запропонував рішення з точки зору заміни, управління та вдосконалення. Графічний пароль [5] є альтернативою текстовому паролю, він більше підходить для керування паролями кількох облікових записів, оскільки людський мозок може розпізнавати та запам’ятовувати зображення краще, ніж текст. Існує три способи встановлення графічного пароля: на основі графічного пароля креслення типовою схемою є DAS [6], запропонований Jermyn та ін.; на основі графічного пароля розташування репрезентативною схемою є метод визначення послідовності позицій зображення, запропонований Блондером [7]; на основі когнітивного графічного пароля типовою схемою є Passfaces [8]. Що стосується керування паролями, широко використовуваний менеджер паролів [9] надає користувачам зручні та безпечні послуги генерації та зберігання паролів, користувачеві потрібно лише запам’ятати основний пароль для входу в менеджер паролів, тоді як паролі інших веб-облікові записи можуть бути встановлені користувачами або автоматично згенеровані менеджером із високою інформаційною ентропією та безпечно зберігати ці паролі в менеджері паролів.

Динамічний пароль, також відомий як одноразовий пароль (OTP) – це секретна інформація, згенерована терміналом користувача або сервером на основі алгоритму синхронізації подій/часу або алгоритму виклик-відповідь. Через невизначені чинники автентифікації паролі, отримані під час кожного процесу автентифікації, відрізняються, тому зловмисник не може отримати доступ до системи за допомогою атаки відтворення. Алгоритм HOTP (одноразовий пароль на основі HMAC) [10] є найбільш класичним OTP-алгоритмом синхронізації подій, псевдокод алгоритму HOTP виглядає так: HOTP(K, C) = Truncate(HMAC-SHA-1(K,C))) У початковому стані термінал користувача уніфікує значення лічильника та розмір кроку з сервером, а потім

кожного разу, коли обчислюється HOTP, значення лічильника з обох сторін додає один крок. K і C використовуються разом як початкові під час генерації HOTP. Термінал користувача спочатку отримує 20-байтовий рядок за допомогою алгоритму HMAC-SHA-1, а потім перетворює рядок, згенерований алгоритмом HMAC-SHA-1, у ряд десяткових цифр: HOTP і надсилає його на сервер, лічильник подій, а потім збільшується на один крок. Після отримання HOTP сервер генерує HOTP у тому самому процесі та виконує порівняльну перевірку

Паролі та біометричні облікові дані зазвичай використовуються для операцій входу. Коли користувачі успішно входять у систему та відвідують інші сторінки веб-сайту, якщо вони щоразу знову надсилають облікові дані вручну, це означає, що користувачі не зможуть працювати, а повторне надсилання збільшить ризик викрадення облікових даних користувача. Однак, з точки зору безпеки, під час відвідування інших сторінок веб-сайту необхідні облікові дані для автентифікації користувача. Щоб вирішити цю проблему, коли користувач успішно входить в систему, «ім’я користувача/пароль» буде замінено прозорими обліковими даними для користувача: ідентифікатор сеансу. Після успішної реєстрації користувача веб-сервер генерує новий сеанс. Сеанс можна розуміти як список, який записує статус користувача, ідентичність та іншу інформацію у формі змінних сеансу. Кожен сеанс має свій унікальний ідентифікатор сеансу, який є єдиною інформацією про сеанс, відомою користувачеві. Браузер автоматично надсилатиме ідентифікатор сеансу на сервер щоразу, коли користувач натискатиме нову сторінку. Сервер запитує сеанс відповідно до ідентифікатора сеансу, якщо сеанс не закінчився, він пройде автентифікацію.

Через велику кількість додатків розподілених серверів через обмеження доменів файлів cookie сервер не може поділитися сеансом з іншими серверами, що призводить до того, що стан сеансу користувача не підтримується належним чином. Веб-токен може вирішити ці проблеми. Коли користувач реєструється, сервер веб-автентифікації кодує ідентифікаційну інформацію користувача для створення рядка та використовує її як маркер, коли клієнт запитує дані. Після цього користувач може пройти автентифікацію, лише представивши маркер під

час входу в систему. Інформація, що зберігається в маркері, включає ідентифікаційні дані користувача, таблицю дозволів тощо, щоб гарантувати, що інформація не піддається зловмисному втручанню хакерам, серверу зашифрує їх, згенерує підпис, додасть підпис до маркера та поверне його клієнту. Коли клієнт отримує маркер, він збереже маркер у локальному файлі. Коли користувач надсилає запит на веб-ресурс, за умови, що юридичний маркер надсилається на сервер, автентифікація може бути завершена, тому йому не потрібно вводити пароль та інші облікові дані. Токен видимий для користувачів і осіб без стану, сервер не зберігає та не записує токен, під час автентифікації потрібно перевірити лише інформацію та підпис у токені. Веб-маркер JSON (JWT) – це основний обліковий запис автентифікації на основі маркера в Інтернеті. У 2015 році IETF створив стандарт веб-токенів JSON (JWT). Веб-токен JSON – це компактний і самодостатній формат представлення декларації. Оголошення закодовано як об’єкт JSON – корисне навантаження веб-підпису JSON (JWS) або відкритого тексту веб-шифрування JSON (JWE). JWT складається з трьох частин: заголовка, корисного навантаження та підпису, кожна частина розділена крапкою «.» і header.payload.signature. JWT має багато переваг, це може бути підтримка між мовами; він має просту структуру та меншу зайнятість байтів, тому дуже підходить для передачі по мережі; йому не потрібно зберігати сеанс на сервері, тому програму легко розширити та підходить для розподіленого середовища.

Стратегія автентифікації становить основну основу системи веб-автентифікації, і ідеальна стратегія автентифікації повинна точно ідентифікувати особу користувача, допускати легальних користувачів і виключати нелегальних користувачів. Розробка політики автентифікації повинна брати безпеку як основну відправну точку та враховувати сценарії застосування веб-сервісів, мережі середовище, в якому вони розташовані, вартість впровадження та всебічний досвід користувача.

Автентифікація на основі сеансу та файлів cookie HTTP – це протокол без стану, статус успішної автентифікації користувача не може бути збережений на

рівні протоколу. Тому для додавання невизначених функцій керування станом у протокол HTTP потрібен сеанс керування файлами cookie. Наразі більшість веб-сайтів використовують автентифікацію на основі сеансових файлів cookie. Процес автентифікації на основі сеансових файлів cookie виглядає наступним чином:

Крок 1: зовнішня форма отримує облікові дані імені користувача та пароля, а потім передає облікові дані на сервер через повідомлення запиту в методі GET або POST через протокол HTTP;

Крок 2: Сервер перевіряє ім'я користувача / пароль. Якщо він пройдений, він генерує сеанс для запису статусу автентифікації, а потім записує ідентифікатор сеансу, що відповідає сеансу, у поле Set-Cookie у заголовку пакета відповіді та повертає його клієнту;

Крок 3: після отримання ідентифікатора сеансу, надісланого сервером, клієнт зберігає його як файл cookie локально. Коли наступний запит надсилається на сервер, браузер автоматично надсилає файл cookie з ідентифікатором сеансу, а сервер розпізнає ідентифікатор користувача шляхом перевірки ідентифікатора сеансу.

У зовнішній формі користувач вводить ім’я користувача та пароль, а сервер об’єднує SQL на основі введеного імені користувача та пароля та підключається до бази даних для запиту на основі SQL. Однак деякі зловмисники введуть у форму спеціальний SQL, щоб дозволити серверу виконати автентифікацію. У цей час, якщо сервер не реалізує принцип «розділення даних і коду», зловмисники успішно виконають атаки SQL-ін’єкції. Існує кілька способів належного захисту від атак SQL-ін’єкцій. Найбільш найкращим способом є використання попередньо скомпільованих операторів, зв’язування змінних безпосередньо, семантика SQL не зміниться, а оператори, вставлені зловмисниками, запитуватимуться лише як імена користувачів або паролі; використовуючи захищені збережені процедури, ми можемо заздалегідь визначити набір збережених процедур у базі даних, які уникають використання динамічних операторів SQL або використання суворої фільтрації вхідних даних

і функцій кодування для максимальної обробки введених користувачем даних; перевірка та обмеження типу вхідних даних може ефективно боротися з впровадженням SQL; за допомогою функцій кодування та фільтрації деякі спеціальні символи будуть екрановані або видалені; нарешті, дозволи, надані веб-програмі, повинні бути мінімізовані за звичайних випадків використання. Життєвий цикл сесії встановлюється сервером. Він починається з серверної програми, яка викликає оператор створення сеансу. Сесія дійсна протягом життєвого циклу. Після закінчення життєвого циклу сервер знищує сеанс. Тепер деякі веб-сервіси призначені для кращих користувачів. Досвід: якщо користувач все ще активний наприкінці життєвого циклу сеансу (виконує такі операції, як запит або надсилання), життєвий цикл сеансу автоматично подовжиться. На додаток до закінчення терміну сеансу, його також можна штучно припинити, наприклад користувач натискає, щоб вийти. Потім серверна програма викличе оператор знищення, щоб завершити сеанс.

1.3 SPA (Single Page Application)

SPA – це веб-програма, яка повністю завантажує всі ресурси в початковому запиті, а потім компоненти сторінки замінюються іншими компонентами залежно від взаємодії користувача. Коли ми порівнюємо SPA з традиційним веб-додатком, ми можемо побачити, що існує аналогія між «станами» SPA та «веб-сторінками», навігація «веб-сторінок» у традиційному веб-додатку аналогічна навігації «станом» у SPA.

«Програма Single Page (SPA) складається з окремих компонентів, які можна замінювати/оновлювати незалежно, без оновлення/перезавантаження всієї сторінки, тому всю сторінку не потрібно перезавантажувати під час кожної дії користувача».

1) Окремі компоненти: уся сторінка розділена на менші компоненти, які взаємодіють один з одним.

2) Замінено/оновлено: компонент/область/частина сторінки замінено/ оновлено з будь-якими змінами за запитами користувачів.

3) Оновлення/перезавантаження: уся сторінка ніколи не перезавантажується/оновлюється, скоріше новий вміст завантажується в деякій частині чи розділі відповідно до нових даних.

4) Дії користувача: односторінкова програма відповідає за дуже швидку обробку всіх дій користувача, таких як натискання кнопок, введення з клавіатури тощо, і, отже, забезпечує дуже гнучкий інтерфейс користувача.

SPA дозволяє більш гнучкий і елегантний спосіб роботи з даними. Оновлення певної частини чи розділу сторінки без оновлення всієї сторінки є головною метою SPA, але вся ця гнучкість вимагає більшого інтерактивного інтерфейсу, а це сприяє покращенню взаємодії з користувачем.

Коли ми створюємо SPA за допомогою JavaScript, інтерфейс стає дуже складним. У великих проектах багато розробників працюють разом над створенням інтерфейсу. Якщо код дуже важко зрозуміти через відсутність розмежування макетів і бізнес-логіки, то обслуговування цього коду стає дуже складним. Це питання підтримки коду більших проектів вирішується, якщо односторінкова програма створена за допомогою AngularJS. Оскільки AngularJS ділить передню частину на 3 частини:

– Модель.

– Перегляд.

– Контролер.

Код, який легко читати та розуміти, стає ключем до досягнення високої ефективності та гарної якості, а також забезпечує плавний інтерфейс користувача.

Як вже було сказано, створення SPA-додатків дозволяє надати легким і швидким веб-сайтам функціональність важких програмних систем з інтерфейсом, що нагадує не веб-ресурси, а прикладні програми. Асинхронний підхід із покроковим виконанням скриптів дозволяє лише частково оновлювати вікно програми SPA при виклику певної функції веб-програми. При цьому

історія навігації і логіка роботи користувача зберігається в адресному рядку і кеші браузера.

Навігація по розділах програми SPA або односторінкового порталу здійснюється без регулярного перезавантаження документів, а клієнтська веб-програма не вимагає періодичного перезапуску при повторенні різних дій, як це відбувається зі звичайними сайтами. Природно, це економить інтернет-трафік і апаратні ресурси клієнтського пристрою.

Загалом варто виділити такі аргументи на користь створення SPA додатків.

Доступність і кросплатформенність. Додатки SPA будуть запущені на всіх пристроях, які мають веб-браузер. Це означає, що вам не потрібно виділяти додаткові кошти на написання програмного забезпечення для різних платформ і операційних систем.

Універсальність і масштабованість. Односторінкові додатки SPA не потребують встановлення та оновлення. Для запуску веб-програми користувачеві достатньо ввести необхідну адресу в рядок веб-браузера.

Швидкість і легкість. Складні апаратні розрахунки виконуються на стороні сервера, а функції веб-додатків завантажуються без повного оновлення веб-сторінок. Це прискорює роботу навіть найскладніших програмних систем.

Надійність і безпека. Дані користувача та інформаційна база веб-додатку зберігаються в хмарі. Таким чином, користувач може повернутися до відкладеного сеансу, навіть якщо він не вдається, і підключається до програми SPA з іншого пристрою.

1.4 Дослідження ринку інструментів для створення SPA

Односторінкова програма SPA є певною мірою оптимальною для побудови програмної логіки та функціональності для веб-проектів. У цьому випадку веб-додаток завантажується на пристрій користувача і працює в рамках лише однієї

веб-сторінки. Функціональність односторінкового додатку SPA можна реалізувати за допомогою не тільки класичної верстки HTML/CSS і мови розмітки веб-документів, але й сучасних програмних технологій, таких як:

JavaScript – мова виконуваних скриптів для веб-сайтів.

React.js, Vue.js і Angular.js – це популярні бібліотеки та фреймворки JavaScript для створення інтерфейсів користувача та розробки односторінкових або мобільних додатків.

Насправді односторінкова програма SPA – це веб-сторінка, до якої підключено багато модулів, сценаріїв та інших файлів, які виконуються не відразу, а під час виклику клієнтського пристрою.

Це лише оболонка для реалізації виконуваних модулів, які можуть бути запущені так званим асинхронним методом, тобто за запитом. Асинхронний принцип обміну даними між пристроями і сервером, або технологія Ajax значно прискорює роботу SPA-додатків і економить інтернет-трафік.

Раніше на пристрій завантажувалися тільки ті модулі, які відповідають за початковий запуск програми SPA. Крім того, залежно від апаратної моделі клієнтського гаджета, набір активних скриптів і бібліотек, необхідних для запуску програми SPA, буде різним.

Така своєрідна логіка SPA-додатків дозволяє запускати на будь-якому пристрої, що має веб-браузер і доступ до Інтернету. Наприклад, найпопулярніші сервіси Google – поштовий клієнт Gmail і перекладач Google Translate – не що інше, як виконувані програми SPA.

РОЗДІЛ 2
ВИБІР ТЕХНОЛОГІЙ

2.1 Вибір фреймворку React для клієнтської частини

ReactJS – це JavaScript бібліотека, яка використовує швидкість JavaScript та впроваджує інноваційний метод рендерингу веб-сторінок, надаючи їм динаміку і реактивність до дій користувача. Ця бібліотека кардинально змінила стратегію розробки у Facebook. З часу її випуску як інструменту з відкритим кодом у 2013 році, React став надзвичайно популярним завдяки своєму прогресивному підходу до створення користувацьких інтерфейсів.

Команда ReactJS покращила швидкість оновлень веб-сторінок за допомогою віртуального DOM. На відміну від інших фреймворків, які використовують реальне DOM-дерево, ReactJS застосовує абстрактну версію. Він може оновлювати дрібні зміни, зроблені користувачем, не втручаючись у інші частини інтерфейсу, що стало можливим завдяки ізольованості компонентів ReactJS.

Переваги ReactJS порівняно з іншими JavaScript фронтенд бібліотеками включають:

– Швидкість завдяки використанню віртуального DOM та різноманітним оптимізаціям рендерингу та оновлення.

– Можливість серверного рендерингу.

– Застосування принципів функціонального програмування, що сприяє легкості тестування та повторного використання компонентів.

– Зручна міграція між версіями завдяки розробленим інструментам для легкого оновлення.

2.2 Використання утилітарних класів Tailwind CSS та підвищення швидкості розробки

Tailwind – це найпопулярніша утиліта, платформа CSS для створення користувацького інтерфейсу найшвидшим і найпростішим способом. Це означає, що на відміну від комплектів інтерфейсу користувача, таких як Bootstrap, tailwind не надає компонентів із попереднім стилем. Він не має теми за замовчуванням. За допомогою tailwind ми стилізуємо елементи, застосовуючи попередньо визначені класи безпосередньо в Html. Ми визначили його для написання вбудованого стилю для досягнення чудового інтерфейсу без запису CSS у різних файлах. Tailwind дозволяє швидко розвивати сучасні веб-сайти, дозволяючи нам писати CSS безпосередньо в нашій розмітці. Він пропонує широкі можливості налаштування, щоб адаптувати стилі відповідно до наших конкретних потреб. Він сканує всі файли Html, компоненти JS та будь-які інші шаблони для імен класів, генерує відповідні стилі, а потім записує їх у статичний файл CSS [23]. Єдина проблема в тому, що наша розмітка може виглядати набагато довшою. Для використання та запуску tailwind ми дотримуємося інструкцій документації tailwind щодо встановлення інструменту CLI, який є найшвидшим способом і рекомендований tailwind [24]. Кроки такі:

– Встановили tailwind через npm і запустили команду init для створення файлів: ‘tailwind.config.js’ і ‘postcss.config.js’.

– Postcss.config.js – єдиний експортований об’єкт JavaScript, який спочатку вкаже запустити tailwind.

Встановивши PostCSS і пов’язані з ним інструменти, такі як «Autoprefixer», ми можемо використовувати можливості фреймворку для оптимізації процесу розробки (рис. 2.1).

Цей файл конфігурації використовується для налаштування теми та властивостей Tailwind за замовчуванням. наприклад: користувальницькі шрифти, колір, міжрядковий інтервал тощо. Використання «PostCSS» значно

підвищує швидкість створення наших проектів, що призводить до підвищення продуктивності.

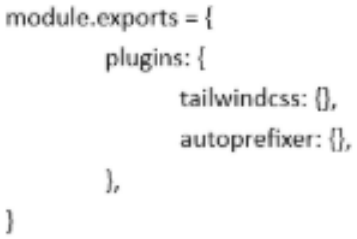

Рисунок 2.1 – Встановлення плагінів Tailwind та Autoprefixer

– Tailwind.configure.js, у цьому файлі ми додали шлях до всіх файлів шаблону.

– У головному файлі css index.css додайте директиви «@tailwind» для кожного з шарів Tailwind: @tailwind base; компоненти @tailwind; та утиліти @tailwind; Це схоже на імпорт стилів попутного вітру, які розпізнаються як справжній синтаксис CSS. Нарешті імпортуйте цей файл css у файл входу react, який є «index.js».

– Запускаємо інструмент CLI, щоб сканувати файли шаблонів для класів і створити CSS: -npm run start.

Tailwind розроблено з можливістю налаштування, що є однією з його ключових переваг. Це дозволяє подолати обмеження, коли це необхідно, дозволяючи нам додавати власний CSS і розширювати різні плагіни. Переваги налаштування Tailwind дають змогу розробникам створювати унікальні, ефективні та узгоджені інтерфейси користувача, пристосовуючись до конкретних потреб проекту та підтримуючи масштабований робочий процес розробки. На рисунку 2.2 показано файл конфігурації tailwind.config.js, який експортує об’єкт module. Цей об’єкт конфігурації містить різні властивості, зокрема вміст, тему, плагіни. У «Вміст» додано шлях до всіх файлів шаблону. Властивість «plugins» визначає додаткові плагіни CSS Tailwind, які потрібно ввімкнути. Перевагою tailwind є його здатність інтегрувати багаторазові плагіни

сторонніх розробників, розширюючи спектр доступних стилів і функцій. Просто встановивши потрібні плагіни через npm та імпортувавши їх, ми можемо легко включити їх у наш проект. Потім ми можемо викликати конкретні плагіни, які хочемо використовувати в масиві «плагіни».

```
module.exports = {
        content: ['./src/**/*.{js,jsx,ts,tsx}', './public/index.html'],
        theme: {
                screens: { ...
                },
                 Extend: { ...
                },
        },
        plugins: [
                require('tailwind-scrollbar'),
                require('tailwindcss-textshadow')
        ],
}
```

Рисунок 2.2 – Файл конфігурації tailwind.config.js

«Тема» містить різні параметри конфігурації. Він включає такі налаштування, як визначення додаткових «ключових кадрів» для анімації, додавання нових кольорів, настроюваних сімейств шрифтів. Всередині екранів ми визначаємо контрольні точки для адаптивного дизайну. На рисунку 2.3 деталі розробки цієї частини.

```
tailwind.config.js
1   // tailwind.config.js
2   const plugin = require('tailwindcss/plugin')
3
4   module.exports = {
5     content: ['./src/**/*.{js,jsx,ts,tsx}', './public/index.html'],
6     // darkMode: false, // or 'media' or 'class'
7     theme: {
8       screens: {…
35      },
36      extend: {
37        keyframes: {…
108       },
109       animation: {…
120       },
121       colors: {
122         transparent: 'transparent',
123         current: 'currentColor',
124         primary: { //lime
125           DEFAULT: 'var(--color-primary-default)',
126           light: 'var(--color-primary-light)'
127         },
128         secondary: { //ocean
129           DEFAULT: 'var(--color-secondary-default)',
130           light: 'var(--color-secondary-light)'
131         },
132         accent: { //fire
133           DEFAULT: 'var(--color-accent-default)',
134           light: 'var(--color-accent-light)'
135         },
```

Рисунок 2.3 – Деталі файлу конфігурації tailwind.config.js

2.3 Застосування Axios замість Fetch API

У сфері розробки програмного забезпечення дуже важливою є можливість безперебійної взаємодії з віддаленими серверами та обміну даними через Інтернет. Незалежно від того, чи йдеться про отримання даних з API, виконання операцій CRUD або виконання будь-яких інших завдань, пов’язаних із мережею, важливість виконання запитів HTTP неможливо переоцінити. Дві широко використовувані бібліотеки JavaScript, Fetch і Axios, стали найкращим вибором для розробників, які прагнуть обробляти HTTP-запити.

У цій статті ми порівняємо Axios і Fetch, щоб побачити, як їх можна використовувати для виконання різних завдань. Сподіваюся, до кінця статті ви краще зрозумієте обидва API.

Axios – це стороння HTTP-клієнтська бібліотека для виконання мережевих запитів. Він заснований на обіцянках і забезпечує чистий і послідовний API для обробки запитів і відповідей.

Ви можете додати його до свого проекту через мережу розповсюдження вмісту (CDN) або за допомогою менеджера пакетів (наприклад, npm). Деякі з його основних функцій включають: створення XMLHttpRequests у браузері, виконання http-запитів у середовищі node.js, скасування запитів і перехоплення запитів і відповідей.

Fetch, як і Axios, є HTTP-клієнтом на основі обіцянок. Це вбудований API; тому нам не потрібно нічого встановлювати чи імпортувати. Він доступний у всіх сучасних браузерах; ви можете перевірити це на caniuse. Fetch також доступний у node.js – докладніше про це можна прочитати тут .

Давайте тепер порівняємо їх базовий синтаксис, функції та випадки використання, намагаючись зрозуміти дві бібліотеки, які широко використовуються.

Axios спрощує HTTP-запити за допомогою ланцюжкового API, який дозволяє легко налаштовувати параметри запиту, такі як заголовки, дані та метод запиту.

Ось як використовувати Axios для надсилання запиту [POST] із спеціальними заголовками на URL-адресу. Axios автоматично перетворює дані в JSON, тож вам не потрібно (лістинг 2.1).

Лістинг 2.1 – Перетворення даних в JSON

```
const axios = require('axios');

const url = 'https://jsonplaceholder.typicode.com/posts';
const data = {
  title: 'Hello World',
  body: 'This is a test post.',
  userId: 1,
};

axios
  .post(url, data, {
    headers: {
      Accept: 'application/json',
      'Content-Type': 'application/json;charset=UTF-8',
    },
  })
  .then(({ data }) => {
    console.log(«POST request successful. Response:», data);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Кінець лістингу 2.1

Порівняйте цей код із кодом API отримання, який дає точно такий же результат (лістинг 2.2):

Лістинг 2.2 – Код API перетворення даних в JSON

```
const url = «https://jsonplaceholder.typicode.com/todos»;
const options = {
  method: «POST»,
  headers: {
    Accept: «application/json»,
    «Content-Type»: «application/json;charset=UTF-8»,
```

```
  },
  body: JSON.stringify({
    title: «Hello World»,
    body: «This is a test post.»,
    userId: 1,
  }),
};

fetch(url, options)
  .then((response) => response.json())
  .then((data) => {
    console.log(«POST request successful. Response:», data);
  });
```

Кінець лістингу 2.2

Fetch використовує властивість body для надсилання даних у запиті POST, тоді як Axios використовує властивість data. Axios автоматично перетворює дані відповіді сервера, тоді як у Fetch вам потрібно викликати метод response.json, щоб розібрати дані в об’єкт JavaScript. Крім того, Axios надає відповідь на дані в межах об’єкта даних, тоді як Fetch дозволяє зберігати кінцеві дані в будь-якій змінній.

Fetch пропонує точний контроль над процесом завантаження, але створює складність, вимагаючи обробки двох промісів. Крім того, під час роботи з відповідями Fetch вимагає розбору даних JSON за допомогою методу .json(). Остаточні отримані дані можуть бути збережені в будь-якій змінній.

Для помилок відповіді HTTP Fetch не видає автоматично помилку. Замість цього він вважає відповідь успішною, навіть якщо сервер повертає код статусу помилки (наприклад, 404 Не знайдено).

Щоб явно обробляти помилки HTTP у Fetch, розробники мають використовувати умовні оператори в .then() блоці для перевірки response.ok властивості. Якщо response.ok значення false, це означає, що сервер відповів кодом статусу помилки, і розробники можуть усунути помилку відповідно (лістинг 2.3).

Ось запит [GET] за допомогою fetch:

Лістинг 2.3 – Запит GET

```
fetch('https://jsonplaceholder.typicode.com/todos')
  .then(response => {
    if (!response.ok) {
      throw Error(`HTTP error: ${response.status}`);
    }
    return response.json();
  })
  .then(data => {
    console.log('Data received:', data);
  })
  .catch(error => {
    console.error('Error message:', error.message);
  });
```

Кінець лістингу 2.3

Axios, з іншого боку, спрощує обробку відповіді, надаючи прямий доступ до властивості даних. Він автоматично відхиляє відповіді за межами діапазону 200-299 (успішні відповіді). Використовуючи .catch() блок, можна отримати інформацію про помилку, включно з тим, чи була отримана відповідь і, якщо так, її код статусу (лістинг 2.4). Це відрізняється від вибірки, де невдалі відповіді все одно вирішуються.

Ось запит [GET] за допомогою Axios:

Лістинг 2.4 – Запит GET за допомогою Axios

```
const axios = require(«axios»);
axios
  .get(«https://jsonplaceholder.typicode.com/todos»)
  .then((response) => {
    console.log(«Data received:», response.data);
  })
  .catch((error) => {
    if (error.response) {
      console.error(`HTTP error: ${error.response.status}`);
    } else if (error.request) {
      console.error(«Request error: No response received»);
    } else {
      console.error(«Error:», error.message);
    }
```

```
});
```

Кінець лістингу 2.4

Автоматично викидаючи помилки для невдалих відповідей, Axios спрощує обробку помилок і дозволяє розробникам зосередитися на реалізації належної логіки обробки помилок без необхідності вручну перевіряти кожну відповідь на успішність чи невдачу.

Однією з ключових особливостей Axios є його здатність перехоплювати HTTP-запити. Перехоплювачі HTTP стають у пригоді, коли вам потрібно перевірити або змінити HTTP-запити від вашої програми до сервера або навпаки. Ця функція необхідна для виконання різноманітних завдань, таких як журналювання, автентифікація або повторна спроба невдалих запитів HTTP.

З перехоплювачами вам не доведеться писати окремий код для кожного HTTP-запиту. Перехоплювачі HTTP корисні, коли ви хочете встановити глобальну стратегію обробки запитів і відповідей (лістинг 2.5).

Ось як перехопити HTTP-запит за допомогою Axios:

Лістинг 2.5 – Перехоплення HTTP-запиту за допомогою Axios

```
const axios = require(«axios»);
// Register a request interceptor
axios.interceptors.request.use((config) => {
  // Log a message before any HTTP request is sent
  console.log(«Request was sent»);
  return config;
});

// Send a GET request
axios
  .get(«https://jsonplaceholder.typicode.com/todos»)
  .then(({data}) => {
    console.log(«Data received:», data);
  })
  .catch((error) => {
    console.error(«Error:», error.message);
  });
```

Кінець лістингу 2.5

У цьому коді axios.interceptors.request.use() метод використовується для визначення коду, який запускається перед надсиланням HTTP-запиту.

Крім того, axios.interceptors.request.use() може бути використаний для перехоплення відповіді від сервера. Наприклад, у разі помилки мережі можна використовувати перехоплювачі відповіді, щоб повторити той самий запит за допомогою перехоплювачів.

За замовчуванням fetch() не надає способу перехоплення запитів, але неважко придумати обхідний шлях. Ви можете перезаписати глобальний метод fetch() (лістинг 2.6).

Ось як перехопити HTTP-запит за допомогою Fetch:

Лістинг 2.6 – Перехоплення HTTP-запиту за допомогою Fetch

```
fetch = ((originalFetch) => {
  return (...arguments) => {
    return originalFetch.apply(this, arguments).then((response) => {
      if (!response.ok) {
        throw new Error(`HTTP error: ${response.status}`);
      }
      console.log(«Request was sent»);
      return response;
    });
  };
})(fetch);

fetch(«https://jsonplaceholder.typicode.com/todos»)
  .then((response) => response.json())
  .then((data) => {
    console.log(«Data received:», data);
  })
  .catch((error) => {
    console.error(«Error:», error.message);
  });
```

Кінець лістингу 2.6

Fetch, як правило, має більше шаблонного коду порівняно з Axios. Однак вибір повинен ґрунтуватися на випадку використання та індивідуальних потребах.

Тайм-аут відповіді – справді ще одна важлива область для порівняння між Fetch і Axios. Час очікування відповіді означає тривалість, протягом якої клієнт чекатиме відповіді від сервера, перш ніж розглядати запит як невдалий.

Простота встановлення тайм-ауту в Axios є однією з причин, чому деякі розробники віддають перевагу цьому Fetch. В Axios ви можете використовувати необов’язкову властивість timeout в конфігураційному об’єкті, щоб установити кількість мілісекунд перед тим, як запит буде скасовано. Цей простий підхід дозволяє розробникам визначати тривалість тайм-ауту безпосередньо в конфігурації запиту, забезпечуючи більший контроль і легкість впровадження (лістинг 2.7).

Ось запит [GET] із вказаним тайм-аутом за допомогою Axios:

Лістинг 2.7 – Запит GET із вказаним тайм-аутом за допомогою Axios

```
const axios = require('axios');

// Define the timeout duration in milliseconds
const timeout = 5000; // 5 seconds

// Create a config object with the timeout property
const config = {
  timeout: timeout
};

// Send a GET request with the specified timeout
axios.get('https://jsonplaceholder.typicode.com/todos', config)
  .then(response => {
    console.log('Data received:', response.data);
  })
  .catch(error => {
    console.error('Error fetching data:', error.message);
  });
```

Кінець лістингу 2.7

Запит GET надсилається за допомогою Axios, передаючи об’єкт конфігурації як другий аргумент для визначення часу очікування. Якщо запит виконано успішно, отримані дані реєструються на консолі. Якщо під час запиту

виникає помилка або якщо досягнуто часу очікування, повідомлення про помилку реєструється на консолі.

Fetch пропонує аналогічну функціональність через інтерфейс AbortController, хоча він не такий простий, як версія Axios (лістинг 2.8).

Ось як перехопити HTTP-запит за допомогою Fetch:

Лістинг 2.8 – Перехоплення HTTP-запиту за допомогою Fetch

```
const timeout = 5000; // 5 seconds
// Create an AbortController instance
const controller = new AbortController();
const signal = controller.signal;

// Set up a setTimeout function to abort the fetch request after the specified timeout
const timeoutId = setTimeout(() => {
  controller.abort();
}, timeout);

// Make the fetch request with the specified URL and signal
fetch('https://jsonplaceholder.typicode.com/todos', { signal })
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => {
    console.log('Data received:', data);
  })
  .catch(error => {
    // Check if the error is due to the request being aborted
    if (error.name === 'AbortError') {
      console.error('Request timed out');
    } else {
      console.error('Error fetching data:', error.message);
    }
  })
  .finally(() => {
    // Clear the timeout to prevent it from firing after the request has completed
    clearTimeout(timeoutId);
  });
```

Кінець лістингу 2.8

Для запиту на вибірку встановлюється 5-секундний тайм-аут, і AbortController створюється повідомлення для його скасування, якщо він перевищує це обмеження. Використовуючи setTimeout, запускається таймер, щоб скасувати запит після закінчення цього часу. Потім надсилається запит на вибірку, включаючи сигнал скасування. Після відповіді він перевіряє успішність і аналізує дані. Будь-які помилки, такі як timeouts, керуються відповідним чином. Нарешті, час очікування очищається, щоб уникнути непотрібних затримок після завершення запиту.

Одночасні запити або можливість робити кілька запитів одночасно є ще одним важливим аспектом для порівняння між Fetch і Axios. Ця функція особливо важлива в програмах, де продуктивність і швидкість реагування є найважливішими.

Щоб зробити кілька одночасних запитів, Axios надає axios.all() метод. Просто передайте цьому методу масив запитів, а потім використовуйте axios.spread() для розподілу відповідей на окремі аргументи. Це дозволяє обробляти кожну відповідь окремо (лістинг 2.9).

Ось як надсилати одночасні HTTP-запити за допомогою Axios:

Лістинг 2.9 – Надсилання одночасних HTTP-запитів за допомогою Axios

```
const axios = require('axios');

// Define the base URL for the request
const baseURL = 'https://jsonplaceholder.typicode.com/todos';

// Define the URLs for the requests
const urls = [
  `${baseURL}/1`,
  `${baseURL}/2`,
  `${baseURL}/3`
];

// Create an array of Axios request promises
const axiosRequests = urls.map(url => axios.get(url));

// Send multiple requests simultaneously using `axios.all()`
```

```
axios.all(axiosRequests)
  .then(axios.spread((...responses) => {
    // Handle responses from all requests
    responses.forEach((response, index) => {
      console.log(`Response from ${urls[index]}:`, response.data);
    });
  }))
  .catch(error => {
    console.error('Error fetching data:', error.message);
  });
```

Кінець лістингу 2.9

Щоб досягти такого ж результату за допомогою Fetch, ми повинні покладатися на вбудований Promise.all() метод. Передайте всі запити на вибірку як масив до Promise.all(). Далі обробіть відповідь за допомогою асинхронної функції (лістинг 2.10).

Ось як надсилати одночасні HTTP-запити за допомогою Fetch:

Лістинг 2.10 – Надсилання одночасних HTTP-запитів за допомогою Fetch

```
// Define the base URL for the requests
const baseURL = «https://jsonplaceholder.typicode.com/todos»;

// Define the URLs for the requests
const urls = [`${baseURL}/1`, `${baseURL}/2`, `${baseURL}/3`];

// Create an array to store the fetch request promises
const fetchRequests = urls.map((url) => fetch(url));

// Send multiple requests simultaneously using `Promise.all()`
Promise.all(fetchRequests)
  .then((responses) => {
    // Handle responses from all requests
    responses.forEach((response, index) => {
      if (!response.ok) {
        throw new Error(
          `Request ${urls[index]} failed with status ${response.status}`
        );
      }
      response.json().then((data) => {
        console.log(`Response from ${urls[index]}:`, data);
```

```
    });
  });
})
.catch((error) => {
  console.error(«Error fetching data:», error.message);
});
```

Кінець лістингу 2.10

Хоча цей підхід можливий, він може створити додаткову складність і витрати, особливо для розробників, які не знайомі з концепціями асинхронного програмування.

Зворотна сумісність – це здатність програмної системи або продукту функціонувати належним чином або ефективно навіть при використанні зі старішими версіями залежностей або в старіших середовищах.

Axios забезпечує кращу зворотну сумісність із коробки та пропонує додаткові функції, які можуть полегшити сумісність зі старими системами чи кодовими базами. Однією з головних переваг Axios є широка підтримка браузера, що робить його сумісним навіть зі старішими браузерами, такими як IE11, завдяки використанню в основі XMLHttpRequest.

Для порівняння, Fetch має більш обмежену підтримку веб-переглядачів, яка обслуговує переважно сучасні браузери, такі як Chrome, Firefox, Edge і Safari. Для проектів, які вимагають функції Fetch у непідтримуваних браузерах, інтеграція polyfill, як-от «whatwg-fetch», може подолати прогалину. Цей polyfill розширює підтримку Fetch до старіших браузерів, забезпечуючи сумісність.

Щоб використовувати його, встановіть його за допомогою команди npm так:

```
npm install whatwg-fetch – save
```

Тоді ви можете робити такі запити:

```
import 'whatwg-fetch'
window.fetch(…)
```

Майте на увазі, що вам також може знадобитися полізаповнення обіцянок у деяких старих браузерах.

2.4 Використання Access та Refresh Tokens для контролю доступу та сесій користувачів

Токени – це дані, що підтверджують особу користувача, аналогічні цифровим підписам.

Ретельний баланс між безпекою та досвідом користувача є важливим для автентифікації та авторизації . Користувач може роздратуватися, якщо протоколи надто суворі. З іншого боку, порушення безпеки є неминучим, якщо системи дозволів занадто слабкі.

Маркери доступу та оновлення забезпечують рішення, яке відповідає обом вимогам.

Access Tokens (від сервера авторизації) дозволяє отримати тимчасовий доступ до обмежених ресурсів, таких як API або веб-сайти.

Як правило, Access Tokens дійсні лише кілька хвилин або годин, залежно від налаштувань для захисту сервера ресурсів. Вони також містять такі функції безпеки, як підписи.

Будь-який користувач із Access Tokens проходить автоматичну автентифікацію, незалежно від того, справжній він чи зловмисний . Давайте розглянемо приклад (рис. 2.4):

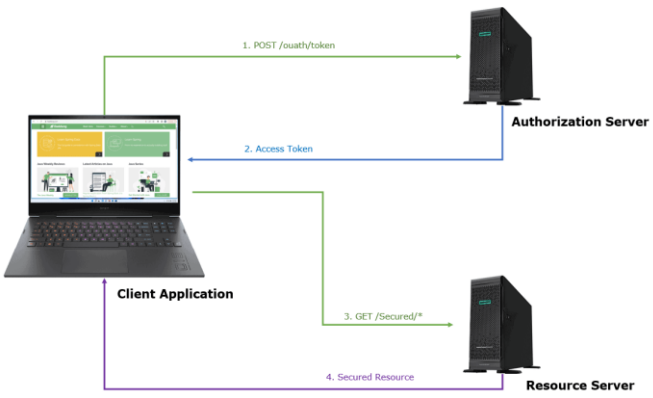

Рисунок 2.4 – Приклад автентифікації користувача [18]

У цьому випадку клієнтська програма може отримати доступ до ресурсу користувача за допомогою Access Tokens. Сервер авторизації видає Access Tokens після успішної автентифікації та згоди користувача.

Імовірність компрометації або викрадення Access Tokens зростає, чим довше він дійсний. Зберігаючи позитивний досвід користувача, поєднання довготривалих маркерів оновлення з короткочасними Access Tokens покращує безпеку.

У структурах авторизації OAuth 2.0 Refresh Tokens дозволяють розробникам керувати сеансами користувачів у рідних, веб-додатках і односторінкових програмах.

Крім того, вони дозволяють користувачам входити в систему та залишатися на зв’язку без надання своїх паролів протягом тривалого часу. Крім того, вони додають рівень безпеки для конфіденційних даних, покращуючи взаємодію з користувачем.

Refresh Tokens можуть тривати від кількох днів до кількох місяців.

Самі по собі Refresh Tokens не надають користувачеві доступу. Щоб уникнути непотрібної повторної автентифікації, вони подовжують тривалість сеансу.

Розглянемо приклад Access Tokens, який діє лише 8 хвилин для доступу до програми. Що станеться, якщо ми витратимо 16 хвилин на використання програми?

З точки зору безпеки, сеанс повинен закінчуватися через 8 хвилин. Користувач, який відчуває це, не принесе жодної користі, і існує ризик низького рівня задоволеності користувачів.

Використання Refresh Tokens в цій ситуації допоможе створити новий набір Access та Refresh Tokens через 8 хвилин, не вимагаючи від користувачів повторного введення своїх облікових даних.

Як правило, ми хочемо налаштувати термін життя Refresh Tokens на набагато довший. Крім того, програми можуть отримувати нові Access Tokens

протягом терміну дії Refresh Tokens, не вимагаючи від користувача повторної автентифікації.

Коли сервер авторизації помічає повторне використання Refresh Tokens, він миттєво скасовує Refresh Tokens та не дозволяє користувачеві отримати доступ до наступних запитів. Потрібна повторна автентифікація, оскільки неможливо визначити, чи Refresh Tokens надходить із надійного джерела.

Виявивши недійсне використання Refresh Tokens справжнім клієнтом чи зловмисником, сервер авторизації може виявити порушення, спричинене скомпрометованим Refresh Tokens. Це тому, що сервер авторизації зберігає старий Refresh Tokens після видачі нового.

Щоб реалізувати маркери, нам знадобляться дві моделі: одна для користувачів, а інша для Access та Refresh Tokens. Моделі користувачів містять такі дані, як імена користувачів, паролі, адреси електронної пошти та інші деталі.

З іншого боку, модель маркера включає значення Refresh Tokens, термін дії та ідентифікатор користувача. Ми можемо зберігати токени в кеші або захищеній реляційній базі даних.

Крім того, нам знадобиться набір секретних ключів , також відомих як відкритий/приватний ключі , для підпису та автентифікації токенів. Access та Refresh Tokens видаються лише користувачам із дійсними іменами користувачів і паролями.

Refresh Tokens використовуються лише між сервером авторизації та наданим клієнтом і мають бути приватними під час передачі за допомогою TLS.

Давайте тепер обговоримо, як налаштувати токени (рис. 2.5):

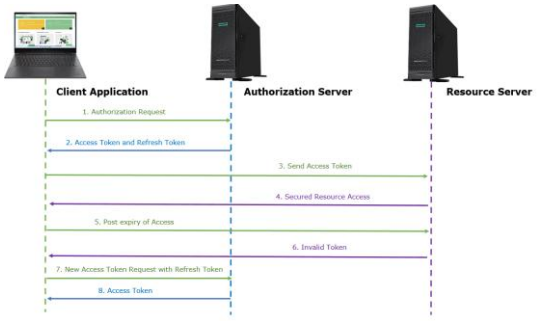

Рисунок 2.5 – Налаштування токенів [18]

Скажімо, клієнт запитує Access Tokens, пройшовши автентифікацію на сервері авторизації та надавши облікові дані, які підтверджують, що власник ресурсу має право використовувати ресурс.

Сервер авторизації перевіряє авторизацію та підтверджує особу авторизованого клієнта. Маркер доступу та маркер оновлення видаються, якщо вони законні. Клієнт повинен безпечно зберігати цей Refresh Tokens.

Тепер клієнт може запитувати сервер ресурсів про захищений доступ до ресурсів, наприклад API, і сервер ресурсів перевіряє Access Tokens. Якщо він дійсний, він повертає потрібний ресурс.

Коли програма запитує ресурс, але Access Tokens більше не дійсний, тоді:

– Сервер ресурсів має відмовитися виконувати запит і надішле відповідь з недійсним маркером.

– Програма надішле запит на новий Access Tokens за допомогою Refresh Tokens.

– Сервер авторизації використовуватиме попередньо наданий Refresh Tokens та надсилатиме новий Access Tokens.

РОЗДІЛ 3
РОЗРОБКА СИСТЕМИ

3.1 Опис схеми БД для зберігання даних користувачів та їх сесій

База даних для системи аутентифікації та управління сесіями користувачів розроблена для ефективного зберігання та швидкого доступу до даних користувачів, їхніх сесій, а також інформації про аутентифікаційні токени. Схема бази даних включає наступні основні таблиці:

Users. Зберігає основні дані користувачів. Кожен запис у цій таблиці містить:

– UserID (Primary Key): унікальний ідентифікатор користувача.

– Email: електронна пошта користувача, яка також використовується як логін.

– PasswordHash: хеш пароля для забезпечення безпеки зберігання паролів.

– CreatedDate: дата створення запису користувача.

– LastLoginDate: дата останнього входу користувача в систему.

Sessions. Таблиця для зберігання інформації про сесії користувачів, що включає:

– SessionID (Primary Key): унікальний ідентифікатор сесії.

– UserID (Foreign Key): посилання на таблицю користувачів, що ідентифікує власника сесії.

– RefreshToken: зберігання refresh токену, який використовується для відновлення access токенів.

– Expires: час закінчення дії refresh токену.

– CreatedDate: дата створення сесії.

– Revoked: поле, що вказує, чи було анульовано токен.

Для реалізації бази даних використовується реляційна СУБД MySQL. Важливо забезпечити:

– Належне індексування: належне індексування поля `Email` в таблиці `Users` для швидкого виконання запитів на аутентифікацію та валідацію, а також

індексація `UserID` у таблиці `Sessions` для ефективного пошуку сесій користувачів.

– Безпека даних: використання зашифрованого з'єднання до бази даних, регулярне оновлення СУБД та її компонентів для забезпечення захисту від вразливостей.

– Backup та відновлення: забезпечення можливостей для регулярного бекапу даних та їх відновлення в разі системних збоїв або критичних помилок.

Ця структура бази даних та її реалізація є ключовими для забезпечення надійної та безпечної роботи клієнтсько-серверної системи, що дозволяє виконувати аутентифікацію користувачів, управління сесіями та забезпечення безперервної роботи.

3.2 Реалізація серверного API. Авторизація, реєстрація та управління профілем користувача

У світі веб-розробки та сучасної автентифікації безпека має першочергове значення. Розробникам потрібні надійні механізми для забезпечення цілісності даних і автентифікації під час взаємодії користувачів. Веб-токени JSON (JWT) стали популярним і безпечним методом автентифікації та обміну інформацією. У цій публікації блогу ми розглянемо концепцію JWT, її структуру, переваги та найкращі практики впровадження.

Веб-токен JSON, який зазвичай називають JWT, є відкритим стандартом (RFC 7519) для безпечної передачі інформації між сторонами як об’єкт JSON. Токен має цифровий підпис, що забезпечує його автентичність і цілісність. JWT в основному використовуються для автентифікації користувачів, авторизації доступу до певних ресурсів і безпечного обміну інформацією.

Структура JWT (рис. 3.1):

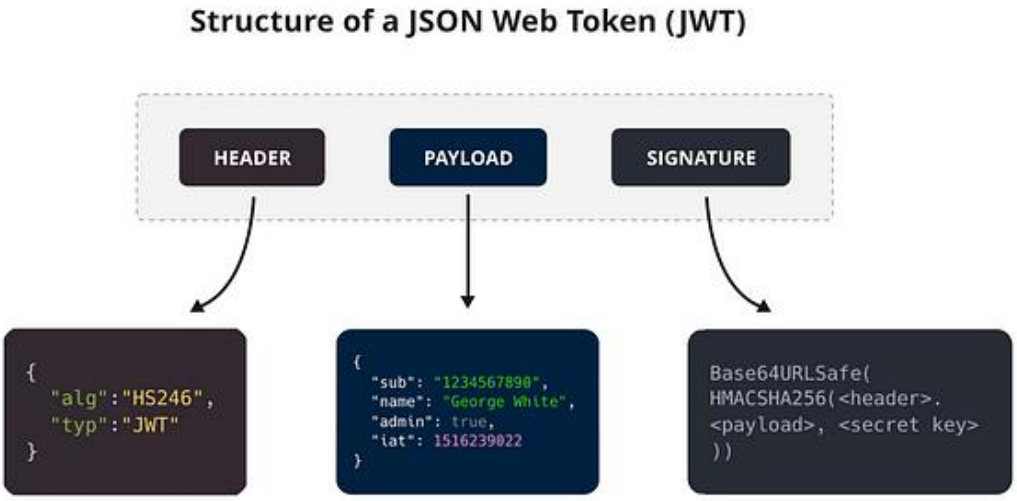

Рисунок 3.1 – Структура JWT

JWT складається з трьох частин, розділених крапками, які є рядками в кодуванні base64url:

1. Заголовок: зазвичай заголовок складається з двох частин – типу маркера (JWT) і використовуваного алгоритму підпису, наприклад HMAC SHA256 або RSA.

{

«alg» : «HS256» ,

«typ» : «JWT»

}

2. Корисне навантаження: корисне навантаження містить претензії, які є заявами про користувача чи інші дані. Позови бувають трьох видів: іменні, публічні та приватні.

{

«sub» : «user123» ,

«name» : «John Doe» ,

«admin» : true

}

3. Підпис: щоб створити частину підпису, вам потрібно взяти закодований заголовок, закодовану корисну інформацію, секрет і алгоритм, указаний у заголовку, а потім підписати це секретом. Підпис використовується для перевірки

того, що відправник JWT є тим, за кого він себе називає, і щоб переконатися, що повідомлення не було змінено в дорозі (рис. 3.2).

```
HMACSHA256(
  base64UrlEncode(header) + «.» +
  base64UrlEncode(payload),
  secret)
```

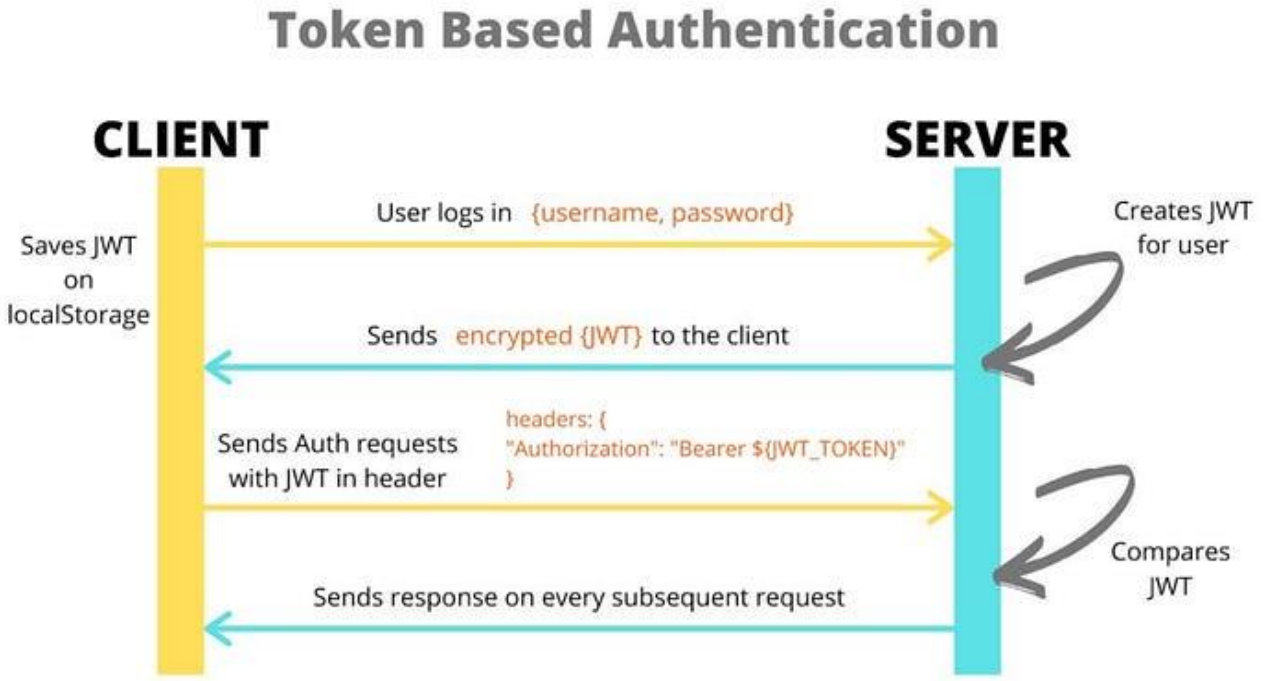

Рисунок 3.2 – Принцип роботи токена

Коли користувач входить або намагається отримати доступ до захищеного ресурсу, сервер генерує JWT після успішної автентифікації. Потім клієнт зберігає цей маркер, як правило, у локальному сховищі або файлі cookie. Для кожного наступного запиту, який потребує автентифікації, клієнт надсилає JWT у заголовках запиту. Сервер перевіряє токен, перевіряючи підпис і декодуючи корисне навантаження, щоб переконатися в автентичності та авторизації користувача.

Переваги JWT:

1. Без стану: оскільки JWT несуть у собі всю необхідну інформацію, серверу не потрібно підтримувати інформацію про сеанс. Це робить JWT без стану, що зменшує навантаження на сервер і спрощує масштабованість.

2. Компактність і ефективність: завдяки своїм компактним розмірам JWT підходять для передачі через мережі та легко аналізуються клієнтами.

3. Безпека: JWT мають цифровий підпис, що забезпечує цілісність даних і запобігає підробці. Використання алгоритмів шифрування ще більше підвищує безпеку.

4. Міждоменне спілкування: JWT можна використовувати в різних доменах або мікросервісах, оскільки вони не покладаються на файли cookie або сеанси на стороні сервера.

Найкращі практики впровадження JWT:

1. Безпечне зберігання: зберігайте JWT у файлах cookie лише HTTP, щоб запобігти доступу з JavaScript, зменшуючи ризик атак XSS.

2. Термін дії маркера: встановіть розумний термін дії для JWT, щоб обмежити часове вікно для можливого зловживання.

3. Відкликання токенів : наявність механізму відкликання або внесення в чорний список скомпрометованих токенів для підвищення безпеки.

4. Використовуйте HTTPS: переконайтеся, що весь зв’язок між клієнтом і сервером використовує HTTPS, щоб запобігти підслуховуванню та атакам типу «людина посередині».

5. Не зберігайте конфіденційні дані: уникайте зберігання конфіденційних даних у корисному навантаженні JWT, оскільки корисне навантаження легко читається після декодування за допомогою base64.

3.3 Клієнтська логіка та взаємодія з API

Axios – це HTTP-клієнт на основі Promise для node.js і браузера. Таким чином, він однаково добре працює у зовнішніх додатках JavaScript і внутрішніх серверах Node.

У цій статті показано, як використовувати Axios у простій програмі React. React – це бібліотека JavaScript для створення користувальницьких інтерфейсів,

тому тут ми будемо використовувати Axios для користувальницьких інтерфейсів на основі браузера.

Налаштування нашого проекту React.

Ми створимо проект, дотримуючись зазначеного вище. Давайте почнемо з виконання такої команди:

npx create-react-app react-axios-tutorial

Тепер ми перейдемо до каталогу проекту, виконавши:

cd react-axios-tutorial

Наразі структура проєкту виглядає так (рис. 3.3).

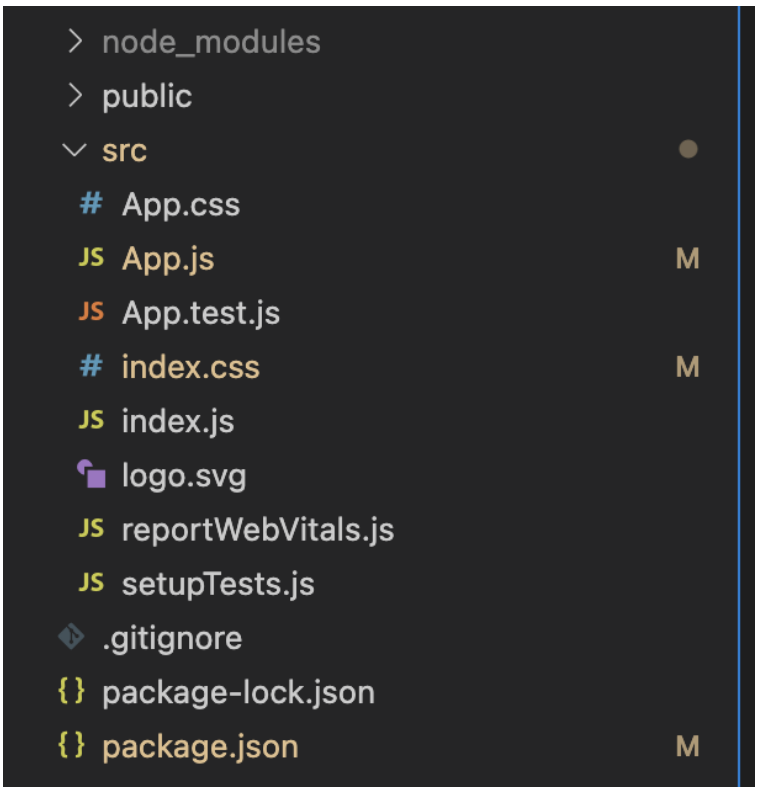

Рисунок 3.3 – Структура проєкту

Щоб встановити Axios, ми запускаємо таку команду:

npm i axios

Важливо перевірити залежності у package.jsonфайлі, щоб підтвердити, чи встановлено Axios (лістинг 3.1).

Поточний стан залежностей:

Лістинг 3.1 – Поточний стан залежностей

```
//...
  «dependencies»: {
    «@testing-library/jest-dom»: «^5.16.5»,
    «@testing-library/react»: «^13.3.0»,
    «@testing-library/user-event»: «^13.5.0»,
    «axios»: «^0.27.2»,
    «react»: «^18.2.0»,
    «react-dom»: «^18.2.0»,
    «react-scripts»: «5.0.1»,
    «web-vitals»: «^2.1.4»
  },
  //...
```

Кінець лістингу 3.1

Першим кроком є створення папки компонентів у каталозі src, щоб зробити запит GET. Ми переходимо до каталогу src і запускаємо наведений нижче код.

Компоненти mkdir.

У цьому каталозі ми створюємо Users.js файл і додаємо наведений нижче код(лістинг 3.2):

Лістинг 3.2 – Додатковий код до файлу Users.js

```
import React, { useEffect, useState } from «react»;
import axios from «axios»;
function Users() {
  const [post, setPost] = useState([]);
  useEffect(() => {
    axios.get(«https://jsonplaceholder.typicode.com/users»).then((data) => {
      console.log(data);
      setPost(data?.data);
```

```
    });
  }, []);
  return (
    <div>
      Users
      {post.map((item, i) => {
        return (
          <div key={i}>
            <p>{item?.name}</p>
          </div>
        );
      })}
    </div>
  );
}
export default Users;
```

Кінець лістингу 3.2

Імпорт React, useEffect і useState хуки. Ми також імпортуємо Axios, щоб робити HTTP-запити. У useEffect хуках ми використовуємо GET метод, щоб зробити GET запит до нашої кінцевої точки, а потім використовуємо then() метод, щоб повернути всі дані відповіді, які ми використовуємо для оновлення стану нашого користувача.

Доступ до масиву даних із адресою властивостей, компанією, електронною адресою, ідентифікатором, іменем, телефоном, іменем користувача та веб-сайтом. Потім властивості призначаються стану користувача та стають доступними в компоненті.

Є також інші відомості про запит, як-от код статусу під res.status або додаткові відомості всередині res.request (лістинг 3.3).

Далі додаємо Users компонент до app.js файлу

Лістинг 3.3 – Додавання Users компонент до app.js файлу

```
import Users from «./Components/Users»;
function App() {
  return (
    <div>
      <Users />
    </div>
  );
}
export default App;
```

Кінець лістингу 3.3

Наступним кроком буде запуск програми за допомогою:

npm start

Отримаємо зображення нижче (рис. 3.4).

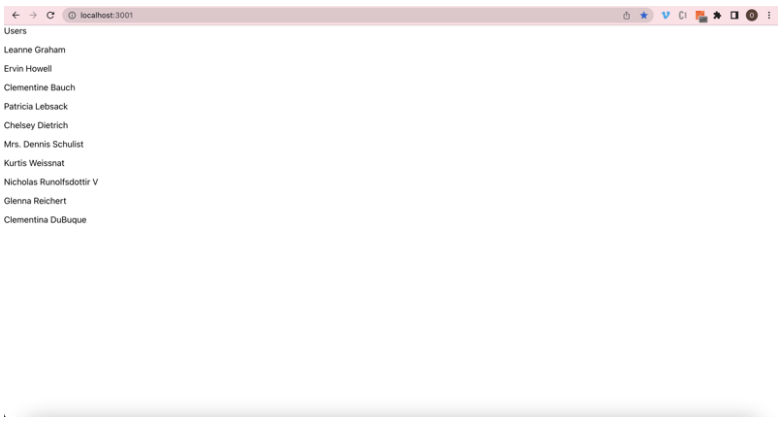

Рисунок 3.4 – Результат виконання npm start

Наступним кроком є використання Axios із POST. Нам потрібно створити новий компонент із назвою AddUser.js в каталозі компонентів.

touch AddUser.js

Наведений нижче код додається AddUser.js для створення форми (лістинг 3.4), яка дозволяє користувачеві вводити дані, а потім надсилає вміст до API:

Лістинг 3.4 – Створення форми

```
import React, { useState } from «react»;
import axios from «axios»;
function AddUser() {
  const [name, setName] = useState({
    name: « «,
  });
  const handleChange = (e) => {
    e.preventDefault();
    setName({
      name: e.target.value,
    });
  };
  const submitForm = (e) => {
    e.preventDefault();
    axios
      .post(`https://jsonplaceholder.typicode.com/users`, { name })
      .then((res) => {
        console.log(res);
        console.log(res.data);
      });
  };
  return (
    <div>
      <p>Add Users</p>
      <div>
        <form onSubmit={submitForm}>
          <label>
            User Name:
            <input type=«text» name=«name» onChange={handleChange} />
```

```
        </label>
        <button type=«submit»>Add</button>
      </form>
    </div>
  </div>
 );
}
export default AddUser;
```

Кінець лістингу 3.4

За допомогою SubmitForm функції ми зупиняємо дію форми за замовчуванням. Після цього ми оновлюємо, state щоб відобразити user введення. Метод POST дає нам той самий об’єкт відповіді з інформацією, яку ми можемо використовувати всередині then() методу.

Необхідно зафіксувати user вхідні дані, перш ніж ми зможемо виконати POST запит. Далі ми додаємо вхідні дані разом із POST запитом, який повертатиме відповідь. Після цього ми можемо отримати console.log відповідь, яка відображає user вхідні дані (лістинг 3.5).

Лістинг 3.5 – Додавання компоненту до app.js

```
import AddUser from «./Components/AddUser»;
import Users from «./Components/Users»;
function App() {
 return (
  <div>
   <Users />
   <AddUser />
  </div>
 );
}
```

```
export default App;
```

Кінець лістингу 3.5

Наступним кроком є використання Axios із методом PUT. Нам потрібно буде створити новий компонент із назвою UpdateUser в каталозі компонентів.

```
touch UpdateUser.js
```

Наведений нижче код додається UpdateUser для створення форми (лістинг 3.6), яка дозволяє вводити дані користувачам, а потім оновлювати вміст до API:

Лістинг 3.6 – Код для введення даних користувачами

```
import React, { useEffect, useState } from «react»;
import axios from «axios»;
function UpdateUser() {
  const [state, setState] = useState({
    Name: ««,
    userName: ««,
  });
  const handleChange = (evt) => {
    const value = evt.target.value;
    setState({
      ...state,
      [evt.target.name]: value,
    });
  };
  const submitForm = (e) => {
    e.preventDefault();
    console.log(e);
    console.log(state);
```

```
  axios
    .put(`https://jsonplaceholder.typicode.com/users/1`, { state })
    .then((res) => {
      console.log(res);
      console.log(res.data);
    });
};
return (
  <div>
    <p>Add Users</p>
    <div>
      <form onSubmit={submitForm}>
        <label>
          User Name:
          <input
            type=«text»
            name=«Name»
            placeholder=«name»
            value={state.Name}
            onChange={handleChange}
          />
          <input
            type=«text»
            name=«userName»
            placeholder=«username»
            value={state.userName}
            onChange={handleChange}
          />
        </label>
        <button type=«submit»>Add</button>
```

```
        </form>
      </div>
    </div>
  );
}
export default UpdateUser;
```

Кінець лістингу 3.6

У наведеному вище коді використовуємо PUT метод із Axios. Як і у випадку з POST методом, ми включаємо властивості, які хочемо додати до оновленого ресурсу.

Знову ж таки, за допомогою цього then() методу дані оновлюються в JSX (лістинг 3.7).

Лістинг 3.7 – Додавання UpdateUser компонент до App.js файлу

```
import AddUser from «./Components/AddUser»;
import UpdateUser from «./Components/UpdateUser»;
import Users from «./Components/Users»;
function App() {
  return (
    <div>
      <Users />
      <AddUser />
      <UpdateUser />
    </div>
  );
}
export default App;
```

Кінець лістингу 3.7

Використовуючи Delete метод і передаючи URL як параметр, ми побачимо, як видалити елементи з API. Нам потрібно створити новий компонент, який називається RemoveUser.js в рамках проекту React.

Щоб видалити користувача (лістинг 3.8), створюємо RemoveUser.js та додаємо такий код:

Лістинг 3.8 – Код для видалення користувача

```
import React, { useState } from «react»;
import axios from «axios»;
function RemoveUser() {
  const [state, setState] = useState(« «);
  const handleChange = (e) => {
    setState({ id: e.target.value });
  };
  const handleRemove = (evt) => {
    evt.preventDefault();
    axios
      .delete(`https://jsonplaceholder.typicode.com/users/${state.id}`)
      .then((response) => {
        console.log(response);
        console.log(response.data);
      });
  };
  return (
    <div>
      Remove User
      <div>
        <form onSubmit={handleRemove}>
          <label>
            User ID:
```

```
          <input type=«number» name=«id» onChange={handleChange} />
        </label>
        <button type=«submit»>Delete</button>
      </form>
    </div>
  </div>
 );
}
export default RemoveUser;
```

Кінець лістингу 3.8

Знову ж таки, response об’єкт містить інформацію про запит. Після надсилання форми ми можемо console.log знову отримати цю інформацію (лістинг 3.9). App.js файл повинен містити цей компонент:

Лістинг 3.9 – Надсилання форми console.log

```
import AddUser from «./Components/AddUser»;
import RemoveUser from «./Components/RemoveUser»;
import UpdateUser from «./Components/UpdateUser»;
import Users from «./Components/Users»;
function App() {
 return (
  <div>
   <Users />
   <AddUser />
   <UpdateUser />
   <RemoveUser />
  </div>
 );
}
```

```
export default App;
```

Кінець лістингу 3.9

Під час оформлення запиту можуть виникати деякі помилки.

Проблема з передачею даних, був запит на неправильну кінцеву точку або проблема з мережею. Щоб імітувати помилку, надсилаємо запит до API кінцевої точки, яка не існує: /users/obmm (лістинг 3.10).

Щоб усунути помилку, створюємо Errorhandling.js та додаємо такий код:

Лістинг 3.10 – Усунення помилки

```
import axios from «axios»;
import React, { useEffect, useState } from «react»;
function Errorhandling() {
  const [users, setUsers] = useState([]);
  const [error, setError] = React.useState(null);
  useEffect(() => {
    axios
      .get(`https://jsonplaceholder.typicode.com/posts/obmm`)
      .then((response) => {
        setUsers(response.data);
      })
      .catch((error) => {
        setError(error);
      });
  }, []);
  if (error) return `Error: ${error?.message}`;
  if (!users) return «No user!»;
  return (
    <div>
      Errorhandling
```

```
      <div>
        Users
        {users.map((item, i) => {
          return (
            <div key={i}>
              <p>{item?.name}</p>
            </div>
          );
        })}
      </div>
    </div>
  );
}
export default Errorhandling;
```

Кінець лістингу 3.10

У результаті Axios видасть помилку замість виконання then() методу.

Ми використовуємо цю функцію, щоб попередити наших користувачів про помилку, беручи дані про помилку та переводячи їх у стан. Таким чином, якщо станеться помилка, з’явиться повідомлення про помилку.

Користувача сповіщають про помилку шляхом розміщення даних про помилку в стані. Якщо виникла помилка, з’явиться повідомлення. Після запуску цього коду ми побачимо: «Помилка: запит не виконано з кодом статусу 404» (лістинг 3.11).

Наш app.js файл повинен містити цей компонент:

Лістинг 3.11 – «Помилка: запит не виконано з кодом статусу 404»

```
import AddUser from «./Components/AddUser»;
import Errorhandling from «./Components/Errorhandling»;
import RemoveUser from «./Components/RemoveUser»;
```

```
import UpdateUser from «./Components/UpdateUser»;
import Users from «./Components/Users»;
function App() {
  return (
    <div>
      <Users />
      <AddUser />
      <UpdateUser />
      <RemoveUser />
      <Errorhandling />
    </div>
  );
}
export default App;
```

Кінець лістингу 3.11

У цьому розділі ми налаштуємо базовий екземпляр із URL-адресою та іншими елементами конфігурації (лістинг 3.12).

Першим кроком є створення окремого файлу з назвою api.js:

Лістинг 3.12 – Створення файлу api.js

```
import axios from 'axios';
export default axios.create({
  baseURL: `http://jsonplaceholder.typicode.com/`
});
```

Кінець лістингу 3.12

Ми будемо використовувати API файл у RemoveUser.js компоненті (лістинг 3.13).

Імпортуємо новий екземпляр таким чином:

Лістинг 3.13 – Імпорт нового екземпляру

```
import React, { useState } from 'react'
import axios from 'axios';
import API from './API.js';
function RemoveUser() {
  // ...
 const handleRemove = (evt)=>{
    evt.preventDefault();
       API.delete(`users/${state.id}`)
      .then(response => {
        console.log(response);
        console.log(response.data);
      })
}
  // ...
}
export default RemoveUser
```

Кінець лістингу 3.13

Більше не потрібно вводити повну URL-адресу кожного разу, коли ми хочемо отримати доступ до іншої API кінцевої точки, оскільки http://jsonplaceholder.typicode.com/ тепер це базова URL-адреса.

Обіцянки вирішуються за допомогою await ключового слова, яке повертає їх value (лістинг 3.14). Після цього value можна призначити змінну.

Лістинг 3.14 – Призначення змінної value

```
import React, { useState } from 'react'
import axios from 'axios';
import API from './API.js';
function RemoveUser() {
```

```
  // ...
 const handleRemove = (evt)=>{
    evt.preventDefault();
      const response = await API.delete(`users/${this.state.id}`);
    console.log(response);
    console.log(response.data);
}
  // ...
}
export default RemoveUser
```

Кінець лістингу 3.14

У наведеному вище прикладі then метод замінено. У результаті promise було виконано, в результаті чого value зберігається в response змінній.

3.4 Тестування та валідація системи

Серверна частина системи вимагає ретельного тестування для забезпечення стабільності, безпеки та надійності. Основні методи тестування серверної частини включають:

– Використання бібліотек таких як Jest або Mocha для написання та виконання юніт-тестів, які перевіряють функції окремих компонентів сервера. Це дозволяє виявити та виправити помилки на ранніх етапах розробки.

– Перевірка взаємодії між різними компонентами системи, включаючи базу даних та інші зовнішні сервіси, для забезпечення їх правильної взаємодії.

– Використання інструментів, як-от JMeter або LoadRunner, для симуляції одночасного доступу великої кількості користувачів, що дозволяє оцінити продуктивність та масштабованість сервера.

Клієнтська частина також вимагає комплексного тестування, особливо в аспектах користувацького інтерфейсу та взаємодії з сервером:

– Використання React Testing Library або Enzyme для тестування React компонентів на правильність їх поведінки та відображення.

– Перевірка функціональності клієнтських функцій, таких як авторизація, реєстрація, оновлення даних користувача. Використання Selenium або Cypress для автоматизації дій користувачів в браузері.

– Переконатися, що веб-додаток правильно працює в різних браузерах та на різних пристроях.

Валідація системи зосереджена на забезпеченні відповідності вимогам безпеки та вимогам користувача:

– Забезпечення, що вхідні дані користувачів правильно обробляються та не призводять до помилок чи уразливостей.

– Використання інструментів, таких як OWASP ZAP або Postman, для тестування API на вразливості та забезпечення їх захисту від зловмисних атак.

– Переконатися, що системи контролю доступу працюють правильно і лише авторизовані користувачі мають доступ до чутливих даних та операцій.

Враховуючи ці аспекти, можна значно підвищити якість та надійність веб-додатку, забезпечуючи позитивний досвід для користувачів та високий рівень безпеки.

Форми авторизації (рис. 3.5) та реєстрації (рис. 3.4), доступні тільки для неавторизованого користувача. Для валідації використовується бібліотека formik. Валідація відбувається як на клієнтській частині, так і на серверній. Поле e-mail – повинно бути не пусте і відповідати патерну, поле password – не пусте і більше, ніж 4 символи. Після успішної клієнтської валідації, відпраляється запит на сервер з введеними даними, де сервер в свою чергу виконує такі дії:

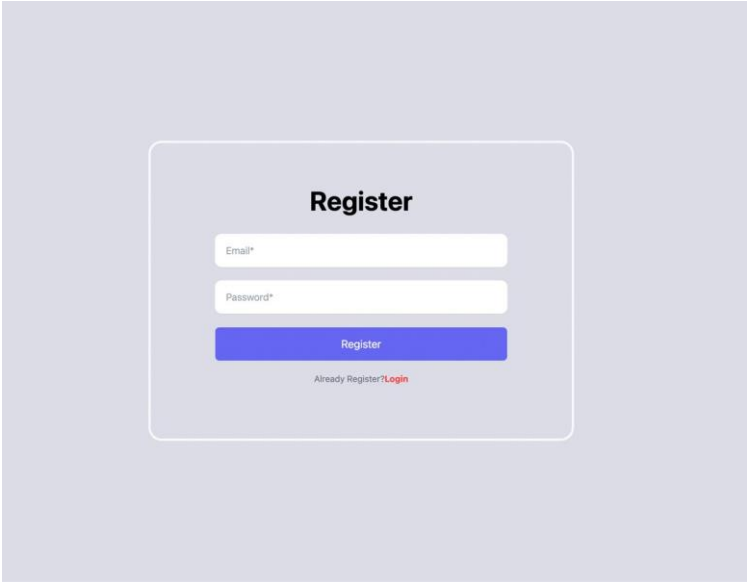

Рисунок 3.4 – Форма реєстрації

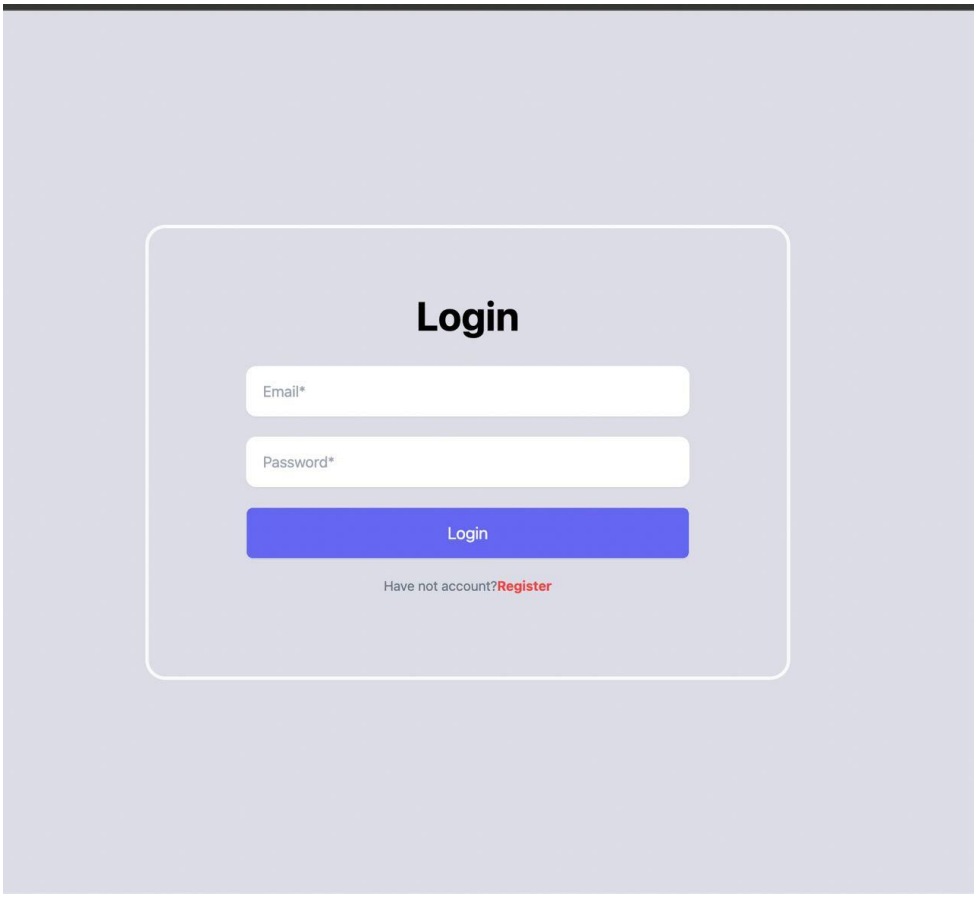

Рисунок 3.5 – Форма авторизації

– В випадку логіну (рис. 3.6) (/api/login), валідує передані дані, перевіряє чи існує переданий e-mail в базі даних, якщо існує, то звіряє пароль. Якщо ж він співпадає, то сервер надсилає до клієнта успішну відповідь. Якщо хоч один з

попередніх пунктів не виконується, клієнтська частина отримує помилку і логін не відбувається.

– В випадку реєстрації (рис. 3.7) (/api/register), валідує передані дані, перевіряє чи даного e-mail немає в базі даних, якщо ж немає повертається успішна відповідь і користувач входить в систему, в інакшому випадку користувач отримує помилку.

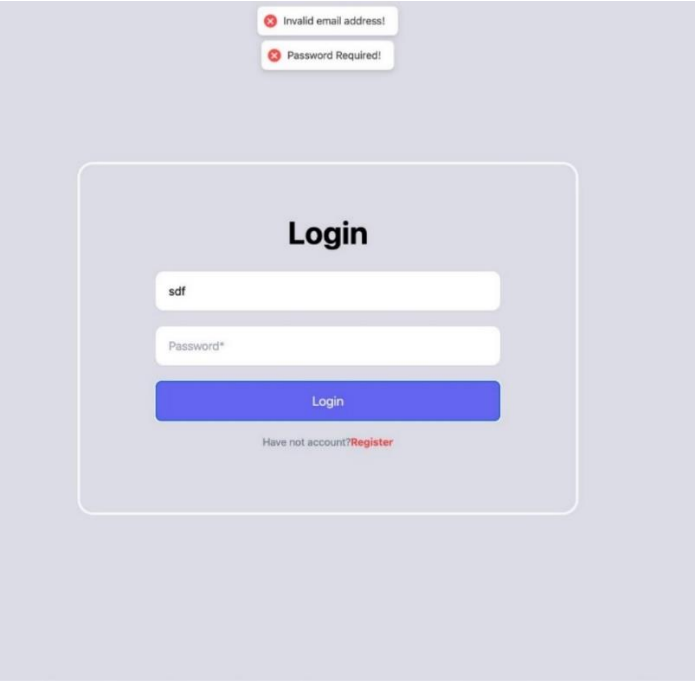

Рисунок 3.6 – Перевірка даних при авторизації

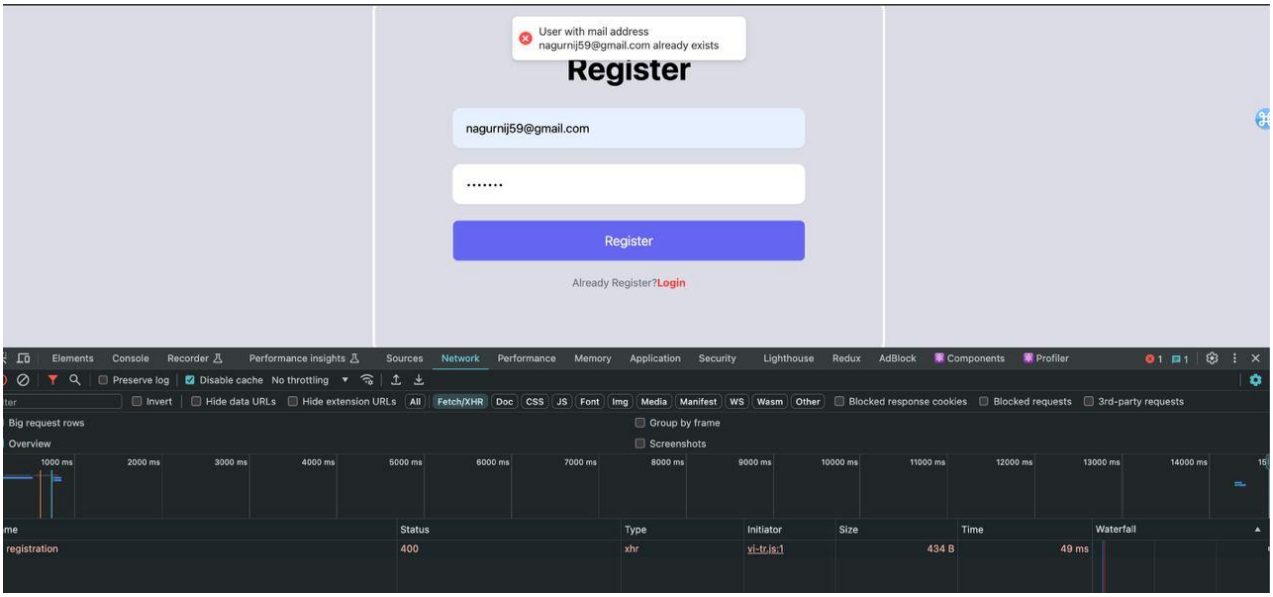

Рисунок 3.7 – Перевірка даних при реєстрації

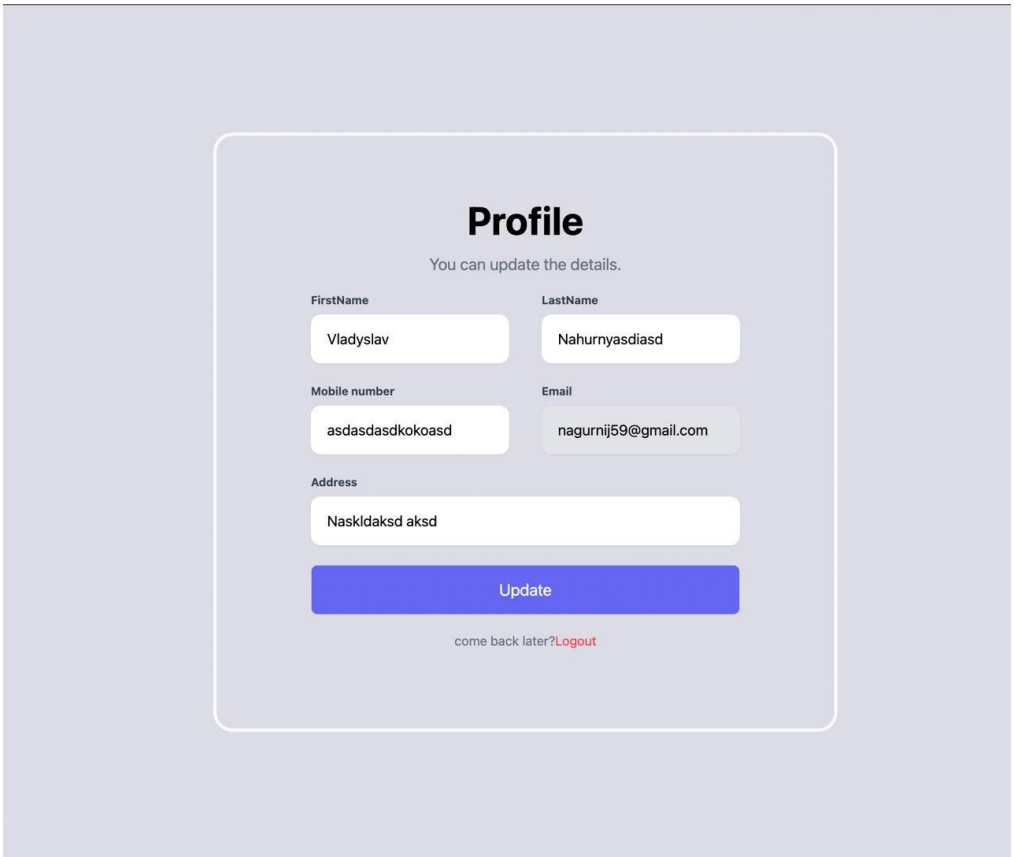

Рисунок 3.8 – Редагування профілю авторизованого користувача

Компонент редагування профілю (рис. 3.8) доступний тільки для авторизованого користувача. Кожен раз, коли людина потрапляє на сайт, відправляється запит на сервер з токеном (accessToken), який був наданий сервером при авторизації, для того, щоб перевірити чи сесія користувача не вичерпалась. Якщо ж сесія ще досі активна, користувач може оновлювати свої дані в формі, яка відправляє запит /api/updateProfile. Цей роут є захищеним accessToken-ом, щоб кожен користувач міг редагувати, тільки свої дані.

Кнопка logout – відповідає за завершення поточної сесії користувача. Відбувається запит на сервер (/api/logout), при якому з бази даних видаляється токен (refreshToken), і користувач потрапляє на екран для неавторизованого.

ВИСНОВКИ

Розроблена система веб-сервера відкриває нові можливості для підвищення рівня безпеки веб-додатків і може бути рекомендована для широкого впровадження в комерційних та державних інформаційних системах. Результати дослідження та розробки можуть слугувати основою для подальших удосконалень у галузі кібербезпеки та веб-технологій.

Аналіз існуючих рішень веб-серверів виявив, що багато сучасних систем авторизації ще недостатньо захищені від злому і фішингу. Виявлені недоліки включають слабкі точки в шифруванні та управлінні сесіями. Проте, також були визначені передові практики, що включають використання двофакторної аутентифікації та застосування сучасних криптографічних протоколів, які слугували основою для подальшої розробки.

Вивчення захищених носіїв показало, що використання таких носіїв, як апаратні ключі аутентифікації, може значно підвищити безпеку системи. Інтеграція цих носіїв у систему авторизації веб-сервера дозволила створити більш надійну архітектуру, здатну протистояти спробам несанкціонованого доступу.

Розробка архітектури веб-сервера забезпечила створення ефективної структури, яка включає в себе не тільки традиційні функції веб-сервера, але й розширені можливості для забезпечення приватності та безпеки даних. Це досягнуто через використання шифрування даних у транзиті та на стороні клієнта.

Імплементація системи авторизації на базі захищеного носія була успішно реалізована і випробувана. Захищені носії використовувалися для генерації та зберігання ключів аутентифікації, забезпечуючи високий рівень безпеки.

Тестування системи підтвердило її ефективність і надійність у захисті приватних даних і управлінні ідентифікаційними даними. Різні сценарії тестування, включаючи спроби злому та фішингу, продемонстрували стійкість системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Paro, A. et al. Hackers leaked 22 million records on the dark web in 2020. URL: https://securityboulevard.com/2021/01/hackers-leaked-22-million-records-on-the-dark-web-in-2020/ (дата звернення: 03.05.2024).

2. Meyer, B. Most common passwords 2022. URL: https://cybernews.com/best-password-managers/most-common-passwords/ (дата звернення: 03.05.2024).

3. Seta, H., Wati, T., Kusuma, I. C. Implement Time Based One Time Password and Secure Hash Algorithm 1 for Security of Website Login Authentication. 2019 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS), 2019, pp. 115-120. doi: 10.1109/ICIMCIS48181.2019.8985196 (дата звернення: 03.02.2024).

4. Kennedy, W., Olmsted, A. Three factor authentication. 12th International Conference for Internet Technology and Secured Transactions (ICITST), pp. 212-213.

5. Saito, T. et al. An Authorization Scheme Concealing Client's Access from Authentication Server. 10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), pp. 593-598. doi: 10.1109/IMIS.2016.110 (дата звернення: 03.02.2024).

6. Bansal, C., Bhargavan, K., Maffeis, S. Discovering Concrete Attacks on Website Authorization by Formal Analysis. IEEE 25th Computer Security Foundations Symposium, pp. 247-262. doi: 10.1109/CSF.2012.27 (дата звернення: 03.02.2024).

7. Pascariu, C., Bacivarov, I. C. Detecting Phishing Websites through Domain and Content Analysis. 2021 13th International Conference on Electronics, Computers and Artificial Intelligence (ECAI), 2021, pp. 1-4. doi: 10.1109/ECAI52376.2021.9515165 (дата звернення: 03.02.2024).

8. Subrayan, S., Mugilan, S., Sivanesan, B., Kalaivani, S. Multi-factor Authentication Scheme for Shadow Attacks in Social Network. International Conference on Technical Advancements in Computers and Communications (ICTACC), pp. 36-40. doi: 10.1109/ICTACC.2017.19 (дата звернення: 03.02.2024).

9. Reeder, R., Schechter, S. When the Password Doesn't Work: Secondary Authentication for Website. IEEE Security & Privacy, vol. 9, no. 2, pp. 43-49. doi: 10.1109/MSP.2011.1 (дата звернення: 03.02.2024).

10. ALSaleem, B. O., Alshoshan, A. I. Multi-Factor Authentication to Systems Login. 2021 National Computing Colleges Conference (NCCC), 2021, pp. 1-4. doi: 10.1109/NCCC49330.2021.9428806 (дата звернення: 03.02.2024).

11. Tatlı, E.İ. Cracking More Password Hashes With Patterns. IEEE Transactions on Information Forensics and Security, vol. 10, no. 8, pp. 1656-1665. doi: 10.1109/TIFS.2015.2422259 (дата звернення: 03.02.2024).

12. Ahmed, S., Mahmood, Q. An authentication based scheme for applications using JSON web token. 2019 22nd International Multitopic Conference (INMIC), 2019, pp. 1-6. doi: 10.1109/INMIC48123.2019.9022766 (дата звернення: 03.02.2024).

13. Laatansa, R., Saputra, R., Noranita, B. Analysis of GPGPU-Based Brute-Force and Dictionary Attack on SHA-1 Password Hash. 2019 3rd International Conference on Informatics and Computational Sciences (ICICoS), 2019, pp. 1-4. doi: 10.1109/ICICoS48119.2019.8982390 (дата звернення: 03.02.2024).

14. De Guzman, F.E., Gerardo, B. D., Medina, R. P. Implementation of Enhanced Secure Hash Algorithm Towards a Secured Web Portal. 2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS), 2019, pp. 189-192. doi: 10.1109/CCOMS.2019.8821763 (дата звернення: 03.02.2024).

15. Zhang, H., Zou, F. A Survey of the Dark Web and Dark Market Research. 2020 IEEE 6th International Conference on Computer and Communications (ICCC), 2020, pp. 1694-1705. doi: 10.1109/ICCC51575.2020.9345271 (дата звернення: 03.02.2024).

16. Pant, P. et al. Blockchain for AI-Enabled Industrial IoT with 5G Network. 2022 14th International Conference on Electronics, Computers and Artificial Intelligence (ECAI), 2022, pp. 1-4. doi: 10.1109/ECAI54874.2022.9847428 (дата звернення: 03.02.2024).

17. Joby, P. P. Expedient information retrieval system for web pages using the natural language modeling. Journal of Artificial Intelligence 2, no. 02 (2020): 100-110.

18. Kumar, H. et al. Rainbow table to crack password using MD5 hashing algorithm. IEEE Conference on Information & Communication Technologies, pp. 433-439. doi: 10.1109/CICT.2013.6558135 (дата звернення: 03.02.2024).

19. Gauravaram, P. Security Analysis of salt||password Hashes. International Conference on Advanced Computer Science Applications and Technologies (ACSAT), pp. 25-30. doi: 10.1109/ACSAT.2012.49 (дата звернення: 03.02.2024).

20. Kharod, S., Sharma, N., Sharma, A. An improved hashing based password security scheme using salting and differential masking. 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions), pp. 1-5. doi: 10.1109/ICRITO.2015.7359225 (дата звернення: 03.02.2024).

21. Zhang, H., Zou, F. A Survey of the Dark Web and Dark Market Research. 2020 IEEE 6th International Conference on Computer and Communications (ICCC), 2020, pp. 1694-1705. doi: 10.1109/ICCC51575.2020.9345271 (дата звернення: 03.02.2024).

22. Chen, H. Dark Web: Exploring and Mining the Dark Side of the Web. European Intelligence and Security Informatics Conference, pp. 1-2. doi: 10.1109/EISIC.2011.78 (дата звернення: 03.02.2024).

23. Sharma, S., Rajawat, A. S. A secure privacy preservation model for vertically partitioned distributed data. International Conference on ICT in Business Industry & Government (ICTBIG), pp. 1-6. doi: 10.1109/ICTBIG.2016.7892653 (дата звернення: 03.02.2024).

24. Pant, P. et al. Blockchain for AI-Enabled Industrial IoT with 5G Network. 2022 14th International Conference on Electronics, Computers and Artificial Intelligence (ECAI), 2022, pp. 1-4. doi: 10.1109/ECAI54874.2022.9847428 (дата звернення: 03.02.2024).

25. Joby, P. P. Expedient information retrieval system for web pages using the natural language modeling. Journal of Artificial Intelligence 2, no. 02 (2020): 100-110.