Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра комп’ютерних систем та мереж

Пояснювальна записка
до кваліфікаційної роботи бакалавра
за спеціальністю 123 «Комп’ютерна інженерія»

на тему: ОНЛАЙН МЕССЕНДЖЕР НА ОСНОВІ ВИКОРИСТАННЯ TELEGRAM

Проектував _____ Д. О. Яцик

Керівник роботи _____ Д. І. Кузнєцов

Нормоконтроль _____ Д. І. Кузнєцов

Завідувач кафедри _____ А. І. Купін

Кривий Ріг
2025

Криворізький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерних систем та мереж

Ступінь вищої освіти бакалавр
Спеціальність 123 «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри, голова циклової комісії

_____ А. І. Купін

"____" _____ 2025__ року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

_____Яцик Денис Олегович_____
(прізвище, ім'я, по батькові)

1. Тема роботи Онлайн мессенджер на основі використання Telegram_____
_____
_____

керівник роботи Д. І. Кузнєцов, к.т.н, доцент _____,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від "___"_____ 20__року №_____

2. Строк подання студентом роботи 05.06.2025_____

3. Вихідні дані до роботи API-endpoints, GraphQL, Zunstad, Next.js _____
_____
_____
_____
_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Технології та інструменти веб-розробника, функціональна логіка та структура проєкту_____
_____
_____
_____
_____

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) слайди презентації (16 штук)_____
_____
_____
_____
_____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 15.12.2024_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк виконання етапів роботи	Примітка
1	Збір загальної інформації про роботу DNS	15.12.2024	
2	Збір інформації проте, як працює браузер	22.12.2024	
3	Збір інформації про каскадність у CSS	02.01.2025	
4	Збір інформації про протоколи HTTP та HTTPS	13.01.2025	
5	Збір інформації про refresh та access токени	19.01.2025	
6	Збір інформації про фреймворки Next.js та Nest.js	27.01.2025	
7	Збір інформації про Git та Zustand	03.02.2025	
8	Збір інформації про Docker та архітектуру FSD	13.02.2025	
9	Написання коду для застосунку	28.02.2025	
10	Написання загальних висновків	10.03.2025	

Студент _____ _____
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка: 76 сторінок, 93 рисунків, 9 таблиць, 22 використаних джерел.

Мета проектування – створити онлайн месенджер з реалізацією основного функціоналу: реєстрація, відправка повідомлень, створення каналів, додавання реакцій, створення посилань-запрошень. Ознайомитися з популярними фреймворками Next.js та Nest.js, також з GraphQL і PostgreSQL. Навчитися базово користуватися Zustand, Docker та Git.

Проект складається з двох розділів.

У першому розділі подано опис основних технологій, необхідних для створення онлайн месенджера. Також наведено більш детальну інформацію про TypeScript, Next.js та Next.js.

У другому розділі описується використана архітектура клієнтської частини – FSD. Ознайомлення з бізнес-логікою додатку та наведення практичних доказів коректної роботи. Створюються контейнери використовуючи Docker. Завантаження коду до віддаленого репозиторію.

Explanatory note: 76 pages, 93 figures, 9 tables, 22 references.

The purpose of the project is to create an online messenger with the implementation of the main functionality: registration, sending messages, creating channels, adding reactions, creating invitation links. Get acquainted with the popular Next.js and Nest.js frameworks, as well as GraphQL and PostgreSQL. Learn how to use Zustand, Docker, and Git in a basic way.

The project consists of two sections.

The first section describes the basic technologies needed to create an online messenger. It also provides more detailed information about TypeScript, Next.js, and Next.js.

The second section describes the client-side architecture used - FSD. It introduces the business logic of the application and provides practical evidence of correct operation. Creating containers using Docker. Uploading the code to a remote repository.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

БД – база даних;

ООП – об’єктно-орієнтоване програмування;

API – Application Programming Interface (програмний інтерфейс програми);

CORS – Cross-Origin Resource Sharing (спільне використання ресурсів з різних джерел);

CSS – Cascading Style Sheets (каскадна таблиця стилів);

CSSOM – CSS Object Model (CSS об’єктна модель);

CSRF – Cross-Site Request Forgery (міжсайтова підробка запиту);

DI – Dependency Injection (впровадження залежності);

DOM – Document Object Model (об’єктна модель документа);

DRY – Don`t Repeat Yourself (не повторюй себе);

DTO – Data Transfer Object (об’єкт передачі даних);

DNS – Domain Name System (система доменних імен);

ER-Diagram – Entity-Relationship Diagram (діаграма сутніть-зв’язок);

FSD – Feature-Sliced Design (функціональний дизайн);

HTML – HyperText Markup Language (мова гіпертекстової розмітки);

HTTP/S – HyperText Transfer Protocol (Secure) (протокол передачі гіпертексту);

ID – Identifier (індентифікатор);

IP – Internet Protocol (Інтернет протокол);

JSON – JavaScript Object Notation (запис об’єктів JavaScript);

JWT – JSON Web Token (веб-токе у форматі JSON);

NPM – Node Package Manager (менеджер пакетів Node);

REST – Representational State Transfer (передача репрезентативного стану);

RSC Payload – React Server Components Payload (навантаження компонентів React Server);

SASS – Syntactically Awesome Stylesheets (синтаксично чудові таблиці стилів);

SSL – Secure Sockets Layer (рівень захисту сокетів);

TLD – Top-Level Domain (домен верхнього рівня);

TLS – Transport Layer Security (захист транспортного рівня);

UI – User Interface (користувацький інтерфейс);

URL – Uniform Resource Locator (уніфікована адреса ресурсу);

XML – eXtensible Markup Language (розширювана мова розмітки).

Вступ

Швидкий розвиток Інтернету сприяв появі багатьох цікавих застосунків. Один із найпопулярніших серед них є онлайн месенджер. Це сервіси, які дозволяють людям обмінюватися повідомленнями в реальному часі, незалежно від їхнього географічного розташування. Єдина вважлива умова - наявність доступу до Інтернету.

Популярними онлайн месенджерами щодня користуються мільйони людей. Бази даних таких сервісів зберігають багато важливих даних про своїх користувачів. Тому сучасні онлайн месенджери повинні забезпечувати три ключові принципи інформаційної безпеки: конфенденційність, цілісність, та доступність.

Основна архітектура таких додатків – клієнт-серверна. На стороні сервера відбувається взаємодія з базою даних і виконується бізнес-логіка. Клієнтська частина отримує дані від сервера та на їх основі формує зображення інтерфейсу на екрані користувача.

Метою курсового проєкту є розробка онлайн-месенджера з базовим функціоналом: реєстрація користувача, створення каналів, обмін повідомленнями, генерація публічних і приватних запрошень.

Спершу буде розглянуто, як працює система доменних імен (DNS). Далі ознайомимося, як браузер обробляє HTML та CSS. Визначимо різницю між протоколами HTTP та HTTPS. Вивчимо архітектурний стиль REST API, потім порівняємо його з GraphQL. Наступним кроком буде ознайомлення з технологіями, які забезпечують миттєву доставку повідомлень. Також буде пояснено, для чого використовується TypeScript. Після цього розглянемо основи роботи з Git та Docker.

У проєкті будуть використовуватися два основні фреймворки: Next.js та Nest.js. Їх використання забезпечить створення гнучкого та легко масштабованого онлайн месенджеру. Next.js змушує розробника використовувати декларативний підхід для написання застосунків, забезпечує оптимізацію шрифтів та зображень, а також підтримує серверний і клієнтський рендеринг компонентів. У свою чергу, Nest.js є надбудовою над Node.js та Express.js і реалізує патерн Dependency Injection. У нього вбудовано багато бібліотек для налаштування захисту застосунку.

Популярним варіантом серед баз даних є PostgreSQL. Вона відповідає вимогам ACID і належить до реляційного типу. Завдяки підтримці зовнішніх і первинних ключів, PostgreSQL дозволяє легко реалізовувати зв’язки між таблицями в базі даних, що робить її зручною для складних проєктів.

1. ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТИ ВЕБ-РОЗРОБНИКА

1.1 Введення до DNS

Коли користувач відкриває браузер і пише назву сайту, браузеру потрібно перетворити цей набір символів на IP-адресу, і для цього використовується DNS.

Система доменних імен (DNS), це записна книжка всього Інтернету. Веб-браузери взаємодіють через Інтернет-протокол (IP-адреси). DNS переводить доменні імена в зрозумілі для комп'ютера IP-адреса, щоб браузери могли завантажувати необхідні ресурси. Завдяки цій технології людям не потрібно запам'ятовувати IP-адреси сайтів.

1.1.1 Принцип роботи DNS

Тепер давайте розберемося, як саме проходить увесь процес перетворення доменного імені на IP-адрес. Для прикладу розглянемо ситуацію, коли клієнт вводить доменне ім'я www.udemy.com. Тоді виконуються такі кроки:

а) (Користувач) - відкриває браузер і вводить доменне ім'я в браузері;

б) (Браузер) - відправляє запит до DNS сервера, який вказаний, як DNS сервер у налаштуваннях мережевого підключення конкретного комп'ютера;

в) (DNS сервер) - відправляє запит до кореневого DNS сервера, який відповідає за зону доменних імен .com;

Г) (Кореневий DNS сервер) - повертає інформацію об іменнах або IP-адресах DNS серверов відповідних за зону .com;

ґ) (DNS сервер) – звертається до DNS сервера відповідного за .com і просить його повернути всі іменна або IP-адреса серверов відповідних за udemy.com;

д) (DNS сервер відповідний за .com) – повертає список DNS серверов відповідних за udemy.com;

е) (DNS сервер) – звертається до DNS серверов відповідних за домен udemy.com;

є) (DNS сервер відповідний за udemy.com) – повертає доменне ім'я www.udemy.com;

ж) (DNS сервер) – повертає відповідь до клієнта.

Весь процес запиту до різних серверів триває секунди і зазвичай лишається непомітним для користувача.

DNS-сервери можуть відповідати на запити як для свого домену, так і для інших доменів.

Якщо сервер отримує запит від стороннього користувача про адресу сайту, який знаходиться в його зоні відповідальності, він надає точну (авторитетну) відповідь.

Якщо ж запит надходить від користувача всередині його мережі про сайт, що знаходиться за межами його домену, сервер перенаправляє запит далі — зазвичай на DNS-сервер інтернет-провайдера.

1.1.2 Структура DNS

Доменне ім'я знаходиться в URL-адресі. Доменне ім'я складається з кількох елементів. Ієрархія домену слід читати справа наліво, де кожна наступна частина уточнює попередній підрозділ. Приклад домену наведено на рисунку 1.1.

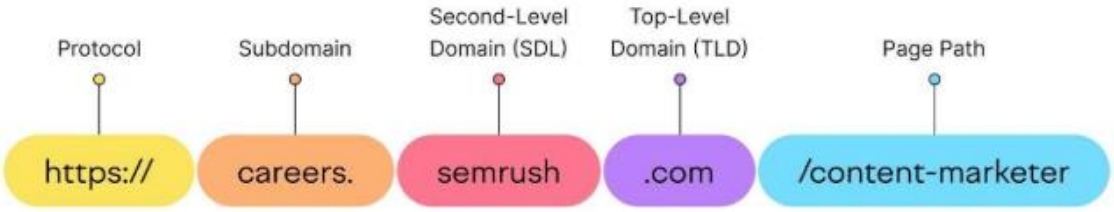

Рисунок 1.1 – Структура домену

Домен верхнього рівня – це частина доменного імені, що розташована після останньої крапки. До прикладу таких доменів можна привести, наприклад, .com, .org і .edu, але є й багато інших. Деякі з них вказують код країни або географічне розташування, наприклад, .us для Сполучених Штатів або .ca для Канади [1].

У кожній URL-адресі частини, що розташовані ліворуч від домену верхнього рівня, є субдоменами доменів, які стоять праворуч. Наприклад, у URL-адресі knu.edu.ua «edu» є субдоменом .ua, а «knu.» - субдоменом edu.ua.

Доменне ім’я може складатися максимум із 127 піддоменів, причому кожна мітка не повинна перевищувати 63 символи. Загальна довжина всього доменного імені може бути до 253 символів. Серед додаткових вимог – заборона використання дефісів на початку або в кінці мітки, а також неможливість повністю цифрового імені домену верхнього рівня (TLD).

1.2 Рендерінг веб-сторінок

Зрозуміли, як браузер перетворює доменні імена на IP-адреси. Тепер потрібно розібратися з чого саме складаються веб-сторінки. У більшості випадків веб-сторінка будується на двух мовах HTML та CSS, але, якщо треба добавити функціональність можна використати JS.

Перед тим, як ідти далі потрібно ознайомитися з трьома поняттями.

DOM (Document Object Model) – це інтерфейс прикладного програмування, який представляє структуру і зміст веб-документа, у формі HTML. DOM представляє елементи веб-сторінки як окремі вузли, що дозволяє

різним мовам програмування взаємодіяти з вмістом сторінки, наприклад JS [2]. Приклад DOM-дерева наведено на рисунку 1.2.

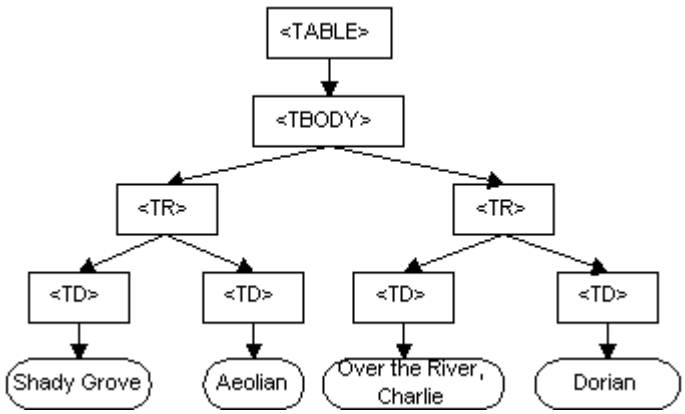

Рисунок 1.2 – Візуальна структура DOM-дерева

CSSOM – це модель об’єктів CSS, яка представляє структуру стилів на веб-сторінці. Вона описує, як браузер обробляє та зберігає CSS-правила, які застосовуються до елементів HTML. CSSOM дозволяє браузеру правильно застосовувати стилі до сторінки і забезпечує коректне відображення елементів відповідно до вказаних стилів. Приклад CSSOM наведено на рисунку 1.3.

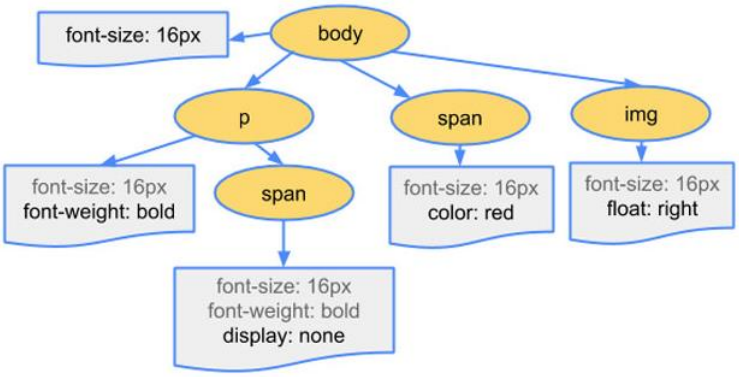

Рисунок 1.3 – Візуальна структура CSSOM

Render tree – це структура, яка створюється браузером після того, як побудовані DOM і CSSOM. Це дерево містить всі елементи на сторінці з урахуванням їхніх стилів і є основою для рендерингу веб-сторінки.

Ознайомившись з основними поняттями, переходимо до більш глибшого розуміння процесу рендеру сторінки. Варто пам’ятати, що комп’ютери можуть обмінюватися лише бітами (0 та 1). Браузер повинен якимось чином перетворити ці біти у формат HTML та CSS.

Спочатку байти перетворюються в символи, приклад наведено на рисунку 1.4.

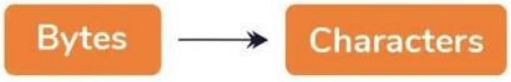

Рисунок 1.4 – Перетворення байтів у біти

Це перетворення виконується на основі кодуванні символів у HTML файлів. Далі символи перетворюються в токени, приклад наведено на рисунку 1.5.

Рисунок 1.5 – Перетворення символів в токени

Під час збереження файлу з розширенням .html браузер інтерпретує його як HTML-документ. Спочатку відбувається процес аналізу. У цьому процесі, зокрема під час токенізації враховуються всі початкові та кінцеві HTML-теги у файлі [3].

Наприклад, токен, який представляє собою тег (anchor) буде мати інші властивості, ніж тег параграфа.

Токен – це своєрідна структура даних, яка містить інформацію про певний HTML-тег. HTML-файл розбивається на невеликі блоки для аналізу, які називаються токенами. На рисунку 1.6 наведено приклад структури токена.

Рисунок 1.6 – Структура токена

Але токен - це не фінальний результат, далі вони перетворюються на вузли. Найкращий спосіб зрозуміти, що таке вузол - це розглядати його як окремий об’єкт у дереві об’єктів документа.

Після створення вузлів вони з’єднуються у спільну деревоподібну структуру даних - DOM.

1.2.1 CSS: каскадна таблиця стилей

CSS – це мова програмування, що побудована на правилах, які визначають стилі для тегів HTML.

CSS має такий самий процес виконання, як і для html. Спочатку отримуємо байти, потім перетворюємо їх у символи, потім у токени, потім у вузли і на кінці отримуємо CSSOM.

Каскадність в CSS – це спеціальний інструмент, який допомагає браузеру вирішити, які стилі використати для елементу. Елементи мають можливість наслідувати стилі від батьківських елементів, саме по цій причині каскадність є важливою. У таблиці 1.1 наведено приклади, як браузер визначає необхідні стилі [4].

Таблиця 1.1 – Визначення специфічності селекторів

Селектор	Стилі в атрибуті style	Ідентифікатор	Селектор класу, псевдокласу або атрибута	Селектор тега	Специфічність
section	0	0	0	1	0.0.0.1
Section h1	0	0	0	2	0.0.0.2
.class					
#someid	0	1	0	0	0.1.0.0
#aside div.show	0	1	1	1	0.1.1.1

1.2.2 Візуалізація структури сторінки

Розібравшись, як браузер будує DOM та CSSOM, варто пам’ятати, що це ще не фінальний результат.

DOM та CSSOM - це дві незалежні структури. DOM зберігає всю інформацію про розміщення HTML-елементів, а CSSOM зберігає інформацію про стилі для кожного елемента. Браузер об’єднує ці два дерева в render tree. Приклад наведено на рисунку 1.7.

Рисунок 1.7 – Дерево рендеру

Дерево рендеру містить інформацію про весь видимий вміст DOM на сторінці та всю необхідну інформацію з CSSOM для різних вузлів. Зауважте, що якщо елемент приховано за допомогою CSS (display: none), вузол не буде відображатися на екрані.

Прихований елемент буде присутній у DOM, але не в дереві візуалізації. Це пояснюється тим, що дерево візуалізації поєднує інформацію як з DOM, так і з CSSOM, тому воно знає, що не можна включати прихований елемент у дерево.

Після створення дерева візуалізації наступним кроком є виконання макета. На цьому етапі вже є інформація про вміст і стилі усього видимого вмісту на екрані, але ще нічого не відображено на екрані.

Браузер повинен обчислити точний розмір і положення кожного об’єкта на сторінці. Це можна порівняти з передачею талановитому математику даних про вміст і стилі всіх елементів, які потрібно відобразити.

Цей етап макета враховує дані про вміст і стиль, отримані з DOM і CSSOM, і виконує всі необхідні обчислення макета.

Після обчислення розмірів і положень елементів залишається останній етап – відрисовка. На цьому етапі вже є інформація про вміст DOM, стилі з CSSOM і точне розташування елементів. Тепер браузер починає відображати елементи на екрані.

1.3 Протокол HTTP

HTTP — це протокол, який забезпечує обмін даними між клієнтом і сервером у мережі Інтернет. Завдяки ньому люди мають можливість відкривати сайти в браузері та взаємодіяти з ними: переходити між сторінками, завантажувати файли, переглядати зображення, спілкуватися й здійснювати оплату покупок [5].

Протокол HTTP працює за принципом клієнт-серверної технології: клієнт надсилає запит на сервер, де спеціальна програма його обробляє, формує відповідь і відправляє назад клієнту.

Зазвичай клієнтом виступає браузер, але його роль можуть виконувати й інші програми, наприклад, пошукові роботи, які переглядають сайти для індексації в пошукових системах. Сервером є веб-сервер — програма, що працює на фізичному сервері та зберігає дані сайту.

Коли клієнт відкриває веб-сторінку, браузер спочатку запитує у сервера HTML-документ із її вмістом. Потім він аналізує цей документ і запитує додаткові файли — стилі, зображення, скрипти, необхідні для коректного відображення сторінки. Після цього браузер об'єднує всі отримані дані та відтворює готову сторінку на екрані.

1.3.1 Методи HTTP-запитів

Це п'ять методів HTTP, які зазвичай асоціюються з RESTful веб-розробкою та протоколом передачі гіпертексту і найчастіше використовуються розробниками RESTful API:

а) GET – призначен для отримання інформації від сервера. Метод вважається безпечною операцією, бо він не повинен змінювати дані;

б) PUT – метод, призначений для оновлення даних на сервері;

в) POST – необхідний метод для надсилання даних на сервер. Не вважається безпечною операцією, оскільки може змінювати дані на сервері та потенійно спричинити збої в роботі програми;
г) DELETE – метод, що видаляє інформацію на сервері. Також є небезпечним тому, що є можливість видалити важливі дані;
ґ) PATCH – застосовується для оновлення окремих полів або частини даних на сервері.

1.3.2 HTTP статус коди

HTTP статус коди розділяють на п’ять різних категорій. Кожна категорія вказує про, що саме була відповідь від сервера. Нижче наведено список усіх кодів [6]:

а) 1xx – інформаціоний;
б) 2xx – успішний;
в) 3xx – перенаправлення;
г) 4xx – клієнтська помилка;
ґ) 5xx – серверна помилка.

Символи «xx » означають різні числа від 00 до 99.

Коди статусу, що починаються з цифри «2», означають успіх. Наприклад, після того, як клієнт запитує веб-сторінку, найчастіше відповіді мають код статусу «200 OK», що означає, що запит було виконано належним чином.

1.4 HTTPS – захищений варіант протоколу HTTP

HTTPS — це безпечне розширення протоколу HTTP, яке шифрує дані при їх передачі. На відміну від звичайного HTTP, де інформація передається у відкритому вигляді, HTTPS використовує шифрування (зазвичай через TLS або SSL). Це допомагає захистити конфіденційні дані від перехоплення та підміни.

Завдяки HTTPS користувачі можуть безпечно вводити паролі, здійснювати онлайн-платежі та передавати іншу важливу інформацію. Використання HTTPS також позитивно впливає на довіру до сайту та покращує його позиції в пошукових системах.

Дізнатися, чи використовує сайт захищений протокол, можна, двічі натиснувши лівою кнопкою миші на URL-адресу, а потім уважно подивитися на протокол. На рисунку 1.8 наведено приклад сайту, який використовує HTTPS протокол.

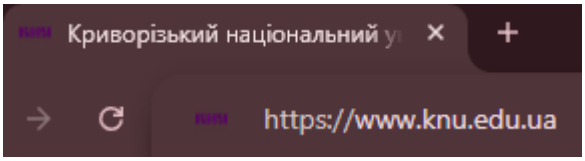

Рисунок 1.8 – Протокол HTTPS

При встановленні HTTPS-з'єднання використовується сеансовий ключ — тимчасовий ключ для шифрування даних під час поточного сеансу користувача. Він генерується кожного разу заново, щоб зловмисники не могли отримати доступ до конфіденційної інформації [7].

Щоб безпечно передати цей ключ, застосовується асиметричне шифрування, де використовуються два ключі:

а) Відкритий ключ (public key) — використовується для зашифрування даних. Він доступний клієнту і серверу;

б) Закритий ключ (private key) — використовується для розшифрування і зберігається лише на сервері.

Спочатку клієнт отримує відкритий ключ сервера, шифрує ним дані (зокрема, сеансовий ключ) і відправляє серверу. Сервер розшифровує повідомлення своїм закритим ключем, отримує сеансовий ключ, і далі сторони використовують симетричне шифрування, що працює швидше.

1.5 Різниця між HTTP та HTTPS

Протокол HTTP використовує порт 80, а HTTPS — порт 443. Головна різниця між ними полягає в тому, що HTTPS забезпечує безпечну передачу даних, тоді як HTTP — ні.

Якщо зловмисник перехопить дані, передані через HTTP, він зможе побачити всю введену на сайті інформацію: логін, пароль, контактні дані або навіть реквізити банківської картки. Саме тому вводити особисту інформацію варто лише на сайтах, що працюють через HTTPS.

Переконатися, який протокол використовується, можна в адресному рядку браузера. Якщо сайт працює через HTTPS, усі дані шифруються, і навіть у разі перехоплення зловмисник отримає лише набір випадкових символів. Для розшифрування потрібен спеціальний ключ, який настільки складний, що навіть найпотужніші комп’ютери не зможуть підібрати його за розумний проміжок часу.

1.6 Ознайомлення з REST API

API – це інтерфейс, який встановлює правила для взаємодії між різними програмними компонентами або системами. Отже, якщо потрібно взаємодіяти з комп’ютером, щоб отримати інформацію або виконати певну функцію, API допомогає передати системі те, що необхідно, щоб вона зрозуміла і виконала запит. Можна вважати API за посередника між користувачами та веб-сервісами.

REST – не є протоколом чи стандартом, це набір архітектурних принципів, яких дотримуються під час проєктування API.

Наприклад клієнт надсилає запит до RESTful API, у відповідь він отримує дані. Ці дані передаються через протоколи HTTP або HTTPS у різних форматах - найчастіше JSON, XML або HTML.

Найчастіше використовують формат JSON, адже він універсальний – незалежний від мови програмування, легко читається як людиною, так і машиною. На рисунку 1.9 наведено приклад того, як виглядає JSON формат.

Рисунок 1.9 – Формат JSON

Основні принципи [8]:

а) Stateless – кожен HTTP-запит виконується в повній ізоляції;

б) Client-Server – автономність клієнта і сервера та їх взаємодія;

в) Cacheable – клієнт зберігає дані, що зменшує навантаження на сервер;

г) Layered System – REST API можуть бути організовані в слої, що робить їх більш гнучкими;

ґ) Uniform Interface –концепція, на якій базується REST API і яка включає чотири ключові аспекти: ресурси, HTTP-методи, формат представлення ресурсів для клієнта та зв'язки між ресурсами. Завдяки цьому забезпечується єдиний, стандартизований підхід до взаємодії між клієнтом і сервером.

Переваги використання:

а) Простота – побудован на простих принципах, таких як HTTP-методи та URL;

б) Універсальність HTTP – використовує HTTP-протокол;

в) Популярні формати даних – використовує JSON, XML та інші;

г) Підтримка кешування – покращує продуктивність застосунку та зменшує навантаження на сервер;

ґ) Масштабування – є можливість додавати нові сервери, без заміни інтерфейса API для клієнтів.

1.7 GraphQL – новий підхід до роботи з API

GraphQL - це мова запитів і середовище виконання API на стороні сервера. Вона надає повну специфікацію для запитів до API, дозволяючи серверам повертати лише необхідні дані без надлишковості. На відміну від RESTful API, запити GraphQL вимагають меншої кількості запитів і пропонують розробникам більше впевненості в результатах, що повертаються.

GraphQL дозволяє розробникам отримувати дані, змінювати їх, а також створювати підписки на оновлення в реальному часі. Такий підхід забезпечує більшу гнучкість у взаємодії з API. Завдяки цьому можна зменшити кількість запитів до сервера та отримувати лише необхідні дані [9].

Основні терміни:

а) Типи об'єктів (Object types): Використовуються для визначення полів об'єктів у моделі даних. Наприклад, об'єкт користувача має такі поля, як ім'я та електронна пошта;

б) Поля (Fields): Визначають типи даних, які сервер GraphQL може отримати при виконанні запиту. Поля зазвичай належать до об'єктних типів, але іноді визначаються як скалярні типи. Скалярні типи включають рядки, цілі числа, числа з плаваючою комою, логічні значення, ідентифікатори тощо;

в) Запит (Query): Визначає точку входу для запитів в API GraphQL. Запити використовуються для отримання даних з сервера GraphQL;

г) Мутація (Mutation): Визначає точку входу для мутацій в API GraphQL. Мутації використовуються для зміни даних на сервері GraphQL;

ґ) Підписка (Subscription): Визначає точку входу для підписки на GraphQL API. Підписки використовуються для отримання оновлень даних в реальному часі з сервера GraphQL.

Компоненти GraphQL:

а) Схеми - модель даних описує структуру об’єктів та зв’язки між ними. Вона визначає, які властивості має кожен об’єкт;

б) Поле - у запитах GraphQL сервер надсилає клієнту лише ті дані, які були чітко вказані у запиті. Це означає, що структура відповіді повністю збігається зі структурою запиту;

в) Параметри - Під час виконання запитів у GraphQL можна передавати параметри, щоб точно вказати умови отримання даних. На рисунку нижче наведено, що користувач.

Завдяки цим компонентам GraphQL забезпечує високу гнучкість та ефективність у роботі з API. До клієнта доходить лише потрібна інформація, що зменшує обсяг трафіку між клієнтом та сервером. Це особливо важливо для мобільних додатків при повільному інтернет з’єднанні.

Переваги GraphQL:

а) Точне отримання даних – клієнти запитують лише необхідні дані, що покращує продуктивність і зменшує витрати на мережу;

б) Гнучкий синтаксис – декларативний підхід дозволяє легко створювати складні запити з урахуванням зв'язків між даними;

в) Без версійності – відсутність необхідності в версіонуванні API, що забезпечує зворотню сумісність;

г) Зменшення запитів – кілька запитів можна об'єднати в один, знижуючи навантаження на мережу;

ґ) Швидка розробка – фронтенд і бекенд можуть працювати незалежно, що пришвидшує ітерації і розвиток продукту.

1.8 Принцип взаємодії через Socket.IO

Socket.IO – інструмент для забезпечення постійного з’єднання між клієнтом і сервером. Він надає можливість ефективно обмінюватися даними в реальному часі [10].

Socket.IO дозволяє двосторонній зв'язок між клієнтом і сервером. Двосторонній зв'язок увімкнений, коли клієнт має Socket.IO в браузері, а сервер також має інтегрований пакет Socket.IO. Хоча дані можна надсилати у різних форматах, JSON є найпопулярнішим.

Для обміну даними та встановлення з’єдання між клієнтом і сервером Socket.IO покладається на бібліотеку Engine.IO, яка працює на нижчому рівні. Вона виконує технічні задачі взаємодії. На сервері використовується Engine.IO, а на клієнті – відповідна бібліотека Engine.IO-client.

Socket.IO працює поверх протоколів HTTP і WebSocket забезпечуючи:

а) Обмін повідомленнями в реальному часі: мінімальна затримка між клієнтом і сервером;

б) Подієву архітектуру: на обох сторонах можна відправляти і приймати події;

в) Автоматичне перепідключення: вбудована логіка для відновлення з’єднання тимчасових розривів;

г) Запасні механізми: у випадку відсутності підтримки WebSocket використовується long polling.

Довге опитування (long polling) – це метод забезпечення реального часу між клієнтом і сервером через HTTP.

Ідея в тому, що клієнт надсилає запит і чекає на відповідь, поки сервер не матиме нових даних. Як тільки з’являється оновлення – сервер відповідає, і клієнт одразу його отримує. При цьому не потрібно постійно підтримувати з’єднання, як у випадку з WebSocket [11].

Довге опитування має низку суттєвих обмежень:

а) Кожен відкритий запит навантажує сервер: навіть у асинхронних середовищах одночасне обслуговування тисяч або мільйонів довготривалих з’єднань збільшує споживання пам’яті та процесорних ресурсів;

б) Тайм-аути та повторні запити створюють додаткове навантаження: при високому навантаженні сервер може не встигати відповідати одночасно, що призводить до хвиль повторних запитів, які ще більше ускладнюють роботу системи;
в) Складно передбачити поведінку інфраструктури: на відміну від протоколів з постійним з'єднанням, long polling не дозволяє ефектинво використовувати з'єднання або об'єднувати кілька потоків, що знижує ефективність системи.

Отже, через ці причини, довге опитування є малопридатним для сучасних реальних застосунків, де важливими є масштабованість, низька затримка та надійність. У таблиці 1.2 наведено порівняння WebSocket з long polling.

Таблиця 1.2 – Порівняння WebSocket з long polling

Характеристика	WebSocket	Long polling
Тип з'єднання	Постійне двостороннє з'єднання	Одностороннє з'єднання
Ініціація	Один раз (після встановлення з'єдання)	Кожен запит вимагає нової ініціації
Протокол	WebSocket (ws://, wss://)	HTTP
Переваги	Менше завантаження, швидший обмін даними	Простота, підтримка через стандартний HTTP
Недоліки	Підтримка через старі браузери, обмеження на порти	Високе навантаження через численні запити

Ознайомившись з таблицею 1.2 робимо висновок, що WebSocket є більш сучаснішим і ефективнішим для сучасним месенджерів та додатків, які працюють у реальному часі. Використання постійного двосторонього з'єднання забезпечує швидку та безперебійну передачу даних з мінімальними затримками. У свою чергу, Long Polling має простішу реалізацію, оскільки використовує HTTP-протокол. Його недоліками є високе навантаження на сервер через велику кількість запитів.

1.9 Refresh та access токени

Коли користувач входить в систему, сервер повертає два токени: refresh та access. Якщо клієнт намагається отримати доступ до захищених ресурсів на сервері, сервер перевіряє за допомогою access токену, чи є у користувача доступ, та чи ще дійсний access токен.

Для створення токенів можна використовувати JSON Web Token. Його структура нагадує звичайний формат JSON, також він дозволяє безпечно обмінюватися даними між клієнтом і сервером.

Вважливо розуміти різницю між access та refresh токенами. Access токен має набагато менший термін дії ніж refresh токен. Це необхідно для того, якщо

зловмисники викрадуть access токен, то в них буде небагато часу на його використання. Refresh токен має більший термін дії та повинен зберігатися лише на сервері, клієнт має отримувати тільки один access токен. Клієнт буде отримувати новий access токен, доки дійсний термін дії refresh токену.

Якщо тривалість refresh токену дорівнює 90 днів, то користувач впродовж цього часу буде отримувати нові access токени. Але коли термін дії refresh токену закінчится, користувачу буде необхідно знову ввійти в систему. На рисунку 1.10 наведено порядок створення токенів.

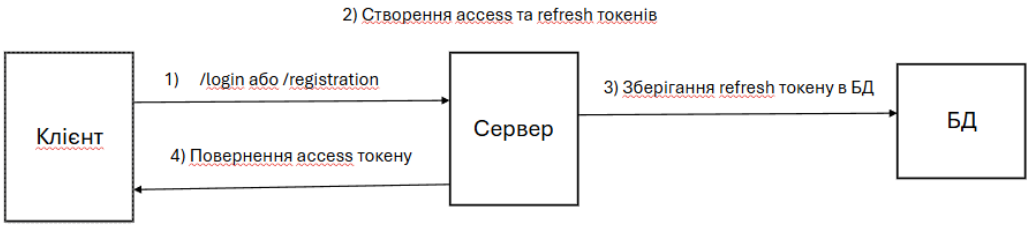

Рисунок 1.10 – Порядок створення токенів

У списку нижче детально описано про кожен крок:

а) Перший крок: користувач заходить у свій акаунт або створює новий. Його дані відправляються на сервер для виконання наступного кроку;

б) Другий крок: сервер на основі отриманої інформації створює access та refresh токени. Обидва токени будуються за допомогою JSON Web Token. Access токен записуємо в cookie;

в) Третій крок: обов'язково зберігаємо refresh токен у БД. Головне пам'ятати правило, що клієнт не повинен отримувати свій refresh токен. Це зменшує ризик витоку даних та забезпечує додаткову безпеку;

г) Четвертий крок: клієнт отримує свій токен доступу, що надає можливість отримувати конфедеційну інформацію. Якщо час дії access токену вичерпався, від клієнта автоматично відправляється запит на оновлення, і сервер перевіряє refresh токен у базі даних.

1.10 Ознайомлення із PostgreSQL та MongoDB

PostgreSQL – це безкоштовна і потужна реляційна система з управління базами даних з відкритим вихідним кодом. Вона сумісна з популярними операційними системами: Linux, UNIX і Windows. Може зберігати дані різних форматів: текст, аудіо, відео, зображення і тд. Також, PostgreSQL підтримує сучасні мови програмування, наприклад: Python, Go, C# та інші [12].

PostgreSQL дотримується ACID властивостей: атомарність, узгодженість, ізольованість, надійність. Ці принципи забезпечують надійне виконання транзакцій навіть при появі збоїв системи. Нижче більш детально написано про кожен із принципів:

а) Атомарність: гарантує, що транзакція виконується повністю або скасовує її;

б) Узгодженість: необхіден для того, щоб залишати базу даних у дійсному стані, тим самим забезпечуючи цілісність даних;

в) Ізоляція: гарантує, що паралельні транзакції не впливають одна на одну;
г) Надійність: забезпечує зберігання даних після виконання транзакції навіть у разі збою системи [13].

Ці принципи необхідні для забезпечення високої якості та надійності даних, що є вирішальним для великих компаній.

MongoDB – це нереляційна база даних, яка використовує документи замість таблиць. Вона є альтернативою для реляційних баз даних. Найчастіше її використовують для зберігання великих обсягів даних. Вважливо зауважити, що MongoDB дозволяє зберігати структуровані та неструктуровані дані. Якщо PostgreSQL для зберігання даних використовує таблиці, то MongoDB застосовує документи – пара ключ-значення [14]. У списку нижче наведено перелік переваг MongoDB:

а) Відсутність схем: дає можливість зберігати будь-який тип даних, що сприяє легкому масштабуванню баз даних;
б) Орієнтована на документи: дані зберігаються у вигляді об’єктів, які в майбутньому легко використовувати в різних мовах програмування. Існує можливість створювати вкладені документи, що дозволяє робити зв’язки між документами, як у реляційних базах даних;
в) Масштабованість: MongoDB має горизонтальну масштабованість, що є особливо важливим критерієм для великих компаній;
г) Балансування навантаження: має можливість для роботи на декількох серверах.

Звичайно, що MongoDB має свої недоліки:

а) Обмеження на запис: запис даних до бази даних обмежений ємністю головного вузла;
б) Узгодженість даних: не існує жорстких зв’язків між колекціями, як у реляційних базах даних, де для цього використовують зовнішні ключі;
в) Безперервність: користувач створює лише один головний вузол, але якщо він вийде з ладу, тоді автоматично інший вузол перетвориться на головоний. Очевидно, що цей перехід не є миттєвим і займає певний час.

У таблиці 1.3 наведено порівняння реляційних та нереляційних таблиць.

Таблиця 1.3 – Порівняння PostgreSQL з MongoDB

Параметр	PostgreSQL	MongoDB
Тип бази	реляційна	Нереляційна
Структура даних	таблиці	Документи
Запити	SQL	JSON-подібні
Зв’язки	Зовнішні ключі	Вкладені документи
Масштабування	Вертикальне	Горизонтальне

Ознайомившись із таблицею, робимо висновок, що вибір між PostgreSQL та MongoDB залежить від потреб проєкту. Для побудови надійної структури, транзакцій та складних зв'язків – краще підходить PostgreSQL. У свою чергу, MongoDB підходить для проєктів, де швидкість є вирішальним критерієм. MongoDB забезпечує простоту і гнучкість в роботі з великими даними, а PostgreSQL надає високу надійність, точність і контроль цілісності даних. У сучасному світі розробники можуть використовувати обидва підходи для досягнення кращого користувацького досвіду.

1.11 Огляд можливостей TypeScript

TypeScript – це статична мова програмування, яка є надбудовою над динамічним JS. Його створила компанія Microsoft у 2012 році. Головною проблемою JS при розробці великих додатків – не розуміння, який тип даних має змінна. Саме для вирішення цієї проблеми був створений TypeScript [15].

Статична типізація – означає, якщо при створені зміної їй було задано тип string, то не можливо буде присвоїти зміній інший тип відміний від string. У свою чергу динамічна типізація дозволяє присвоїти зміній будь-яке значення.

JavaScript має сім примітивних типів даних: number, string, boolean, null, undefined, symbol, bigint та один не примітивний – object. TypeScript надає можливість створювати власні типи даних. Найчастіше використовують interface та type. Цікаво зауважити, що TypeScript додає спеціальні типи даних: any, unknown, void, never, enum. На рисунку 1.11 наведено приклад типізації параметрів функції.

```
4   function add(a: number, b: number): number {
5       return a + b;
6   }
```

Рисунок 1.11 – Типізація функції

Спробуємо передати до функції параметр з не правильним типом даних. Результат наведено на рисунку 1.12.

```
4   func
5
6   }
7
8   add("1", 2);
```

Argument of type 'string' is not assignable to parameter of type 'number'. (2345)

View Problem (Alt+F8) No quick fixes available

Рисунок 1.12 – Повідомлення про помилку

Навіть до запуску програми, компілятор вже повідомляє про помилку в коді. Звичайно, що це простий приклад і він не показує всієї сили TypeScript. Отже, щоб остаточно вирішити різницю між JavaScript та TypeScript ознайомимося з таблицею 1.4.

Таблиця 1.4 – Порівняння JavaScript з TypeScript

Характеристика	JavaScript	TypeScript
Типізація	статична	Динамічна
Типи даних	7 примітивних та object	7 примітивних, object, any, unkown, void, never, enum
Можливість створювати власні типи даних	-	+ (type, interface, enum)
Виявлення помилок під час розробки	Виявляються лише під час виконання	Помилки виявляються під час компіляції
Підтримка ООП	часткова	повноцінна

Проаналізувавши головні відміності між цими технологіями робимо висновок, що TypeScript більше підходить для створення великих проєктів. Великим недоліком JavaScript є динамічна типізація, яка може викликати не розуміння, який тип має зміна. У той час, як TypeScript відобразить помилку, ще до запуску програми. Також, великим плюс при використані TypeScript буде можливість створювати власні типи даних.

1.11.1 Створення власних типів даних

При використані ООП програмісти часто створюють класи, які є абстракцією для реальних або нереальних речей. Для прикладу візьмемо додаток онлайн магазину. Очевидно, що в кожному магазині є товар, який має якісь властивості. За допомогою TypeScript створимо спеційльний тип даних та застосуємо його для класу, на рисунку 1.13 наведено приклад.

```
type Product = {
    title: string;
    description: string;
    price: number | string;
    sku: string;
};

class iPhone implements Product {
    title: string;
    description: string;
    price: number | string;
    sku: string;

    constructor(
        title: string,
        description: string,
        price: number | string,
        sku: string
    ) {
        this.title = title;
        this.description = description;
        this.price = price;
        this.sku = sku;
    }
}
```

Рисунок 1.13 – Створення спеціального типу Product

За допомогою TypeScript було створено тип Product, який визначає структуру товару: назву, опис, ціну та унікальний код. У класі iPhone реалізовано цей тип, використовуючи ключове слово implements. Отже, перевага TypeScript у тому, що він дозволяє заздалегіть задавати чітку структуру даних і перевіряти відповідність цієї структури під час розробки. Це робить код більш надійним.

1.11.2 Відмінності type від interface

Початківці, які тільки почали вивчати TypeScript не розуміють різницю між типами type та interface. На перший погляд вони здаються дуже схожими. Проте між ними є важливі відмінності, які наведено у таблиці 1.5.

Таблиця 1.5 – Порівняння type з interface

Характеристика	Type	Interface
Призначення	Опис об’єкта, кортежів, примітивів	Опис структури об’єкта або класу
Наслідування	За допомогою ключового слова extends та через &	За допомогою ключового слова extends
Декларативне розширення	Неможливо	Можна доповнювати ту саму interface кілька разів
Сумісність із класами	Працює, але краще не використовувати	Ідеально підходить

Отже, interface більше підходить для опису класів і об’єктів. У свою чергу type є більш гнучким. Вибір між type та interface залежить від конкретної задачі.

1.11.3 Декоратори в TypeScript

Декоратор – це синтаксис мета-програмування для класів, властивостей класу та методів. Сучасний TypeScript підтримує декоратори як експериментальну функцію.

Перед кожним декоратором має стояти символ "@". Можна сказати, що декоратор нагадує middleware тим, що перед виконанням методу класу, який має декоратор, спочатку виконається сам декоратор, а вже потім - основна функція. Отже, можна реалізувати додаткову логіку та тримати її в одному місці. Крім того, декоратори збільшують читабельність програми, чітко відокремлюючи основну функцію від допоміжних перевірок.

Для прикладу візьмемо декоратор onlyNumbers, який дозволяє виконання методу лише у випадку, якщо всі аргументи є числами. На рисунку 1.14 зображено приклад декоратору.

```
function onlyNumbers() {
  return function (
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    const originalMethod = descriptor.value;

    descriptor.value = function (...args: any[]) {
      const allNumbers = args.every(arg => typeof arg === "number");

      if (!allNumbers) {
        throw new Error(`Method ${propertyKey} accepts numbers only.`);
      }
      console.log(`onlyNumbers(): validation passed for ${propertyKey}`);
      return originalMethod.apply(this, args);
    };
  };
}

class ExampleClass {
  @onlyNumbers()
  method(a: number, b: number) {
    console.log("Method executed with", a, b);
  }
}

const ex = new ExampleClass();
ex.method(10, 5);
```

Рисунок 1.14 – Декоратор

Використання декораторів робить додаток більш гнучким і зручним для масштабування. Один метод може приймати декілька декораторів одночасно, якщо необхідно. Найчастіше їх використовують для валідації вхідних даних, логування або контролю доступу. Декоратори дозволяють винести повторювану логіку за межі основного коду. Завдяки цьому розробники можуть зосередитися на бізнес-логіці.

1.12 Next.js – фронтенд фреймворк

Фреймворк – це спеціальний інструмент, який має свої правила написання коду.

Next.js – це популярний фронтенд фреймфорк, який побудований на можливостях React. Він має особливу структуру проєкту та організації файлів. Існують зарезервовані імена файлів:

а) Layout (.js, .jsx, .tsx) – використовується для задання шаблону сторінок. Найчастіше на сайтах вверхня частина (header) є однаковою на всіх сторінках. Отже, можна створити один шаблон і використовувати його на декількох сторінках;

б) Page (.js, .jsx, .tsx) – звичайна сторінка на сайті;

в) Loading (.js, .jsx, .tsx) – потрібна для відображеня процесу завантаження;

г) Not-found (.js, .jsx, .tsx) – відображається, коли сторінку не знайдено;

ґ) Error (.js, .jsx, .tsx) – використовується, коли відбулася якась помилка;

д) Global-error (.js, .jsx, .tsx) – спеціальний файл, який задає глобальний шаблон помилки для всіх сторінок. Може бути корисним, коли розробник не бажає створювати для кожної сторінки унікальний дизайн помилки.

Зарезервовані імена файлів, це ще не все, що має Next.js. Він дає можливість: оптимізувати зображення використовуючи спеціальний тег <Image />, зручно завантажувати шрифти, підтримку CSS Modules, Tailwind

CSS, Global CSS та Sass.

Великою перевагою є можливість серверного та клієнтського рендеру компонентів. По замовчуванню Next.js помічяє всі файли, як серверні, якщо нам необхідно зробити їх клієнтськими потрібно в першій строчці коду написати “use client”.

Для рендерингу серверних компонетів Next.js використовує спеціальний формат React Server Component Payload (RSC Payload). Після цього ці дані відправляються на клієнт. React на стороні клієнта отримує потік цих даних і виконує процес гідрації.

У випадку клієнтських компонентів Next.js завантажує JavaScript-модулі, які є необхідними для роботи. Далі браузер завантажує ці скрипти, додаючи до них інтерактивність. Важливо зазначити, що клієнтські компоненти рендеряться у браузері.

Серверні компоненти потрібно використовувати:

а) Отримувати дані з бази даних або API;

б) Використовувати ключи авторизації або токени;

в) Зменшити кількість JavaScript-коду, який відправляється до браузера.

Клієнтські компоненти потрібно використовувати:

а) Використовувати React хуки: useState, useEffect та інші;

б) Обробники подій: onChange, onClick та інші;

в) Доступ до API браузера: localStoreage, window, indexedDB;

г) Створення своїх хуків.

Ознайомившись із базовою інформацією про фреймворк Next.js, зрозуміло, що це дуже простий та швидкий інструмент розробки веб-додатків.

1.12.1 Створення динамічних роутів

Динамічний роут – це маршрут, який приймає спеціальний параметр, на основі якого генерується відповідна динамічна сторінка. Наприклад, на сайті магазину є багато товарів, кожен з яких має свою сторінку з унікальними даними. Звичайно, що ніхто не буде створювати під кожен товар нову сторінку.

У фреймворку Next.js для вирішення такого питання можна створити спеціальну директорію з назвою [slug], де замість slug може бути написано все, що завгодно. За параметром у квадратних дужках, далі будемо діставати значеня динамічного сегменту, приклад наведено на рисунку 1.15.

Рисунок 1.15 – Приклад динамічного маршруту

Виділений прямокутник показує динамічний сегмент URL-адресу. Тепер потрібно розібратися, як саме в коді отримати цей сегмент. Для цього можна написати код, приклад якого наведено на рисунку 1.16.

```
export const DynamicPage = async ({
  params,
}: {
  params: Promise<{ slug: string }>;
}) => {
  const { slug } = await params;
  return <div>My dynamic segment: {slug}</div>;
};
```

Рисунок 1.16 – Код для отримання динамічного сегменту

У прикладі коду сегмент має назву slug, але це не обов'язково - можна задати будь-яке інше ім'я, головне, щоб воно співпадало з назвою директорії. Після отримання сегменту можна створювати запроси до API або серверу.

1.12.2 Ознайомлення з основними хуками React

Розробники бібліотеки React зобов'язують програмістів використовувати спеціальні хуки. Вони потрібні для зберігання даних, зменшення кількості рендерів або створення прямого посилання на елемент у DOM. У таблиці 1.6 наведено детальнішу інформацію про кожен із хуків.

Таблиця 1.6 – Основні хуки в React

Назва хука	Використовується
useState	Зберігає значення між рендерами компонента
useEffect	Виконує асинхрону логіку
useRef	Створює зв'язок зі справжнім DOM деревом
useMemo	Кешує результати обчислень
useCallback	Кешує саму функцію
useContext	Дозволяє отримати інформацію зі створеного контексту (context)

У разі потреби зберігати важливі дані, які потрібно зберегти між рендерами компонента, тоді треба використовувати useState.

Хук useEffect виконується після повного рендерингу компонента. Він є ідеальним місцем для додавання побічних ефектів.

У випадках, коли треба отримати прямий доступ до елементу в DOM-дереві, використовується хук useRef.

Оптимізувати код можна використовуючи хуки useMemo та useCallback. Вони зменшують кількість рендерів компонента.

Останній з основних хуків – це useContext. Він дозволяє отримати значення з контексту, тобто замість передавання пропсів через багато компонентів, просто напряму отримаємо необхідні дані.

1.12.3 Віртуальний DOM

Віртуальний DOM – це точна копія справжнього DOM-дерева, але вона займає менше місця в пам'яті. Завдяки віртуальному DOM бібліотека React може обчислювати мінімальний набір необхідних змін [16].

Коли стан компонента змінюється, React створює нову версію віртуального DOM, потім порівнює її з пепередньою, знаходить відмінності та оновлює лише необхідні частини в справжньому DOM-дереві. Приклад наведено на рисунку 1.17.

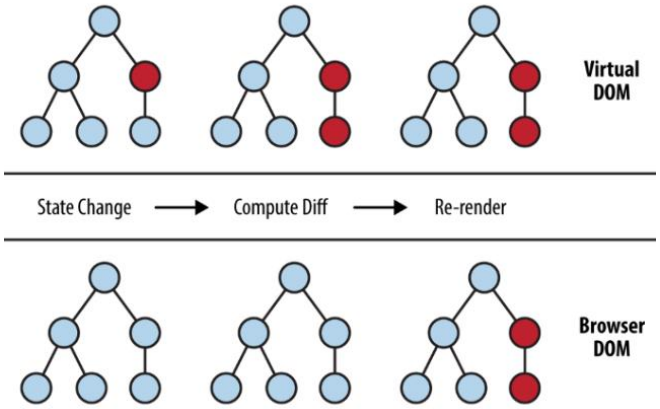

Рисунок 1.17 – Оновлення Virtual Dom

Переваги використання віртуального DOM-дерева:

а) Декларативність – розробники пишуть, який результат має відобразитися замість детальних покрокових інструкцій;

б) Краща продуктивність – маніпуляція з віртуальним DOM набагато швидше ніж із справжнім DOM;

в) Покращений користувацький досвід – оновлення інтерфейсу відбуваються лише в потрібних місцях, замість оновлення всієї сторінки.

Віртуальний DOM є ключовим компонентом у бібліотеці React. Завдяки його перевагам можна створювати швидкі та ефективні інтерфейси. Саме з цих причин React залишається популярних інструментом для створення вебдодатків.

1.13 Nest.js: бекенд фреймворк

Nest.js – це фреймворк для створення серверних додатків на Node.js. В основі він використовує вже відомий TypeScript, що забезпечує статичну типізацію. Nest.js має можливість працювати з REST API, GraphQL або WebSocket. Також, він підтримує інтеграцію: мікросервісів, Redis, Kafka та багато інших.

Основою Nest.js є модульна архітектура, яка включає в себе:

а) Модулі: потрібні для логічного розділення функціональності. Якщо необхідно один модуль може використовувати інший;

б) Контролери: це те місце куди потрапляють усі HTTP-запити, потім їх направляють до відповідних сервісів;

в) Сервіси: містять головну бізнес-логіку додатка.

За допомогою такого підходу дуже легко розуміти, де код повинен знаходитися. Кожна частина виконує своє завдання, що надає краще розуміння проєкту та полегшує масштабування.

Також, Nest.js підтримує два підходи для роботи з базами даних, це використання TypeORM і Mongoose.

TypeORM – це інструмент, який надає можливість працювати з реляційними базами даних: PostgreSQL, MySQL, SQLite та іншими. Він дозволяє використовувати TypeScript для визначення моделей даних і виконання запитів. На рисунку 1.18 наведено приклад моделі створеної з використанням TypeORM.

```
import { Entity, Column, PrimaryGeneratedColumn, createConnection } from 'typeorm';

@Entity()

export class User {

  @PrimaryGeneratedColumn()

  id: number;

  @Column()

  name: string;

  @Column()

  age: number;

}
```

Рисунок 1.18 – Модель TypeORM

Mongoose – це популярна бібліотека, яка надає можливість зручно працювати з базою даних MongoDB. На рисунку 1.19 наведено приклад моделі побудованої на Mongoose.

```
import { Schema, model, connect } from 'mongoose';

const userSchema = new Schema({

  name: String,

  age: Number,

});

const User = model('User', userSchema);
```

Рисунок 1.19 – Модель Mongoose

Додатково Nest.js має стандартні класи:

а) Middleware – це функція, яка виконується перед контролером;

б) Exception filters – потрібен для обробки всіх винятків у програмі. Якщо розробник забув опрацювати помилку, то її перехватить цей фільтр;

в) Pipes – виконує перетворення отриманих даних в інший формат. Наприклад з типу string зробити int;

г) Guards – фільтр, який перевіряє чи є у клієнта доступ до приватного роуту чи ні. Ідеально підходить для написання логіки авторизації.

1.13.1 Способи захисту серверу

Nest.js надає декілька способів захистити сервер: CORS, CSRF Protection, Rate limiting та Helmet. Ці інструменти допомагають запобігти поширеним типам атак та зробити наш API міцнішим і надійнішим. Детальніше про кожен із них у списку нижче:

а) CORS: дозволяє встановити, з якими доменами буде працювати наш API. Також є можливість перелічити методи (GET, POST), які приймає сервер;

б) CSRF Protection: тип хакерської атаки, при якій хакер змушує користувача виконати дію на сайті. При цьому в зловмисника є критичні дані користувача (логін, токен);

в) Rate limiting: дає можливість встановити скільки максимум запитів за певний період часу може виконувати роут. Якщо не бажаєте встановлювати окремо на кожен роут ліміт, можна встановити глобальний ліміт, який діє на всі роути;

г) Helmet: додає спеціальні HTTP-заголовки, які захищають сайт від поширених атак.

Ці способи захисту є обов’язковими під час розробки сучасного онлайн-месенджера. Вони зменшують ризики, пов’язані з спамом, крадіжкою даних та іншими загрозами. Nest.js має всі необхідні API для впровадження кожного з цих механізмів. Чим більше способів захисту буде використано, тим більше рівень захищеності серверу.

1.13.2 Обробка помилок

У фреймворку Nest.js є вбудований клас HttpException. За замовчуванням він має дві властивості:

а) Status code – статус помилки, що відповідає стандарту HTTP;

б) Message – використовується для детальнішого опису помилки.

У більшості випадків цих властивостей достатньо для обробки типових даних. Проте, для складніших випадків можна створювати власні винятки з розширеним набором даних.

У випадку, коли розробник бажає вказати, через що саме виникла помилка, потрібно перезаписати клас HttpException. Можна задати конструктору власне повідомлення, статус-код і додаткову інформацію про причину помилки. Це забезпечує гнучкість і дозволяє повертати більш інформативні відповіді. Приклад наведено на рисунку 1.20.

```
@Get()
async findAll() {
  try {
    await this.service.findAll()
  } catch (error) {
    throw new HttpException({
      status: HttpStatus.FORBIDDEN,
      error: 'This is a custom message',
    }, HttpStatus.FORBIDDEN, {
      cause: error
    });
  }
}
```

Рисунок 1.20 – Перезапис класу HttpException

У цьому випадку конструктор приймає три параметри:

а) Перший параметр – результат, який отримає клієнт;

б) Другий параметр – HTTP-статус код;

в) Третій параметр – потрібен для логування помилки, що є корисним для адміністраторів.

1.14 Паттерн Dependency injection

Ін’єкція залежностей (Dependency Injection) – це патерн архітектури програмного забезпечення, який дозволяє передавати класам необхідні залежності, від яких залежить клас, замість створення їх всередині класу.

Ін’єкція залежностей дотримується принципам SOLID [17]:

а) Принцип єдиної залежності (Single responsobility) – у класа має бути одно причина для зміни;

б) Принцип відкритості / закритості (Open-closed) – класи мають бути відкритими для розширення, але закритими для модифікації;

в) Принцип підстановки Лісков (Liskov substitution) – один об’єкт може замінити інший без змін поведінки програми;

г) Принцип розподілу інтерфейса (Interface segregation) – використання специфічних інтерфейсів для зменшення залежностей;

ґ) Принцип інверсії залежностей (Dependency inversion) – модулі низького рівня не залежать від верхніх модулів, а обидва типи залежать від абстракції.

Цей паттерн широко використовується в сучасних фреймворках. Головною перевагою DI є те, що він відокремлює класи від їх залежностей. Класи залежать від абстракції, яка надає необхідну поведінку.

Основними причинами вибору патерну ін’єкції залежностей є:

а) Послаблення зв’язку: немає тісного зв’язку між класами;

б) Легше тестування: можна прибрати залежності, що спрощує модульне тестування;

в) Покращена читабельність: кожен клас має свою зону відповідальності;

г) Гнучкість: можна змінювати або модифікувати залежності не впливаючи на залежні класи.

1.15 Yarn проти NPM: основні відмінності менеджерів пакетів

У сучасному світі програмування розробники створюють власні бібліотеки або фреймворки. Одна програма може включати багато різних бібліотек, які написані різними людьми. Кожна з бібліотек може оновлюватися, а інші, навпаки, стати застарілими (outdated). Саме по цим причинам і створили менеджери пакетів: npm і yarn. Вони допомагають легко відстежувати всі бібліотеки, які використовуються в програмі. У місяць реєстр npm отримує приблизно до 5 мільярдів завантажен.

NPM – це стандартний менеджер пакетів для Node.js

Yarn – це менеджер пакетів, який розроблений для JavaScript компанією Facebook.

Yarn встановлює пакети паралельно та використовує механізми кешування. З іншого боку, npm встановлює пакети послідовно, що звичайно набагато повільніше. Тому yarn вважають більш швидким та надійним рішенням.

1.16 Система контролю версій Git

Уявімо таку ситуація: розробник працює над проєктом, усе працює добре, але потім він додає нову можливість в додаток - і з'являється баг. Як же розробнику повернутися до того стану програми, коли все працювало? У цьому допоможе Git – система контролю версіями, яка дозволяє повернутися до будь-якого стану програми, немов здійснюючи подорож у часі.

Git – це система контролю версій, яка створена Лінусом Торвальдсом.

Спочатку визначимо головні терміни Git [18]:

а) Репозиторій – місце, де зберігаються проєкт із усіма файлами, історією змін та гілками;

б) Коміт – зберігає поточні зміни у коді;

в) Гілка – окрема лінія розробки, яка дозволяє працювати паралельно;

г) Злиття – об'єднання двох гілок в одну;

ґ) Оновлення Pull – отримати зміни з віддаленого репозиторію;

д) Відправка Push – надсилання змін у віддалений репозиторій.

Завдяки існуванню Git розробникам працюючим у одній команді дуже зручно працювати над проєктом. На рисунку 1.21 наведено приклад гілок у Git.

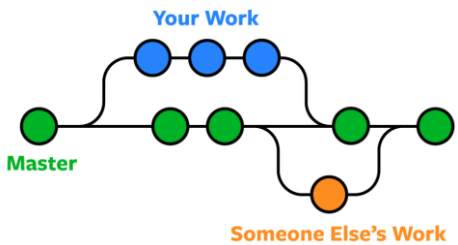

Рисунок 1.21 – Приклад гілок у Git

1.17 Локальний менеджер станів - Zustand

Zustand – це бібліотека управління станами з відкритим вихідним кодом.

Він використовує хуки для побудови API. Увесь стан додатку зберігається в централізованому сховищі. Складовою сховища є зрізи стану, які представляють різні частини програми, що забезпечує модульність [19].

Якщо необхідно оновити стан, потрібно створити новий об’єкт, замість заміни існуючого. До переваг Zustand можно віднести підписки - вони оновлюють компоненти при зміні стану, за яким слідкує підписка.

Для кращого ознайомлення з бібліотекою Zustand створимо сховище, приклад наведено на рисунку 1.22.

```
const useCounter = create((set) => {
  return {
    counter: 0,
    incrCounter: () => set((state) => ({ counter: state.counter + 1 })),
  };
});
```

Рисунок 1.22 – Сховище в Zustand

Зверніть увагу на те, що функція set є функцією зворотнього виклику. Саме функція set використовується для зміни стану в сховищі. Звичайно, що стани можуть мати усі стандартні типи: примітиви та посилальні. У нашому прикладі створено два стани в сховищі: counter та incrCounter.

Counter – змінна, яка зберігає поточне значення лічильника, а incrCounter – функція, яка збільшує значення лічильника на одиницю.

У компоненті для отримання значень із сховища потрібно імпортувати useCounter, приклад наведено на рисунку 1.23.

```
const counter = useCounter((state) => state.counter);
```

Рисунок 1.23 – Отримання даних із сховища

Бачимо, що є можливість обрати, який саме фрагмент потрібно дістати з сховища. На рисунку 1.24 наведено приклад, як дістати функцію incrCounter.

```
const CounterControl = () => {
  const incrCounter = useCounter((state) => state.incrCounter);

  return (
    <div>
      <button onClick={incrCounter}>Incr. Counter</button>
    </div>
  );
};
```

Рисунок 1.24 – Використання функції incrCounter

Бачимо наскільки легко працювати з даними в Zustand. Саме за простоту та невеликий розмір бібліотеки, він є настільки популярний серед розробників.

1.18 Мобільна адаптація

Сучасні веб-сайти повинні мати мобільну адаптацію, оскільки користувачі користуються не тільки комп'ютерами. Згідно з рисунком 1.25, у 2023 році більше 60% користувачів використовують телефони для моніторингу веб-сайтів.

Рисунок 1.25 - Мобільний веб-трафік

Отже, вважливо, щоб сайт добре відображався на комп'ютері та телефоні. У CSS використовується ключове слово media, яке визначає, за якого розміру екрану слід змінювати стилі. Воно може мати вигляд, як показано на рисунку 1.26.

```
@media only screen and (max-width: 600px) {
  body {
    background-color: lightblue;
  }
}
```

Рисунок 1.26 – Ключове слово media

Стилі зміняться, коли ширина екрану буде дорівнювати або менше 600px. На рисунку 1.27 та 1.28 наведено приклади, як один сайт виглядає на різних пристроях.

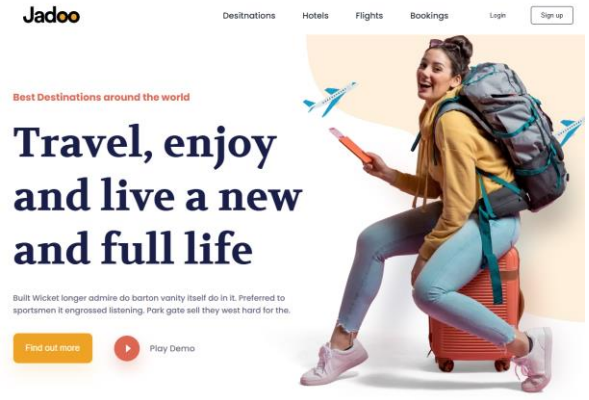

Рисунок 1.27 – Відображення сайту на комп'ютері

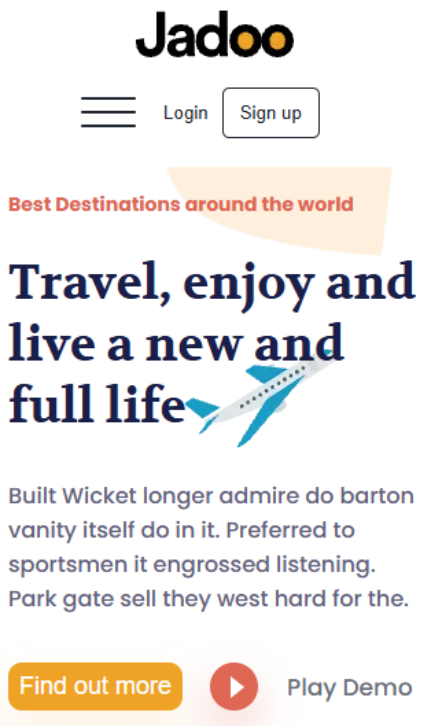

Рисунок 1.28 – Відображення сайту на телефоні

Бачимо, що сайт оптимізован під мобільні пристрої. Помітно, що картинка дівчини зникла на мобільному пристрої. Це не є критично важливим елементом або інформацією на сайті, тому можна не додавати його. Вважило пам’ятати, що сайт має передавати всю необхідну інформацію користувачеві.

Існує спеціальний вебресурс, який показує, наскільки швидко завантажується сайт на різних пристроях, та дає йому оцінку. Висока оцінка свідчить про хорошу оптимізацію ресурсу, що позитивно впливає на користувацький досвід. Якщо ж показники низькі – варто звернути увагу на розмір зображень і порядок завантаження скриптів. Регулярне використання таких інструментів допомагає покращити SEO. На рисунках 1.29 та 1.30 зображено результати перевірки сайту.

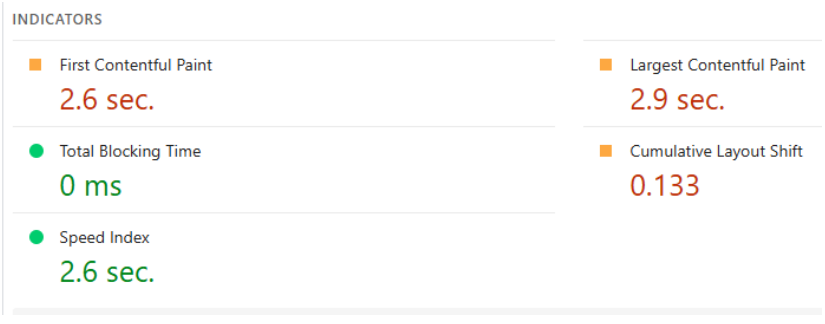

Рисунок 1.29 – Результати для мобільних пристроїв

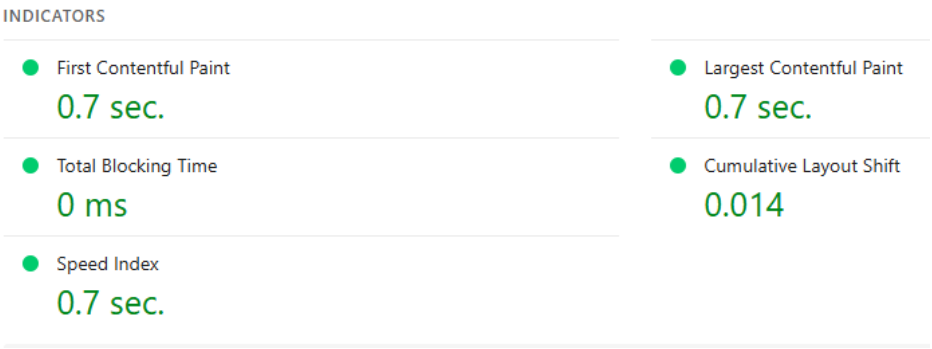

Рисунок 1.30 - Результати для комп’ютеру

Сайт демонструє високі показники, що додає йому додаткових балів перед пошуковими роботами.

Існують два підходи до адаптації сайту для мобільних пристроїв: адаптивний та гнучкий.

Адаптивний дизайн – це фіксовані макети для різних розмірів екрану. Мінімум треба створити три макети сайту для: комп’ютерів, планшетів та телефонів. Це дуже простий та інтуїтивно зрозумілий спосіб, до його переваг належать:

а) Оптимізація під користувача – зробивши макети для кожного можливого пристрою, кожному клієнту буде легко користуватися сайтом;

б) Поліпшення продуктивності – кожен пристрій завантажує лише необхідні йому стилі;

в) Легша підтримка – простіше оновлення макету завдяки тому, що кожен макет є окремою версією сайту.

Гнучкий дизайн – це реактивний дизайн, який реагує на зміну положення елементів на сторінці відповідно до розмірів пристрою. На відміну від адаптивиного дизайну, який використовує багато макетів, гнучкий дизайн створює лише один макет, який підлаштовується під кожного користувача. До його переваг відноситься [20]:

а) Менш затратний по часу – для його створення витрачається набагато менше часу;

б) Кращий для пошукових систем – Google надає перевагу веб-сайтам, зручним для мобільних пристроїв;

в) Краще індексування – пошукові роботи індексують більше вмісту сайту.

Серед цих підходів немає найкращого. Обидва надають можливість переглядати веб-додаток на мобільних пристроях і комп’ютерах, що є критично важливим для сучасних користувачів. Кожен розробник вирішує сам, який підхід обирати в залежності від потреб проєкту.

1.18.1 Lazy Loading: метод оптимізації завантаження зображень

Сайт може мати багато фотографії. За замовчуванням усі зображення, які є на сторінці, завантажуються, коли користувач переходить на певну сторінку. Звичайно, що це потребує часу. Існує спосіб оптимізувати цей процес.

Lazy loading – це метод оптимізації сайтів, який завантажує фотографії тільки тоді, коли вони з'являється на екрані користувача. На рисунку 1.31 зображено ілюстрацію лінивого завантаження.

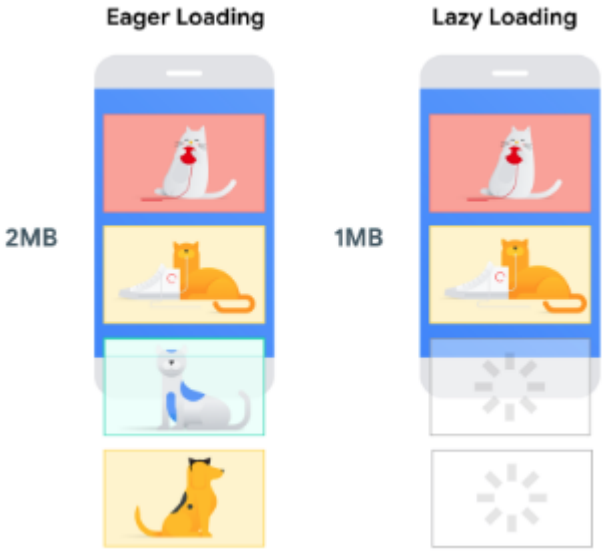

Рисунок 1.31 – Ілюстрація lazy loading

Переваги використання lazy loading:

а) Прискорення завантаження сторінок;

б) Покращення користувацького досвіду;

в) Економія ресурсів.

Отже, цей метод оптимізації допомогає покращити роботу сайту, збільшує час утримання клієнтів на сайті та робить сайт кращим для SEO.

Головне - пам'ятати, що треба робити "лінивими" тільки ті зображення, які знаходяться нижче першого екрана.

1.19 Структура JWT токену

JWT – це популярний стандарт, який забезпечує безпечну передачу інформації.

По суті, токен – це рядок, який містить певну інформацію, яку можна безпечно передавати в мережі. Структура JWT токену містить:

а) Заголовок (Header) – об'єкт, який містить алгоритм та тип токена;

б) Корисне навантаження (Payload) – зберігає JSON-об'єкт;

в) Підпис (Signature) – згенерований криптографічним алгоритмом рядок, який потрібен для перевірки цілісності інформації.

Цей токен часто використовується для аутентифікації користувачів веб-сайтів. Вваживо знати, що JWT не шифрує дані, а лише кодує їх у формат Base64. По цій причині не можна зберігати в Payload конфідеційну інформацію.

На рисунку 1.32 зображено структуру JWT токену.

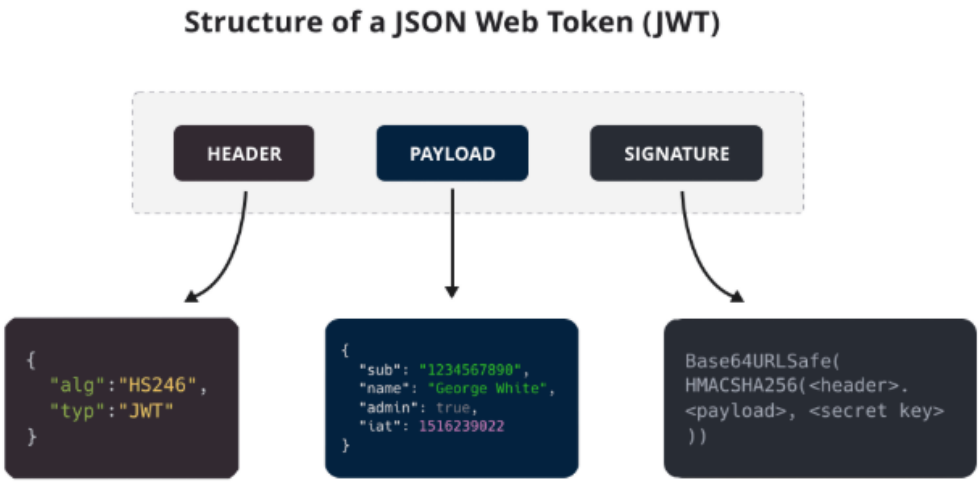

Рисунок 1.32 – Структура JWT токену

JWT-токен має кілька переваг:

а) Аутентифікація без статусу - серверу не потрібно зберігати інформацію про сеанси;

б) Компактність - необхідна інформація знаходиться в токені;

в) Продуктивність - зменшує кількість запитів до сервера.

Завдяки цим властивостям JWT став популярним серед розробників. Але треба пам’ятати про добре спланований підхід до безпеки, особливо щодо зберігання токенів та управлінням строком їх дії.

Отже, JWT-токен забезпечує надійний та зручний механізм авторизації та передачі даних.

1.20 Docker

Docker – це спеціальна програма, яка збирає весь необхідний код додатку, що дозволяє йому працювати в різних середовищах. Розробники можуть з легкістю налаштувати автоматизоване створення артефактів [21].

Основні поняття про Docker:

а) Container – зберігає код та всі необхідні залежності, що дозволяє запускати додаток з різних середовищ;

б) Engine – програма, яка використовується для створення та керування контейнерами;

в) Images – набір інструкцій, необхідних для запуску контейнера;

г) Registry – місце, де розробники можуть зберігати та завантажувати артефакти.

Кожен контейнер виконуються в ізольованому середовищі. Згідно цього виникають основі переваги:

а) Переносимість – контейнеру байдуже, на якій операційній системі від запущений. Вони буквально відокремлені від середовища;

б) Узгодженість – контейнери будуть завжди працювати однаково, незалежно від того, де вони розгорнуті;

в) Швидкість розгортання – контейнер запускається швидко, бо не виділяється час на створення середовища або виділення пам’яті.

Віртуальна машина – це інший спосіб створення ізольованого середовища. Вона не така швидка, як Docker, бо потребує більше часу на створення. Інакше кажучи, віртуальна машина - це окремий "маленький комп’ютер" всередині основного. Створення віртуальної машини відбувається шляхом віртуалізації фізичного обладнання. Кожна віртуальна машина працює на встановленій операційній системі, отже потребує багато місця на жорсткому дисці.

Порівняння віртуальних машин і Docker наведено в таблиці 1.7.

Таблиця 1.7 – Порівняння Docker з віртуальною машиною

Характеристика	Docker	Віртуальні машини
Архітектура	Працює на рівні операційної системи	Емуляція окремої ОС
Швидкість запуску	Швидко	Повільно
Потреба в ресурсах	Мінімальне	Високе
Розмір образів	Невеликий	Великий
Використання	Мікросервіси	Повна емуляція окремої ОС

Робимо висновок, що Docker більше підходить для DevOps розробників, яким потрібна легкість та швидкість налаштування. Контейнери споживають менше ресурсів і швидше запускаються. У свою чергу, віртуальні машини забезпечують повну емуляцію окремої операційної системи, що дозволяє створювати ізольоване середовище для тестування без можливості пошкодити основну систему.

2. ФУНКЦІОНАЛЬНА ЛОГІКА ТА СТРУКТУРА ПРОЄКТУ

2.1 Архітектура FSD

Архітектура FSD – це набір правил для frontend-додатків. Вона розділяє проєкт на шість слоїв:

а) App – головні елементи для всього додатку (маршрутизація, стилі, провайдери);

б) Pages – повні або великі частини сторінки;

в) Widgets – великі незалежні фрагменти функціональності;

г) Features – забезпечують повторне використання бізнес-логіки;

ґ) Entities – сутності з якими працює додаток, наприклад, користувач або товар;

д) Shared – функціональність, яку можна багато разів використовувати.

Слої складаються із зрізів. Можна встановити будь-яку та створювати скільки потрібно. Зрізи потрібні для полегшення навігації у коді.

Існує правило, що зрізи не можуть використовувати інші зрізи на тому ж рівні. Також слої можуть імпортувати функціональність тільки із нижче стояших слоїв, наприклад, слой app може використовувати слой features, але навпаки не можна [22].

Сегменти є складовою частиною зрізів. Є загально прийняті назви для сегментів:

а) Ui – зберігає розмітку елементу;

б) Api – зберігає інтеграцію з API;

в) Model – модель даних: схеми, сховища, інтерфейси;

г) Lib – бібліотечний код, який потрібен іншим модулям на цьому зрізі.

Якщо розробнику потрібно створити власні сегменти, то архітектура FSD не забороняє цього, але для більшості випадків стандартних буде достатньо. На рисунку 2.1 наведено приклад архітектури FSD.

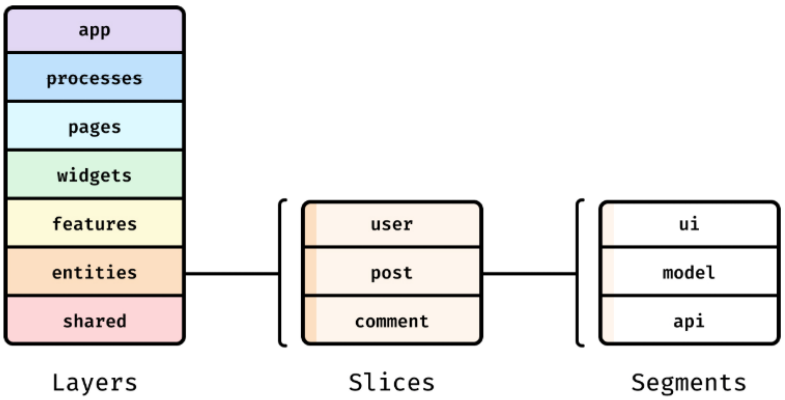

Рисунок 2.1 – Архітектура FSD

Перевагами цієї архітектури є:
а) Стандартирізація – проєкти схожі один на одного, що полегшує командну роботу;
б) Безпечніше змінювати код – модуль на одному рівні не може використовувати модулі на тому ж рівні або на рівнях вище. Отже, безпечніше вносити зміни без виникнення непередбачуваних наслідків;
в) Повторне використання логіки – дотримуємося принципів DRY.

Зробимо висновок, що архітектура FSD надає командам розробників стандартизовану та масштабовану архітектуру. Отже, вона спрощує підтримку проєкту в довгостроковій перспективі. Завдяки модульності зменшується кількість помилок та зростає читабельність коду.

2.2 Логіка реєстрації та авторизації

У будь-якому онлайн-месенджері обов'язково має бути реєстрація користувачів. Вона потрібна, щоб сервер міг відрізнити одного клієнта від інших.

Нехай, модель користувача буде складатися з наступних полів: унікальний ідентифікатор, ім'я, пошта, пароль. Це мінімальний набір необхідних атрибутів для створення таблиці користувачів. Зображення на рисунку 2.2 наводить необхідний код у фреймворку Nest.js, який використовує GraphQL.

```
@Entity({ name: 'users' })
@ObjectType()
export class User {
  @PrimaryGeneratedColumn()
  @Field(() => Int)
  id: number;

  @Column()
  @Field({ nullable: false })
  username: string;

  @Column({ unique: true })
  @Field(() => String, { nullable: false })
  email: string;

  @Column()
  password: string;
```

Рисунок 2.2 – Код для створення таблиці користувачів

Позначаємо атрибут id, як первиний ключ. Очевидно робимо пошту унікальну для кожного користувача, бо не може існувати два клієнта з однаковою поштою.

Після створення моделі користувачів потрібно створити модуль, який буде відповідати за логіку роботи з цією моделю. Нехай назва модулю буде auth. Він має складатися з наступних файлів:
а) Auth.module.ts – зберігає всі моделі з якими працює модуль;
б) Auth.resolver.ts – потрібен для створення кінцевих API;
в) Auth.service.ts – виконує всю бізнес-логіку.

Ознайомимося з логікою реєстрації користувача, на рисунку 2.3 наведено приклад необхідного коду.

```
47  async registration(
48    username: string,
49    email: string,
50    password: string,
51    context: { res: Response }
52  ): Promise<{ access_token: string; userId: number }> {
53    const user = await this.userRepository.findOneBy({ email });
54    if (user) {
55      throw new UnauthorizedException('User with this email already exists');
56    }
57
58    const passHashed = await bcrypt.hash(password, 10);
59
60    const newUser = this.userRepository.create({
61      username,
62      email,
63      password: passHashed
64    });
65
66    await this.userRepository.save(newUser);
67
68    const tokens = await this.generateUserTokens(newUser.id);
69
70    context.res.cookie('refresh_token', tokens.refresh_token, {
71      path: '/',
72      httpOnly: true,
73      secure: process.env.NODE_ENV === 'production',
74      sameSite: 'lax',
75      maxAge: 7 * 24 * 60 * 60 * 1000
76    });
77
78    return {
79      access_token: tokens.access_token,
80      userId: newUser.id
81    };
82  }
```

Рисунок 2.3 – Логіка реєстрації

Використовуючи пошту знаходимо користувача. Якщо нікого не знайдено повертаємо помилку. В іншому випадку потрібно обов'язково перетворити пароль в криптографічну функцію, за для конфенденційності інфомрації. Потім зберегаємо користувача до бази даних. Генеруємо access та refresh токени у рядку 68. На наступному етапі потрібно записати refresh токен до cookie, токен має срок дії 3 дні. До користувача повертаємо access токен та його унікальний айді.

Перевіримо написаний код на практиці. Для цього спробуємо зареєструвати нового користувача, на рисунку 2.4 зображено приклад заповнення форми реєстрації.

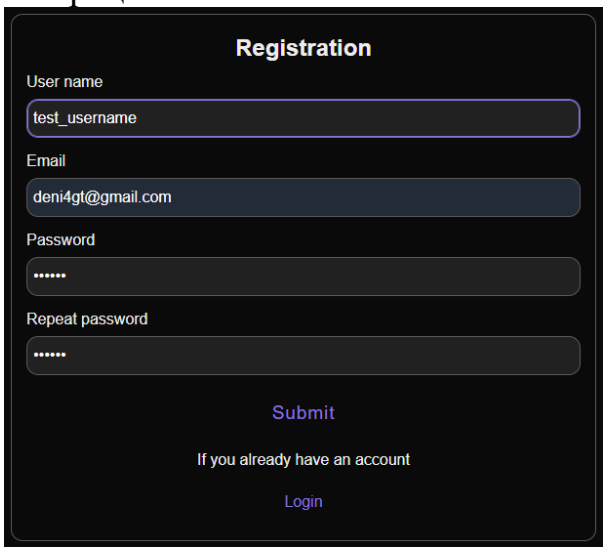

Рисунок 2.4 – Форма реєстрації

Після натискання на кнопку Submit у базі даних має створитися запис у таблиці users, приклад наведено на рисунку 2.5.

	id [PK] integer	username character varying	email character varying	password character varying
1	1	test_username	deni4gt@gmail.com	$2b$10$YeP02bjfjfLPAYhYEFrx/.ZHF98FqMFZXy5kivWjtfSovy7FTuL...

Рисунок 2.5 – Таблиця users

Дані в таблиці збереглися в правильному форматі, отже бізнес-логіка реєстрації користувачів працює, як і очікувалося.

Наступим кроком ознайомимомся з бізнес-логікою авторизації користувачів, на рисунку 2.6 наведено необхідний код.

```
20   async login(
21     email: string,
22     pass: string,
23     context: { res: Response }
24   ): Promise<{ access_token: string; userId: number }> {
25     const user = await this.userRepository.findOneBy({ email });
26
27     if (!user || !(await bcrypt.compare(pass, user.password))) {
28       throw new UnauthorizedException();
29     }
30
31     const tokens = await this.generateUserTokens(user.id);
32
33     context.res.cookie('refresh_token', tokens.refresh_token, {
34       path: '/',
35       httpOnly: true,
36       secure: process.env.NODE_ENV === 'production',
37       sameSite: 'lax',
38       maxAge: 7 * 24 * 60 * 60 * 1000
39     });
40
41     return {
42       access_token: tokens.access_token,
43       userId: user.id
44     };
45   }
```

Рисунок 2.6 – Логіка авторизації

Спочатку знаходимо користувача по пошті, пам’ятаємо, що одна пошта для одного користувача. Якщо нічого не знайдено, тобто користувача з такою поштою не існує повертаємо помилку . У випадку, якщо користувача знайдено генеруємо токени: access та refresh. Знову записуємо refresh токен до cookie. У кінці повертаємо access токен та унікальний айді до користувача.

У додатку відкриваємо форму авторизації та заповнюємо поля використовуючи дані, які записали при реєстрації користувача. Після успішної авторизації access токен буде записан до localStorage. Отже, після заповнення форми перевіряємо localStorage, приклад наведено на рисунку 2.7. Якщо токен присутній, користувач автоматично перенаправляється на головну сторінку застосунку. В іншому випадку користувачу буде показано повідомлення про помилку авторизації.

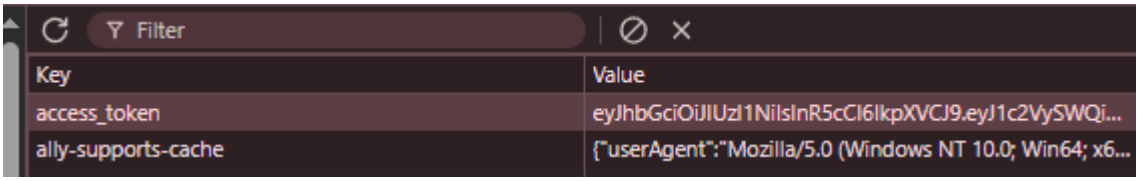

Рисунок 2.7 – Access токен у localStorage

Бачимо, що все спрацювало, як і очікувалося. Тепер додаток має правильно налаштовану логіку реєстрації та авторизації, що дозволить користувачам створювати аккаунти. Реалізація цієї логіки є критичною для створення онлайн месенджера.

Необхідно розібрати логіку виходу з аккаунта. Вона повина видаляти refresh токен з бази даних, також потрібно очистити localStorage. Приклад логіки наведено на рисунку 2.8.

```
17    async logout(userId: number) {
18      const result = await this.refreshTokenRepository.delete({ userId });
19
20      if (result.affected === 0) {
21        throw new Error('Refresh token not found');
22      }
23
24      return { success: true };
25    }
```

Рисунок 2.8 – Логіка виходу з аккаунта

Використовуючи унікальний індентифікатор користувача, видаляється потрібний запис у таблиці refresh_token. На наступному кроці відбувається перевірка, чи було виконано видалення. У випадку, якщо нічого не було видалено, повертається помилка. В іншому випадку клієнт отримує повідомлення про успіше виконання операції.

На стороні клієнта після отримання відповіді видаляється access токен і відкривається сторінка авторизації.

2.3 Створення каналів

Тепер розберемося, як буде працювати логіка створення чатів. Спочатку визначимо, що має зберігати модель каналу та які зв’язки мати з іншими таблицями. Користувач може створювати багато каналів, а канал може мати багато користувачів. Також канал може зберігати багато повідомлень або жодного. У канала може бути декілька посилань на запрошення або жодного. Остання функціональність, яку має мати канал, це можливість зберігати реакції – використовуються, щоб користувач міг залишити під повідомленням свою реакцію. Канал виступає центральною сутністю, яка об’єднує користувачів, повідомлення, запрошення та реакції в єдину логіку. Завдяки цьому можна легко реалізувати функції групового спілкування, контролю доступу та взаємодії між учасниками чату. На рисунку 2.9 наведено приклад ER-діаграми.

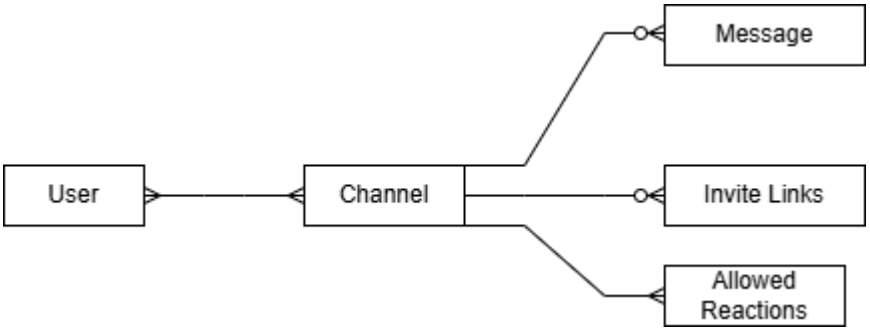

Рисунок 2.9 – ER-діаграма у нотації crow`s foot

Перейдемо до ознайомлення з бізнес-логікою створення каналу. Створимо окрему директорію під назвою channel, яка буде зберігати необхідну логіку пов’язону з каналами.

Директорія channel буде зберігати логіку створення, оновлення, видалення та інше. Уся бізнес-логіка зберігається в потрібному місці, що забезпечує чисту архітектуру та збільшує читабельність коду. На рисунку 2.10 зображено код для створення каналу.

```
async createChannel(
  channelName: string,
  userId: number,
  description?: string,
  avatarUrl?: string
): Promise<ChannelModel> {
  const user = await this.userRepository.findOne({ where: { id: userId } });

  if (!user) {
    throw new Error('User not found');
  }

  const channel = this.channelRepository.create({
    channelName,
    description: description || null,
    amountSubscriber: 1,
    avatarUrl: avatarUrl || null,
    publicLink: '',
    isPrivate: true,
    users: [user]
  });

  await this.channelRepository.save(channel);

  const emojies = await this.emojiRepository.find();

  const channelReactions = emojies.map((emoji) =>
    this.channelReactionRepository.create({
      emoji,
      channel,
      status: false
    })
  );

  await this.channelReactionRepository.save(channelReactions);

  return channel;
}
```

Рисунок 2.10 – Логіка створення каналу

Створити канал має можливість тільки зареєстрований користувач. Тому знаходимо користувача використовуючи його унікальний ідентифікатор. Якщо нікого не знайдено - повертаємо помилку. В іншому випадку створюємо канал та додаємо до нього всі існуючі реакції. Для цього використовується метод map, який дозволяє обійти весь масив. До користувача повертаємо об’єкт channel.

Перевіримо бізнес-логіку на практиці, для цього заповнюємо необхідні дані, приклад зображено на рисунку 2.11.

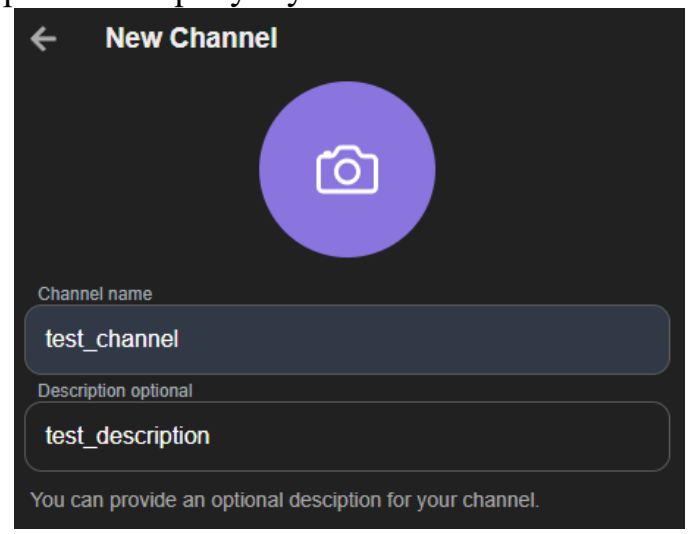

Рисунок 2.11 – Заповнення форми для створення каналу

Перейдемо до pgAdmin та відкриємо таблицю channel, приклад зображено на рисунку 2.12.

	id [PK] integer	channelName character varying	description character varying	amountSubscriber integer	avatarUrl character varying	publicLink character varying	privateLink character varying (16)	isPrivate boolean
1	1	test_channel	test_description	1	[null]		[null]	true

Рисунок 2.12 – Таблиця каналів

Бачимо, що з’явився новий рядок. Отже, логіка створення каналу працює правильно.

2.4 Створення реакцій

У каналу є можливість зберігати реакції. Кожна з них може бути дозволена або заборонена для використання. Пам'ятаємо, що кожен канал зберігає багато реакцій, а кожна реакція може використовуватися різними каналами. Отже, тут зв’язок many-to-many. Визначимо, як у фреймворку Nest.js налаштувати такий зв’язок.

Спочатку створимо файл під назвою channelReaction.model.ts. У ньому реалізуємо структуру таблиці, приклад наведено на рисунку 2.13.

```
@Entity({ name: 'channel_reactions' })
@ObjectType()
export class ChannelReactionModel {
  @PrimaryGeneratedColumn()
  @Field(() => Int)
  id: number;

  @ManyToOne(() => EmojiModel, { nullable: false })
  @Field(() => EmojiModel)
  emoji: EmojiModel;

  @ManyToOne(() => ChannelModel, (channel) => channel.allowedReactions, {
    onDelete: 'CASCADE',
    nullable: false
  })
  @Field(() => ChannelModel)
  channel: ChannelModel;

  @Column({ type: 'boolean' })
  @Field(() => Boolean)
  status: boolean;
}
```

Рисунок 2.13 – Модель channel reactions

Модель має унікальний ідентифікатор, статус, а також зовнішні ключи до EmojiModel та ChannelModel. У самій моделі ChannelModel необхідно створити атрибут з назвою allowedReactions та зв’язати його з таблицею ChannelReactionModel. Виконавши ці кроки Nest.js після запуску програми створить таблицю з назвою channel_reactions, тим самим буде реалізовано зв’язок many-to-many.

Переходимо до ознайомлення з бізнес-логікою реалізації зміни статусу реакції. На рисунку 2.14 зображено приклад необхідного коду.

```
async changeEmojiStatus(channelId: number, emojiId: number) {
  const emoji = await this.channelReactionRepository.findOne({
    where: {
      channel: {
        id: channelId
      },
      emoji: {
        id: emojiId
      }
    }
  });

  if (!emoji) {
    throw new Error('Emoji reaction not found for this channel');
  }

  emoji.status = !emoji.status;

  await this.channelReactionRepository.save(emoji);

  return {
    channelId,
    emojiId,
    status: emoji.status
  };
}
```

Рисунок 2.14 – Логіка зміни статусу реакції

Для пошуку потрібної реакції використовується унікальний ідентифікатор каналу та унікальний ідентифікатор реакції. Якщо такого запису не існує - повертається помилка. У разі існування такого запису його статус змінюється на протилежний, бо цей атрибут має тип даних boolean.

Спробуємо код на практиці, на рисунку 2.15 наведено приклад.

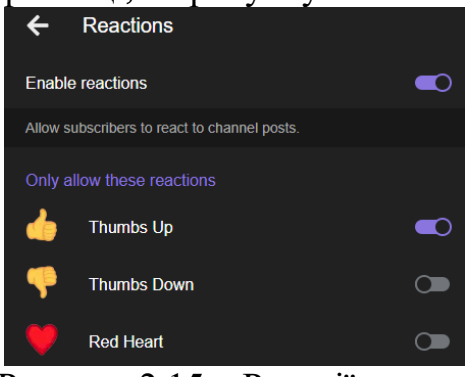

Рисунок 2.15 – Реакції каналу

Змінили статус на реакції Thumbs Up. Перевіримо, чи змінився статус у таблиці channel_reactions. Результат наведено на рисунку 2.16.

	id [PK] integer	status boolean	emojiId integer	channelId integer
1	1	true	1	1
2	2	false	2	1
3	3	false	3	1
4	4	false	4	1
5	5	false	5	1

Рисунок 2.16 – Таблиця channel_reactions

Тепер у каналі реакція Thumbs Up стала активною для використання. Також користувач має можливість одразу змінити статус на всіх реакціях. На рисунку 2.17 наведено приклад коду.

```
296   async changeAllEmojiStatus(channelId: number, status: boolean) {
297     const emojies = await this.channelReactionRepository.find({
298       where: {
299         channel: {
300           id: channelId
301         }
302       }
303     });
304
305     if (!emojies) {
306       throw new Error('No emoji reactions found for this channel');
307     }
308
309     for (const emoji of emojies) {
310       emoji.status = status;
311     }
312
313     await this.channelReactionRepository.save(emojies);
314
315     return {
316       channelId,
317       updatedEmojies: emojies.map((emoji) => ({
318         emojiId: emoji.id,
319         status: emoji.status
320       }))
321     };
322   }
```

Рисунок 2.17 – Логіка зміни статусу одразу для всіх реакцій

Спочатку шукаємо у таблиці channel_reaction потрібний канал за допомогою унікального ідентифікатора. Якщо каналу не існує - повертаємо помилку. В іншому випадку проходимо по всьому масиву emojies та змінюємо статус для кожної реакції. У кінці зберігаємо результат у базі даних та повертаємо необхідні дані клієнту.

2.5 Створення посилань запрошень

Корисно було б, якби канал мав власні посилання для запрошення користувачів. Реалізація цієї логіки є одним із ключових факторів для покращення користувацького досвіду.

Визначимо необхідні дані для створення посилання запрошення. У першу чергу це буде саме посилання, яке має бути унікальним. Також має бути назва посилання, але воно є не обов'язковим. Наступні дані: дата до якої посилання буде працювати, максимальна кількість користувачів, які можуть перейти по цьому посиланю, кількість користувачів, які вже перейшли. Останім атрибутом буде посилання на таблицю ChannelModel, приклад реалізації моделі наведено на рисунку 2.18.

```
@Entity({ name: 'channel_invite_links' })
@ObjectType()
export class ChannelInviteLinks {
  @PrimaryGeneratedColumn()
  @Field(() => Int)
  id: number;

  @Column({ unique: true, length: 16 })
  @Field(() => String)
  link: string;

  @Column({ nullable: true })
  @Field(() => String, { nullable: true })
  name: string;

  @Column({
    type: 'timestamp',
    default: () => 'CURRENT_TIMESTAMP',
    nullable: true
  })
  @Field(() => String, { nullable: true })
  expireDate: string;

  @Column({ nullable: true })
  @Field(() => Int, { nullable: true })
  maxInviteUsers: number;

  @Column()
  @Field(() => Int, { nullable: false })
  numberAttractedUsers: number;

  @ManyToOne(() => ChannelModel, (channel) => channel.inviteLinks)
  @Field(() => ChannelModel)
  channel: ChannelModel;
}
```

Рисунок 2.18 – Модель channel_invite_links

Тепер перейдемо в онлайн-месенджер та спробуємо створити посилання з наступними даними:

а) Name – channel_test_inbite_link;

б) ExpireDate – 1 день;

в) MaxInviteUsers – 100.

На рисунку 2.19 наведено приклад заповнення форми створення посилання.

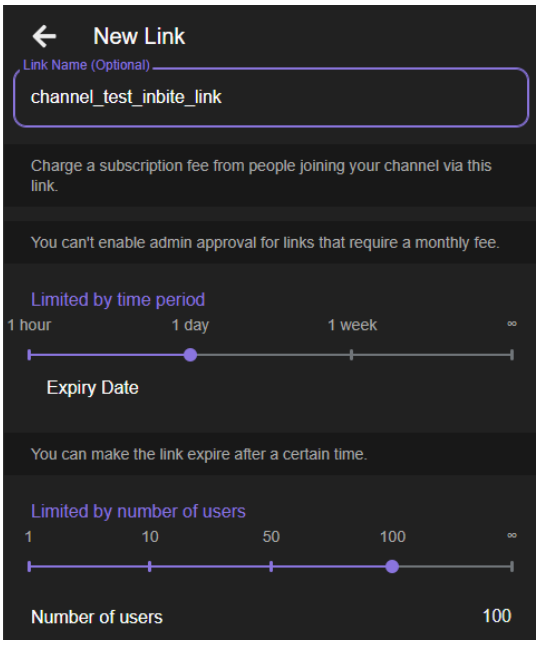

Рисунок 2.19 – Форма створення посилання

Заповнивши форму та натиснувши кнопку відправки потрібно перевірити таблицю channel_invite_links, приклад наведено на рисунку 2.20.

	id [PK] integer	link character varying (16)	name character varying	expireDate timestamp without time zone	maxInviteUsers integer	numberAttractedUsers integer	channelId integer
1	1	fCBL5SV7l8gjhJJ3	channel_test_inbite_link	2025-05-27 14:56:40.451	100	0	1

Рисунок 2.20 – Таблиця channel_invite_links

Усі дані записалися в правильному форматі. Отже, логіка створення посилання працює правильно.

Спробуємо використати створене посилання. Заходимо до застосунку з іншого аккаунта. Переходимо по посиланю запрошення і натискаємо приєднатися до каналу. У таблиці user_channels має додатися новий запис, приклад наведено на рисунку 2.21.

	user_id [PK] integer	channel_id [PK] integer
1	1	1
2	2	1

Рисунок 2.21 – Таблиця user_channels

Таблиця зберегла запис про нового користувача. Логіка додавання користувачів через посилання-запрошення працює коректно. Також таблиця channel_invite_links збільшила значення атрибута numberAttractedUsers на одиницю. Коли кількість запрошених користувачів перевищить максимальне значення, відповідний запис автоматично видаляється.

2.5.1 Відображення всіх посилань запрошення

Дізнавшись, як працює бізнес-логіка створення посилань-запрошень, потрібно розібратися, як відобразити їх всіх на екрані користувача. Відображення списку всіх запрошень покращить користувацький досвід. Також при натисканні на запрошення, воно буде копіюватися до буферу обміну.

Приклад необхідного коду наведено на рисунку 2.22.

```
404   async getAllChannelInviteLink(channelId: number) {
405     const channel_invite_links = await this.channelInviteLinksRepository.find({
406       where: { channel: { id: channelId } }
407     });
408
409     if (!channel_invite_links) {
410       throw new Error('Channel does not have any invite links');
411     }
412
413     return channel_invite_links;
414   }
```

Рисунок 2.22 – Пошук усіх посилань запрошень

За допомогою унікального ідентифікатора каналу отримуємо всі записи з таблиці channel_invite_links. Якщо нічого не має - повертається помилка. В іншому випадку клієнт отримує небхідну інформацію. Приклад наведено на рисунку 2.23.

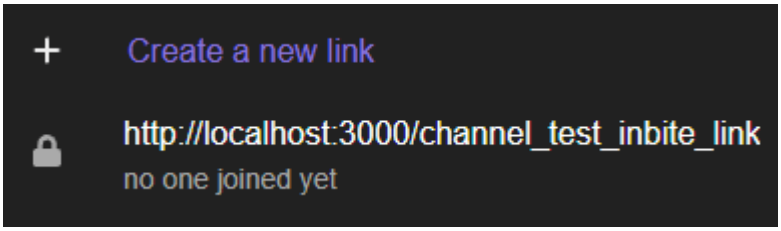

Рисунок 2.23 – Список посилань запрошень

Список відображається, отже бізнес-логіка налаштована правильно. Тепер користувачі можуть бачити всі створені на каналі посилання запрошення.

2.6 Статуси каналу: публічний або приватний

Канал може бути у публічному доступі або приватному. Така функціональність необхідна, якщо адмін каналу бажає зробити канал доступним тільки для певних осіб. Якщо потрібно зробити канал публічним потрібно написати тег, який має бути унікальним. Тег дозволяє користувачам легко знаходити публічний канал. У випадку приватного каналу тег не обов'язковий, бо доступ надається напряму через посилання.

Розібралися з різницею між статусами каналу. Переходимо до ознайомлення з кодом, приклад бізнес-логіки наведено на рисунку 2.24.

```
218   async savePublicLink(channelId: number, publicLink: string) {
219     const existingPublicLink = await this.channelRepository.findOne({
220       where: { publicLink }
221     });
222
223     if (existingPublicLink) {
224       throw new Error('This public link is already in use');
225     }
226
227     const channel = await this.channelRepository.findOne({
228       where: { id: channelId }
229     });
230
231     if (!channel) {
232       throw new Error('Channel not found');
233     }
234
235     channel.publicLink = publicLink;
236     channel.isPrivate = false;
237
238     await this.channelRepository.save(channel);
239
240     return {
241       publicLink
242     };
243   }
```

Рисунок 2.24 – Логіка збереження каналу в публічному доступі

Спочатку виконується пошук по публічному посиланю. Якщо не буде знайдено запис - повертаємо помилку. В іншому випадку знаходимо потрібний канал, використовуючи для знаходження унікального ідентифікатора. Якщо канал не знайдено – також повертаємо помилку. Знайшовши канал, змінюємо значення атрибутів: publicLink та isPrivate. У кінці зберігаємо все до бази даних і повертаємо публічне посилання до клієнту.

Перевіримо на практиці, для цього спробуємо задати тег каналу art_nova та натискаємо зберегти. Відкриваємо таблицю user_channels та перевіряємо, чи змінилися дані. Приклад наведено на рисунку 2.25.

	id [PK] integer	channelName character varying	description character varying	amountSubscriber integer	avatarUrl character varying	publicLink character varying	privateLink character varying (16)	isPrivate boolean
1	1	test_channel	test_description	1	[null]	art_nova	[null]	false

Рисунок 2.25 – Таблиця user_channels

Атрибут publickLink та isPrivate змінилися, інші залишилися не зміненими. Отже, логіка працює правильно та змінює лише необхідні атрибути, що забезпечує цілістність даних.

Переходимо в браузер та спробуємо відкрити канал за публічним посиланням, приклад наведено на рисунку 2.26.

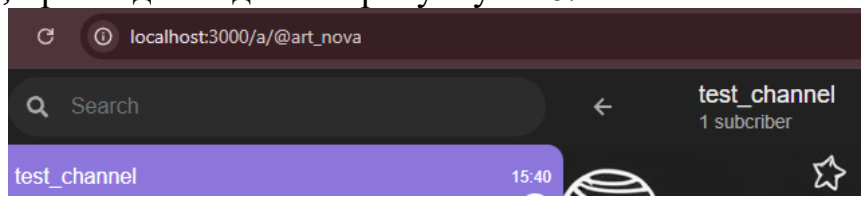

Рисунок 2.26 – Відкриття каналу за публічним посиланням

Відкрився потрібний канал, це означає, що публічне посиланням працює вірно.

Публічне посилання проходить перевірку перед тим, як дозволити користувачу зберегти його. Перевірка потрібна, щоб у базі дних не було два однакових публічних посилання. У випадку невірного посилання користувач побачить візуальну зміну поля форми. Приклад наведено на рисунку 2.27.

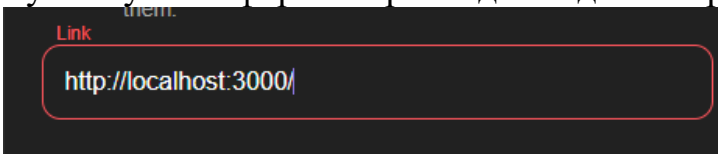

Рисунок 2.27 – Візуальне відображення помилки

Тепер потрібно перевірити зворотню логіку. Спочатку визначимо, що код зміни каналу на приватний майже нічим не відрізняється від публічного, тому наводити рисунок не має сенсу.

У відповідній формі натискаємо на приватний статус каналу та зберігаємо результат. Після цього канал має бути в приватному доступі. Відкриваємо браузер та переходимо до каналу, приклад наведено на рисунку 2.28.

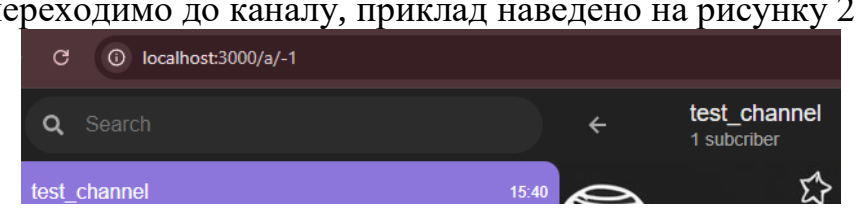

Рисунок 2.28 – Відкриття каналу за приватним посиланням

Бачимо, що динамічний параметр дорівнює -1, це означає, що канал в приватному доступі та доступний лише тим користувачам, які вже в ньому знаходяться.

Приватне посилання може мати наступний вигляд +GknucpiH9fg_14KS . Завжди на початку стоит знак +, потім йде 16 рандомних символів (). На рисунку 2.29 наведено приклад функції генерації приватного посилання.

```
private createHashedLink(channelId: number) {
  const currentTime = Date.now();

  const rawString = `${channelId}${currentTime}`;

  const hash = crypto
    .createHash('sha256')
    .update(rawString)
    .digest('base64url');

  const safeHash = hash
    .replace(/\//g, '_')
    .replace(/\+/g, '-')
    .replace(/=+$/, '');

  return safeHash.slice(0, 16);
}
```

Рисунок 2.29 – Генерація приватного повідомлення

Спочатку беремо поточний час. Потім задаємо змінній rawString присвоюється значення, яке складається з індентифікатора каналу та поточного часу. Використовуємо алгоритм хешування sha256 для створення хеш-рядка у форматі base64url. З отриманого рядка видаляються небезпечні символи: /, +, = для забезпечення безпечного використанння в URL. У кінці відрізаємо перші 16 символів результату та повертаємо їх.

2.7 Створення та призначення DTO

DTO – визначає типи даних та їх обмеження, яких мають дотримуватися параметри функції. Вони можуть визначати і типи вихідних даних.

Наприклад візьмемо вже знайому функцію авторизації. Логічно, що вона приймає пошту та пароль, тип кожного із цих параметрів є string та вони є обов’язковими, приклад наведено на рисунку 2.30.

```
19  @InputType()
20  export class LoginInputDto {
21    @Field(() => String, { nullable: false })
22    email: string;
23
24    @Field(() => String, { nullable: false })
25    password: string;
26  }
```

Рисунок 2.30 – DTO для вхідних параметрів функції авторизації

У таблиці 2.1 наведено порівняння всіх необов’язкових параметрів.

Таблиця 2.1 – Необов'язкові параметри

Параметр	Тип	Призначення
Nullable	boolean	Дозволяє схемі GraphQL повернути null
DefaultValue	any	Значення за замовчуванням

Продовження таблиці 2.1

Параметр	Тип	Призначення

Name	string	Задає ім’я параметра
Description	string	Опис параметра в GraphQL-документації
DeprecationReason	string	Позначає параметр застарілим

Кожен з параметрів використовується для детального налаштування параметра в GraphQL-схемі, що покращує гнучкість і зручність розробки.

Тепер потрібно записати DTO для даних, які повертає функція. Пам’ятаємо, що функція авторизації повертає access токен та унікальний індентифікатор. Приклад наведено на рисунку 2.31.

```
@ObjectType()
export class LoginResponse {
  @Field(() => String)
  access_token: string;

  @Field(() => Int)
  userId: number;
}
```

Рисунок 2.31 – DTO для результату функції авторизації

GraphQL надає можливість зручно перевірити всі створені моделі, функцій мутацій та запити. Для цього відкриваємо браузер, пишемо URL, на якому працює сервер, та додаємо /graphql. Відкривається зручний інтерфейс, який дозволяє швидко перевірити створені функції.

Знайдемо функцію авторизації для перевірки її DTO. Приклад наведено на рисунку 2.32.

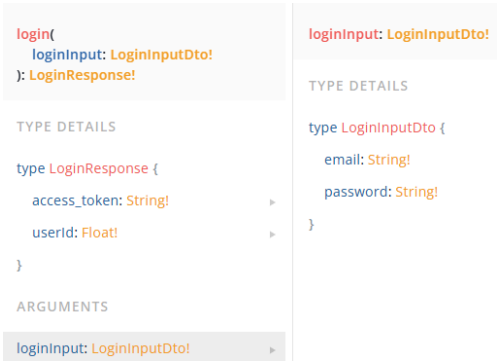

Рисунок 2.32 – Функція авторизації у додатку від GraphQL

Бачимо, що всі значення відповідають тим, які були записані в коді. Якщо дані не відображаються, потрібно перезапустити сервер та знову зайти в додаток від GraphQL, бо сервер може не мати автоперезавантаження при зміні коду.

2.8 Завантаження усіх можливих емодзі

Онлайн месенджер має всього 73 стандартних емодзі. Тому їх кількість не може збільшитися або зменшитися. Фреймворк Nest.js дозволяє автоматично завантажити усі емодзі після запуску сервера. Отже, у головному

модулі app потрібно перейти в файл app.service.ts та створити метод onModuleInit. У цьому методі викликаємо необхідну функцію для завантаження усіх емодзі. На рисунку 2.33 наведено приклад loadEmojis функції.

```
async loadEmojis() {
  try {
    const existingEmojis = await this.emojiRepository.find({
      select: ['text']
    });
    const existingUnicodeCodes = new Set(
      existingEmojis.map((emoji) => emoji.text)
    );

    const emojisToAdd = defaultEmojis.filter(
      (emoji) => !existingUnicodeCodes.has(emoji.text)
    );

    if (emojisToAdd.length > 0) {
      const newEmojis = emojisToAdd.map((emoji) => {
        return this.emojiRepository.create({
          text: emoji.text,
          image: emoji.image
        });
      });
      await this.emojiRepository.save(newEmojis);
    }
  } catch (error) {
    console.log(error);
  }
 }
}
```

Рисунок 2.33 – Функція loadEmojis

Спочатку дістаємо усі існуючі записи в таблиці emoji. Далі потрібно визначити, які емодзі необхідно додати, якщо довжина масиву emojisToAdd буде більше за нуль, тоді додаємо необхідні емодзі. У іншому випадку не робимо нічого.

Таблиця emoji має атрибут image, який відповідає за те, яким чином буде відображатися емодзі. Можно зберігати у вигляді UNICODE символів, щось типу U+1F600, або просте посилання на фотографію. У додатку використовується підхід через посилання на фотографію.

У Node.js є можливість створити директорію та зробити її статичною. Тобто в неї можна покласти фотографії та потім до них звернутися до символу. Пригадаймо, що фреймворк Nest.js є надбудовою над Node.js, отже, також може зберігати статичні файли. Треба створити директорію з будь-яким ім’ям, далі потрібно вказати кореневий шлях – складається з __dirname та шляху до створеної директорої відносно поточного шляху. Виконавши налаштування можна запускати сервер та відкрити в браузері фото, яке знаходиться в статичній директорії.

2.9 Логіка відправки повідомлень

Будь-який онлайн месенджер обов’язково має мати функціонал відправки повідомлень. Найчастіше обирають протокол WebSockets. Він є оптимізований для великої кількості клієнтів. Також, WebSocket забезпечує те, що користувач отримає нові повідомлення майже миттєво. Його можна використовувати реалізації різного функціоналу: статус онлайн та індикатору набору тексту. Ідеально підходить для систем із високою взаємодією між

користувачами. Додатковою перевагою WebSocket є зниження навантаження серверу.

У фреймворку Nest.js можна легко інтегрувати бібліотеку GraphQL WebSocket. За замовчуванням ця можливість вимкнена в налаштуваннях GraphQL, щоб її увімкнути потрібно записати значення параметру graphql-ws рівному true в об’єкті subscriptions, приклад наведено на рисунку 2.34.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  context: ({ req, res }) => ({ req, res }),
  driver: ApolloDriver,
  autoSchemaFile: 'src/schema.gql',
  subscriptions: {
    'graphql-ws': true
  }
}),
```

Рисунок 2.34 – Вклюбчение WebSockets

Параметр autoSchemaFile вказує, де буде зберігатиметься файл schema.gql. Він зберігає в собі всі створені моделі, DTO-типи та багато іншого GraphQL коду.

Тепер визначимо, як саме буде працювати логіка запису повідомлення до бази даних. Приклад наведено рисунку 2.35.

```
23  async createMessage(
24    senderId: number,
25    content: string,
26    channelId: number,
27    type: MessageType,
28    reactionCount: any[] = []
29  ): Promise<MessageModel> {
30    const sender = await this.userRepository.findOne({
31      where: { id: senderId }
32    });
33    const channel = await this.channelRepository.findOne({
34      where: { id: channelId }
35    });
36
37    if (!sender || !channel) {
38      throw new Error('Sender or Channel not found');
39    }
40
41    const newMessage = this.messageRepository.create({
42      content,
43      sender,
44      channelId: channel,
45      type,
46      createdAt: new Date().toISOString(),
47      reactionCount
48    });
49
50    return this.messageRepository.save(newMessage);
51  }
```

Рисунок 2.35 – Код для створення повідомлення

Спочатку йде пошук користувача та каналу використовуючі відповідні унікальні індентифікатори. Якщо користувача або каналу не існує повертається помилка. Інакше зберігаємо повідомлення до бази даних та повертаємо його до клієнта.

Запускаємо додаток та пробуємо відправити тестове повідомлення. Воно має зберегтися в таблиці messages, також на клієнті має оновитися стрічка, щоб користувач побачив своє повідомлення. На рисунку 2.36 наведено приклад таблиці messages, після відправки повідомлення.

	id [PK] integer	content character varying	type messages_type_enum	createdAt timestamp without time zone	senderId integer	channelIdId integer
1	1	message	channel	2025-05-26 21:03:22.74	1	1

Рисунок 2.36 – Таблиця messages

Новий запис у таблиці з’явився, значить, що логіка працює правильно. Для реалізації функціоналу, коли створюється повідомлення потрібно викликати функцію, яка додає нові дані на клієнті. У Nest.js існує спосіб підписатися на будь-яку функцію. Таким чином, коли функція успішно виконується, вона буде викликати відповідну підписку. Після цього відбувається автоматичне оновлення інтерфейсу. Приклад коду наведено на рисунку 2.37.

```
52    const { data: addedMessage } = useSubscription<
53      MessageAddedSubscriptionData,
54      SubscriptionVariables
55    >(MESSAGE_ADDED_SUBSCRIPTION, {
56      variables: { channelId: Number(channelId) },
57    });
```

Рисунок 2.37 – Підписка на клыэнты

2.10 Метод оптимізації Debounce

Поганим кодом є коли користувач пише текст в поле input та після кожного написаного символу відразу відправляється запит до серверу. Правильно на такі поля ставити спеціальну функцію Debounce. Вона приймає два значення: текст та час в секундах. Логіка її така, що коли змінюється текст потрібно дочекатися закінчення учього часу і тільки потім буде відправлятися запит до серверу. Використовуючи такий підхід можна оптимізувати роботу серверу. На рисунку 2.38 зображено приклад debounce функції.

```
3  export const useDebounce = <T>(value: T, delay: number): T => {
4    const [debounceValue, setDebounceValue] = useState(value);
5    useEffect(() => {
6      const timerId = setTimeout(() => {
7        setDebounceValue(value);
8      }, delay);
9
10     return () => {
11       clearTimeout(timerId);
12     };
13   }, [value, delay]);
14
15   return debounceValue;
16 };
```

Рисунок 2.38 – Функція debounce

Створюємо стан використовуючи useState, він потрібен для зберігання поточного значення. У середині useEffect буде створюватися новий таймер, кожного разу, коли значення value або delay змінюються. Якщо час вийшов та значення value не змінилося, тоді повертається поточне значення.

2.11 Структура GraphQL моделі

Моделі GraphQL визначають, яку структуру будуть мати таблиці в базі даних. Вони дозволяють встановити обмеження, задати умову та тип даних, який буде мати атрибут.

Використовуючи моделі GraphQL можна реалізувати такі типи зв'язків між сутностями:

а) one-to-one (один до одного);

б) one-to-many (один до багатьох);

в) Many-to-many (багато до багатьох).

Отже, необхідно розуміти, як правильно будувати моделі, для створення надійного та масштабованого коду.

Розглянемо модель GraphQL на прикладі сутності emoji, приклад наведено на рисунку 2.39.

```
4   @Entity({ name: 'emoji' })
5   @ObjectType()
6   export class EmojiModel {
7     @PrimaryGeneratedColumn()
8     @Field(() => Int)
9     id: number;
10
11    @Column({ nullable: false })
12    @Field(() => String, { nullable: false })
13    text: string;
14
15    @Column({ nullable: false })
16    @Field(() => String, { nullable: false })
17    image: string;
18  }
```

Рисунок 2.39 – Модель emoji

Додавання потрібної логіки виконується завдяки декораторам. Спершу в декоратор @Entity передаємо ім'я, яке визначає назву вихідної таблиці в базі даних. Наступний декоратор @ObjectType необхідний для задання типу, який буде використовуватися у схемі GraphQL для відповіді на запити.

У кожній реляційній таблиці має бути первинний ключ. Задати його можна використовуючи декоратор @PrimaryGeneratedColumn. Створити додаткові стовпці в таблиці можна встановивши декоратор @Column. Задаючи тип стовпця використовується декоратор @Field.

Створивши модель GraphQL та розібравшись з основними декораторами потрібно підключити її до головного модуля, щоб фреймворк Nest.js зміг її побачити. Приклад коду наведено на рисунку 2.40.

```
TypeOrmModule.forRoot({
  type: 'postgres',
  host: 'localhost',
  port: 5432,
  username: 'postgres',
  password: 'admin',
  database: 'TG_CLONE_DB',
  entities: [
    User,
    RefreshTokenModel,
    ChannelModel,
    MessageModel,
    MessageReactionsModel,
    ChannelReactionModel,
    ReactionCountModel,
    EmojiModel,
    ChannelInviteLinks
  ],
  synchronize: true
}),
```

Рисунок 2.40 – Налашутвання

Серверу необхідно вказати конфенденційні дані для підключення до бази даних (пароль, тип бази даних, порт).

Параметр synchronize відповідає за автоматичне оновлення структур таблиць в базі даних при кожному запуску програми.

Запустивши сервер у базі даних мають створитися усі необхідні таблиці, приклад наведено на рисунку 2.41.

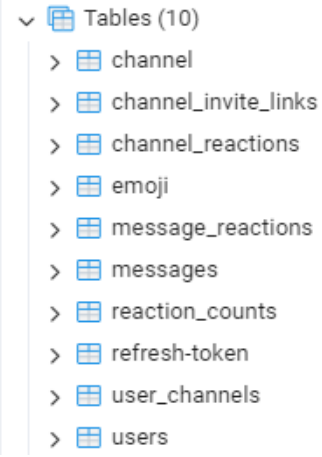

Рисунок 2.41 – Необхідні таблиці в базі даних

Таблиці створилися, помилок після запуску серверу не виникло, отже все відпрацювало правильно.

Розібравшись, як будувати моделі на сервері потрібно навчитися створювати запити на стороні клієнту. GraphQL має три види запитів: mutation, query, subscription. Навчатися будемо на прикладі мутації, яка відповідає за реєстрацію користувача, приклад наведено на рисунку 2.42.

```
export const REGISTER = gql(`
  mutation Register($username: String!, $email: String!, $password: String!) {
    register(
      registerInput: { username: $username, email: $email, password: $password }
    ) {
      access_token
      userId
    }
  }
`);
```

Рисунок 2.42 –Мутація для реєстрації користувача

Спочатку записується тип запиту - ключове слово mutation. Далі записуємо назву мутації (у цьому випадку Register), яка повинна точно відповідати назві на сервері. Наступним кроком визначаються типи параметрів.

На рядках 17 та 18 записано поля, які необхідно витягти з сервера. Пам’ятаємо, що в GraphQL можна повернути з сервера тільки потрібні дані.

Вважливо приділяти особливу увагу до узгодженості між frontend і backend частинами додатку. Можна встановити бібліотеку codegen, яка допомогає розробникам впевнитися, що в мутаціях на стороні клієнта немає помилок.

2.12 Використання Zustand на практиці

Якщо додаток має багато даних, які використовуються різними компонентами потрібно використовувати центральне сховище. Популярним рішенням є бібліотека Zustand. Вона швидко налаштовується та є простою для розуміння, навіть для новачків.

У Zustand для створення сховища потрібно написати спеціальний хук, приклад наведено на рисунку 2.43.

```
export const channelStore = create<ChannelState>((set) => ({
  id: 0,
  channelName: "",
  description: "",
  amountSubscriber: 0,
  avatarUrl: "",
  isPrivate: false,
  publicLink: "",
  privateLink: "",
  allowedReactions: [
    {
      id: 0,
      emoji: {
        id: 0,
        text: "",
        image: "",
      },
      channel: {
        id: 0,
      },
      status: false,
    },
  ],
  setChannelData: (data: Partial<ChannelState>) =>
    set((state) => ({
      ...state,
      ...data,
    })),
}));
```

Рисунок 2.43 – Хук для збереження даних поточного каналу

Ключове слово create використовується для створення сховища. Усередині хука задаються стандартні значення стану. Також можна створити функцію для зміни даних у сховищі - у цьому прикладі це функція setChannelData.

Ця функція приймає частковий об'єкт типу ChannelState і за допомогою оператора розширення (…) оновлює стан сховища, зберігаючи попередні дані. Використовуючи такий підхід можна змінювати лише необхідні поля без перезапису всього стану.

Отримати необхідні дані в компоненті можна через імпортування потрібного сховища. Існує можливість дістати лише необхідні дані зі сховища або всі у разі потреби. На рисунку 2.44 наведено приклад коду, який отримує лише функцію для зміни даних.

```
const setChannelData = channelStore.getState().setChannelData;
```

Рисунок 2.44 – Отримання методу зі стану Zustand

На жаль, у Zustand немає можливості відображати поточний стан сховища через спеціальний інтерфейс, як, наприклад, у Redux. Тому для перевірки стану потрібно використовувати console.log.

Запускаємо додаток та переходимо до будь-якого каналу, після чого перевіряємо консоль розробника, приклад наведено на рисунку 2.45.

Рисунок 2.45 – Дані в сховищі

Бачимо, що дані групи записалися до сховища. Отже, функція setChannelData працює правильно.

2.12.1 Порівняння Zustand та пропсів

У випадках, коли дані використовуються лише двома компонентами або їх кількість невелика, можна не застосувавати Zustand. У React можна передати дані від поточного компонента до його дочірніх елементів. Такий підхід використовується, коли дані в обох компонентах тісно зв’язані між собою та більше ніде не використовуються. У таблиці 2.2 наведено детальнішу інформацію.

Таблиця 2.2 – Порівняння Zustand з прокидуванням пропсів

Характеристика	Прокидування пропсів	Zustand
Глибина вкладеності компонентів	1-2 рівні	Багато компонентів на різних рівнях вкладеності
Обсяг даних	Найчастіше невеликий обсяг	Великий обсяг
Повторне використання	Складно реалізувати	Ідельно підходить
Взаємодія зі станом	Мінімальна логіка	Складна логіка

Висновок з таблиці такий, що пропси потрібні для простих випадків. У випадку великої системи краще використовувати Zustand.

2.13 Пошук каналів

В онлайн-месенджері користувачі можуть створювати приватні та публічні канали. Приєднатися до каналу можна, перейшовши за посиланням-запрошенням. У випадку, коли канал має публічний статус, інші клієнти мають мати можливість написати в пошук назву каналу та перейти до нього. Для цього потрібно створити пошуковий функціонал, який повератиме лише публічні канали за відповідністю назви. Після вибору каналу зі списку користувач зможе переглянути його опис та приєднатися без запрошення. Приклад бізнес-логіки зображено на рисунку 2.46.

```
504   async searchChannels(userId: number, searchValue: string) {
505     const matchedChannels = await this.channelRepository
506       .createQueryBuilder('channel')
507       .leftJoinAndSelect('channel.users', 'user')
508       .where('channel.channelName ILIKE :search', { search: `${searchValue}%` })
509       .getMany();
510
511     const userChannels: ChannelModel[] = [];
512     const globalChannels: ChannelModel[] = [];
513
514     for (const channel of matchedChannels) {
515       const userInChannel = channel.users.some((user) => user.id === userId);
516
517       if (userInChannel) {
518         userChannels.push(channel);
519       } else if (channel.isPrivate === false) {
520         globalChannels.push(channel);
521       }
522     }
523
524     return {
525       globalChannels,
526       userChannels
527     };
528   }
529 }
```

Рисунок 2.46 – Логіка пошуку каналу

Використовується ключове слово JOIN, яке дозволяє об’єднати дві таблиці в одну (у цьому випадку - таблиці channel та users). Пошук відбувається за назвою каналу. Оператор ILIKE означає, що порівняння виконується без урахування регістру.

Отриманий результат розділяємо на два масиви: userChannels та globalChannels. У кінці повертаємо результат до клієнта.

Спробуємо перевірити логіку на практиці: для цього будемо шукати раніше створений канал з назвою test_channel. Приклад результату наведено на рисунку 2.47.

Рисунок 2.47 – Пошук каналу

На екрані відображається потрібний канал відповідно до запиту. Логіка пошуку працює коректно. Форма пошуку каналів також використовує функцію debounce для зменшення кількості запитів до сервера.

2.14 Логіка видалення каналу

Користувач може вирішити видалити канал з багатьох причин. У нього має бути така можливість. Після успішного видалення всі пов'язані дані мають бути видалені відповідно до бізнес-логіки.

Приклад необхідної логіки наведено на рисунку 2.48.

```
324   async deleteChannel(channelId: number) {
325     const channel = await this.channelRepository.findOneBy({ id: channelId });
326
327     if (!channel) {
328       throw new Error(`Channel with ID ${channelId} not found`);
329     }
330
331     await this.channelRepository.remove(channel);
332     return { success: true };
333   }
```

Рисунок 2.48 – Логіка видалення каналу

Спочатку знаходимо потрібний канал за унікальним ідентифікатором. Якщо запису в базі даних не існує, повертається помилка. У разі знаходження потрібного каналу використовується метод remove для повного видалення запису. Після цього клієнту надсилається інформація про успішне видалення каналу.

2.15 Логіка пошуку підписників каналу

Адміністратор повинен мати можливість шукати підписників каналу за їхнім іменем. Така функціональність є необхідною для пошуку конкретного користувача. Пошук може виконуватися із використанням оператора LIKE у SQL. Приклад реалізації бізнес-логіки наведено на рисунку 2.49.

```
async getUsersChat(channelId: number, userName: string) {
  const query = this.channelRepository
    .createQueryBuilder('channel')
    .leftJoinAndSelect('channel.users', 'user')
    .where('channel.id = :channelId', { channelId });

  if (userName.trim().length > 0) {
    query.andWhere('user.username ILIKE :userName', {
      userName: `${userName}%`
    });
  }

  const users = await query.getOne();

  if (!users) {
    throw new Error('Channel does not have user with this userName');
  }

  return { users: users.users ?? [] };
}
```

Рисунок 2.49 – Логіка пошуку підписників каналу

Використовуючи оператор JOIN, з'єднуються дві таблиці - channel та users. Пошук каналу відбувається за допомогою унікального ідентифікатора. Якщо параметр userName має довжину більше 0, до запиту додається умова для пошуку користувача з відповідним іменем у таблиці users. Якщо користувача з таким ім'ям не знайдено - повертається помилка. В іншому випадку користувач отрмає необхідну інформацію.

Для перевірки на практиці спробуємо знайти в каналі раніше створеного користувача з іменем test_username. Приклад результату наведено на рисунку 2.50.

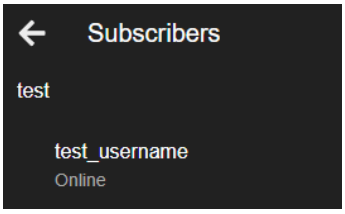

Рисунок 2.50 – Пошук підписника в каналі

У відповідь сервер повернув користувача з потрібним ім'ям. Робимо висновок, що бізнес-логіка пошуку підписника працює правильно.

2.16 Архітектура маршрутів у Next.js

Фреймворк Next.js надає гнучке налаштування маршрутів. Вони можуть бути статичними або містити динамічні сегменти. Динамічні маршрути дозволяють передавати параметри прямо в URL, що зручно для відображення сторінок користувачів, постів або товарів. Next.js підтримує вкладені маршрути, що дозволяє структурувати сторінки відповідно до логіки проєкту. Приклад наведено на рисунку 2.51.

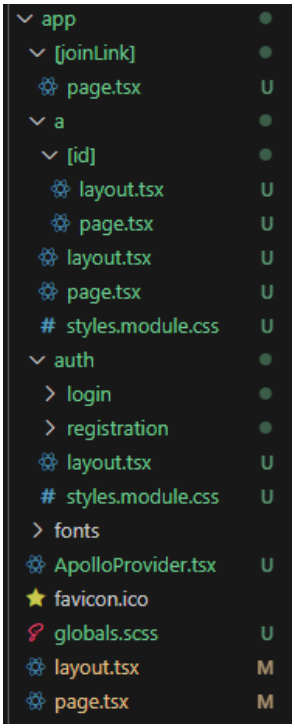

Рисунок 2.51 – Маршрути в Next.js

Для створення нового маршруту потрібно додати файл або директорію до папки app. Наприклад, щоб додати маршрут реєстрації (/auth/registration) потрібно створити папку з назвою auth, а в ній - папку registration.

Динамічні маршрути повинні записувати динамічний сегмент всередині символів [], для прикладу /a/[id]. Потім у коді можна дістати цей сегмент та здійснювати запити до сервера.

Обов'язково, папка, яка є кінцевим сегментом маршруту, повинна містити файл з назвою page.tsx. Якщо дати іншу назву, тоді фреймворк Next.js не побачить маршрут.

Ознайомимося з маршрутом, який відповідає за пошук потрібного каналу. Приклад коду наведено на рисунку 2.52.

```
11  const setChannelData = channelStore.getState().setChannelData;
12
13  let { id } = useParams();
14
15  if (Array.isArray(id)) {
16    id = id[0];
17  }
18
19  if (typeof id === "string") {
20    id = decodeURIComponent(id).slice(1);
21  }
22
23  const isNumericId = /^-?\d+$/.test(id ?? "");
24
25  const queryVariables = isNumericId
26    ? { channelId: Number(id) }
27    : { publicLink: id?.replace(/^@/, "") || "" };
28
29  const { data, loading } = useQuery(GET_CHANNEL_INFO, {
30    variables: queryVariables,
31    skip: !id,
32  });
```

Рисунок 2.52 – Пошук потрібного каналу

Використовуючи стандартний хук useParams, який імпортується з бібліотеки next/navigation, отримаємо динамічний сегмент. Якщо сегмент буде масивом потрібно дістати лише перший елемент, тобто той, який має індекс 0.

У випадку, якщо динамічний сегмент має тип даних string, він декодується за допомогою функції decodeURIComponent та видаляється перший символ рядка за допомогою методу slice.

Пам'ятаємо, що коли канал має приватний статус - його пошук відбувається за унікальним ідентифікатором. Коли канал має публічний статус - пошук здійснюється за допомогою публічного посилання. Саме тому у рядку 23 перевіряється, чи є сегмент числом, за допомогою регулярного виразу.

Якщо динамічний сегмент має вірну структуру, тоді хук useQuery відправляє запит до серверу.

2.17 Оновлення даних каналу

Створивши канал, користувач можливо забажає змінити назву або опис. Така функціональність є необхідною для створення сучаних додатків, які орієнтовані на зручність та контроль з боку користувача. Приклад реалізації зображено на рисунку 2.53.

```
143   async changeChannelName(
144     channelId: number,
145     newChannelName: string,
146     newDescription?: string
147   ) {
148     const channel = await this.channelRepository.findOne({
149       where: { id: channelId }
150     });
151
152     if (!channel) {
153       throw new Error('Channel not found');
154     }
155
156     channel.channelName = newChannelName;
157     channel.description = newDescription;
158     await this.channelRepository.save(channel);
159
160     return {
161       newChannelName: newChannelName,
162       newDescription: newDescription
163     };
164   }
```

Рисунок 2.53 – Оновлення даних каналу

Знаходимо потрібний канал. Якщо запису не існує - повертаємо помилку. В іншому випадку оновлюємо дані та зберігаємо зміни за допомогою методу save. У кінці до клієнта відправляються нові дані.

Спробуємо на практиці змінити дані каналу. Відкриваємо додаток і переходмо до створеного каналу (test_channel), після чого змінюємо його назву на test_channel1. Заповнивши форму для оновлення даних каналу, натискаємо зберегти. У таблиці channel повині змінитися відповідні дані каналу. Приклад наведено на рисунку 2.54.

	id [PK] integer	channelName character varying	description character varying	amountSubscriber integer	avatarUrl character varying	publicLink character varying	privateLink character varying (16)	isPrivate boolean
1	1	test_channel1	test_description	1	[null]		GknucpiH9fg_14KS	false

Рисунок 2.54 – Таблиця channel

У таблиці channel канал з унікальним ідентифікатором змінив дані. Отже, логіка оновлення даних працює вірно.

2.18 Отримання останніх повідомлень

Користувачі повинні отримувати останні повідомлення для кожного чату. Такий функціонал дозволить отримувати своєчасну інформацію про кожний канал.

Для цього потрібно реалізувати запит, який повертатиме список каналів разом із останнім повідомленням для кожного з них. Це можна зробити за допомогою SQL-запиту з об'єднанням таблиць або підзапиту, що вибирає останній запис за датою. У фреймворку Nest.js така логіка може бути реалізована через окремий сервіс або метод репозиторію. Отримані дані зручно виводити у списку чатів, що дозволяє користувачу швидко оцінити активність у кожному каналі.

На сервері потрібно додати підписку на метод createMessage. Після цього на клієнті потрібно створити новий API для повідомленнь, приклад наведено на рисунку 2.55.

```
63  export const LAST_MESSAGE_SUBSCRIPTION = gql(`
64    subscription lastMessageUpdated($channelIds: [Int!]!) {
65      lastMessageUpdated(channelIds: $channelIds) {
66        channelId
67        content
68      }
69    }
70  `);
```

Рисунок 2.55 – Підписка на останнє повідомлення

Метод приймає список унікальних ідентифікаторів та повертає необхідні дані. На рисунку 2.56 наведено логіку відображення повідомлень.

```
14  const [lastMessages, setLastMessages] = useState<Record<number, string>>({});
15
16  const channelIds = data?.chats?.map((chat) => chat.id) || [];
17
18  useSubscription(LAST_MESSAGE_SUBSCRIPTION, {
19    variables: { channelIds },
20    skip: channelIds.length === 0,
21    onData: ({ data }) => {
22      const message = data.data?.lastMessageUpdated;
23      if (message) {
24        setLastMessages((prev) => ({
25          ...prev,
26          [message.channelId]: message.content,
27        }));
28      }
29    },
30  });
```

Рисунок 2.56 – Логіка відображень останніх повідомлень

Змінна lastMessages зберігає словник, де ключем є унікальний ідентифікатор чату, а значення - повідомлення. Далі з об'єкта чатів отримуються унікальні ідентифікатори, які записуються у змінну channelIds. Хук useSubscription з'єднує клієнтську частину з відповідним методом на сервері. Коли створюється нове повідомлення, воно додається до словника lastMessages.

2.18 Створення контейнеру

Використовуючи Docker, можна створити контейнери для серверної та клієнтської частин додатку. Це потрібно для зручного перенесення, розгортання та масштабування проєкту на різних середовищах. Контейнер містить усе необхідне для роботи сервісу, що допомогає уникнути проблем із залежностями. У Docker можна забезпечити однакове середовище для розробки або тестування. Для цього створюються окремі Dockerfile для клієнта та сервера, а їх взаємодія налаштовується через docker-compose.yml. Це дозволяє запускати весь стек застосунку однією командою, що значно спрощує роботу команди. Також, Docker забезпечує ізоляцію сервісів, що підвищує безпеку та стабільність проєкту.

Основні файли, які потрібно створити:

а) Dockerfile - це набір покрокових інструкцій, використовуючи, які Docker створює контейнер;

б) Docker Compose – дозволяє одночасно запустити кілька контейнерів.

У клієнтській та серверній частині потрібно створити Dockerfile. Приклад налаштування серверного Dockerfile наведено на рисунку 2.57.

```
1  FROM node:18-alpine
2  WORKDIR /app
3  COPY package*.json .
4  RUN npm install
5  COPY . .
```

Рисунок 2.57 – Серверний Dockerfile

Збірка базується на офіційному образі Node.js версії 18 на базі Alpine. Цей образ є невеликий за розміром, що надає швидке завантаження. Далі вказується робоча директорія з назвою /app. Потім копіюються всі файли з package.json і package-lock.json до директорію /app. Всередині контейнеру виконується команда npm install. У кінці копіюються всі інші файли проєкту до робочої директорії контейнера /app.

Приклад клієнтського Dockerfile наведено на рисунку 2.58.

```
1  FROM node:18-alpine
2  WORKDIR /app
3  COPY package*.json .
4  RUN npm install
5  COPY . .
6  RUN npm run dev
```

Рисунок 2.58 – Клієнтський Dockerfile

Основою для збірки є образ Node.js версії 18 на базі Alpine. Встановлюється робоча директорія з назвою /app. Копіюються файли package.json і package-lock.json до директорії /app. Контейнер виконує команду npm install. Наступна команда копіює всі інші файли з локальної директорії у контейнер /app. Наприкінці виконується команда для запуску клієнтської частини.

Створивши два необхідні Dockerfile потрібно додати Docker Compose. Приклад його реалізації наведено на рисунку 2.59.

```
version: "3.8"

Run All Services
services:
  Run Service
  client:
    build: ./client
    ports:
      - "8080:80"
  Run Service
  server:
    buid: ./server
    ports: -"3005:3005"
    command: npm run start

```

Рисунок 2.59 – Docker Compose файл

Встановлена 3.8 версія синтаксису Docker. Створено два сервіси: клієнтський і серверний. Команда build створює образи з Dockerfile, які знаходяться у відповідних директоріях. Потім вказуються зовнішні та внутрішні порти для кожного сервісу.

2.19 Завантаження застосунку до Git

Використання Git дозволяє зручно керувати версіями застосунку. Спочатку потрібно створити репозиторій на Github. У консолі за допомогою команди git add додаємо файли до області підготовки, потім зберігаємо зміни використовуючи git commit, і на останок - відправляємо зміни до віддаленого репозиторію. Приклад результату наведено на рисунку 2.60.

Рисунок 2.60 – Репозиторій в Github

Усі дані збереглися у віддаленому репозиторії. Тепер у локальному репозиторії можна робити зміни без страху щось видалити. У віддаленому репозиторії можна відстежувати всі зміни в коді. Також GitHub надає зручний інтерфейс для перегляду історії комітів та управління гілками. Крім того, GitHub дозволяє створювати pull request для обговорення та перевірки коду перед злиттям у основну гілку. Це корисно при командній роботі, де важливо проводити перевірку коду. Завдяки цьому забезпечується контроль якості та зменшується ймовірність помилок у кодовій базі.

Основні команди для роботи з GitHub:

а) git add – додає файли до області підготовки;

б) git commit – фіксує зміни у локальному репозиторію;

в) git push – відправляє зміни з локального репозиторію до віддаленого;

г) git pull – отримує зміни з віддаленого репозиторію;

ґ) git merge – об’єднує дві гілки в одну;

д) git init – ініціалізує новий Git-репозиторій;

е) git branch – створює нові гілки;

є) git checkout – змінює поточну гілку або стан проєкту.

Ці команди є основою для роботи в команді розробників. Вони дозволяють створювати гілки, фіксувати зміни, синхронізувати локальний репозиторій із віддаленим та багато іншого.

Завдяки Git кожен розробник може працювати незалежно, не заважаючи іншим членам команди. У разі конфліктів при злитті гілок, Git надає інструменти для їх ефективного вирішення. Також існує можливість створення pull request'ів, які дозволяють обговорити та перевірити зміни перед їх інтеграцією в основну гілку. Використання GitHub у поєднанні з Git забезпечує надійне зберігання коду, контроль версій і зручну колаборацію над проєктами.

На рисунку 2.61 наведено приклад жєиттєвого циклу Git.

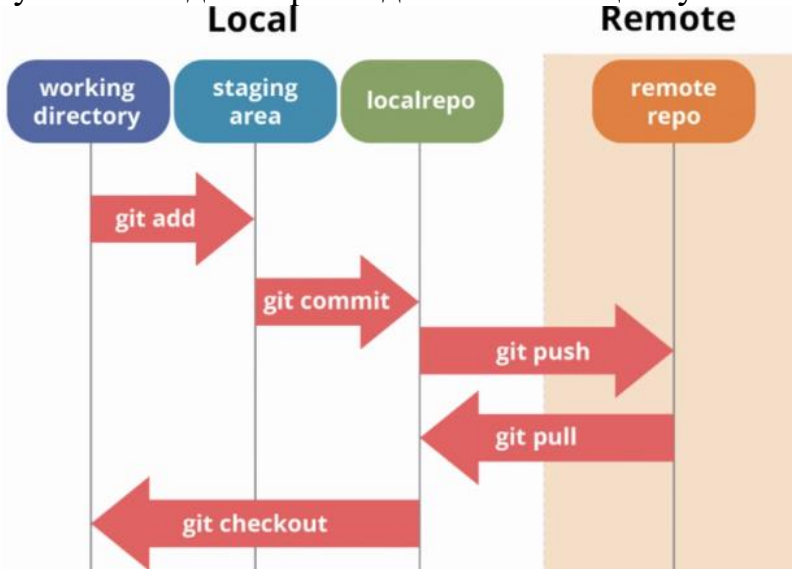

Рисунок 2.61 – Життєвий цикл Git

Локальний репозиторій розподіляється на три частини: робочу директорію, область підготовки та локальний репозиторій. Спочатку файли додаються в область підготовки за допомогою команди git add. Після цього зміни фіксуються в локальному репозиторії командою git commit. Для того щоб передати зміни до віддаленого репозиторію, використовується команда git push, а для отримання оновлень – git pull. Команда git checkout дозволяє перемикатися між гілками або станами проєкту. Такий підхід забезпечує чітку організацію роботи з кодом та зручний контроль версій.

ВИСНОВКИ

Ознайомилися з основними поняттями та технологіями, необхідними для створення онлайн месенджера. Зрозуміли, як браузер завантажує сайти в Інтернеті за допомогою DNS. Також отримали базові знання про HTML і CSS.

Вивчили різницю між двома протоколами - HTTP та HTTPS. Потім ознайомилися з архітектурним підходом REST API. Порівняли його з альтернативним підходом до створення API – GraphQL. Вивчили спосіб авторизації через використання refresh і access токенів. Далі зрозуміли, як реалізовувати передачу повідомлень у реальному часі за допомогою Socket.IO.

Визначили, що PostgreSQL є популярною реляційною базою даних. Зрозуміли наскільки важливо використовувати TypeScript у великих проєктах. Ознайомилися з популярними фреймворками Next.js та Nest.js. Порівняли менеджери пакетів Yarn та NPM. Наступим кроком було вивчення основ роботи з Git. Розглянули, як передавати інформацію між компонентами незалежно від структуру проєкту за допомогою Zustand.

На практиці онлайн месенджер реалізує такий функціонал: авторизація, реєстрація, відправка повідомлень, створення чатів, створення посилань-запрошень, можливість зробити канал публічним або приватним, а також додавання реакцій. Застосунок використовує сучасні технології та методи захисту. Також онлайн месенджер має мобільну адаптацію, що покращує користувацький досвід.

У проєкті використовувалася архітектура FSD, що забезпечило логічне розташування файлів і зменшило час розробки застосунку. Для оптимізації взаємодії з сервером реалізовано метод debounce, що зменшує кількість запитів. Навчилися створювати таблиці з різними типами зв’язків: one-to-one, one-to-many та many-to-many. Вивчили, як працювати з динамічними маршрутами в Next.js. Зробили контейнери для клієнтської та серверної частини онлайн месенджера. Завантажили застосунок до віддаленого репозиторію Git.

Отже, робимо висновок, що створений онлайн месенджер повністю відповідає поставленим йому вимогам.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Domain Name System. URL: https://www.geeksforgeeks.org/domain-name-system-dns-in-application-layer/ (дата звернення: 15.12.2024).
2) Document Object Model structure. URL: https://www.w3.org/TR/REC-DOM-Level-1/introduction.html (дата звернення: 22.12.2024).
3) Browser rendering process. URL: https://blog.logrocket.com/how-browser-rendering-works-behind-scenes/ (дата звернення: 27.12.2024).
4) Cascading in CSS. URL: https://www.geeksforgeeks.org/what-does-the-cascading-portion-of-css-means/ (дата звернення: 02.01.2025).
5) Introduction to HTTP. URL: https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/ (дата звернення: 13.01.2025).
6) HTTP status codes. URL: https://www.techtarget.com/whatis/definition/HTTP-Hypertext-Transfer-Protocol (дата звернення: 15.01.2025).
7) HTTPS working principle. URL: https://www.cloudflare.com/learning/ssl/what-is-https/ (дата звернення: 16.01.2025).
8) Принципи REST-архітектури. URL: https://foxminded.ua/shcho-take-rest-api/ (дата звернення: 17.01.2025).
9) GraphQL query language. URL: https://apidog.com/blog/what-is-graphql/ (дата звернення: 17.01.2025).
10) Sokcet.IO communication library. URL: https://apidog.com/blog/what-is-graphql/ (дата звернення: 19.01.2025).
11) WebSockets and long polling comparison. URL: https://ably.com/blog/websockets-vs-long-polling (дата звернення: 21.01.2025).
12) Introduction to PostgreSQL. URL: https://aiven.io/blog/an-introduction-to-postgresql (дата звернення: 26.01.2025).
13) ACID властивості. URL: https://aiven.io/blog/an-introduction-to-postgresql (дата звернення: 29.01.2025).
14) MongoDB NoSQL database. URL: https://www.oracle.com/database/mongodb/ (дата звернення: 02.02.2025).
15) Ознайомлення з TypeScript. URL: https://foxminded.ua/typescript/ (дата звернення: 05.02.2025).
16) Virtual DOM concept. URL: https://www.freecodecamp.org/news/what-is-the-virtual-dom-in-react/ (дата звернення: 07.02.2025).
17) Принципи SOLID. URL: https://journal.gen.tech/post/principi-solid-sho-ce-ta-yak-yih-zastosovuvati-kejsi-ta-porady (дата звернення: 13.02.2025).

18) Основні команди для роботи з GIT. https://foxminded.ua/git-komandy/ (дата звернення: 16.02.2025).
19) Zustand state management. URL: https://medium.com/@thekinv21/what-is-zustand-7ffa8cea6043 (дата звернення: 16.02.2025).
20) Гнучкий та адаптивний дизайн. URL: https://drukarnia.com.ua/articles/responsive-chi-adaptive-dizain-KAnOA (дата зверення: 20.02.2025).
21) Docker container platform. URL: https://www.oracle.com/ua/cloud/cloud-native/container-registry/what-is-docker/ (дата звернення: 26.02.2025).
22) Архітектура FSD. URL: https://feature-sliced.github.io/documentation/ru/docs/get-started/overview (дата звернення: 05.03.2025).