ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ МОБІЛЬНОГО ЗАСТОСУНКУ З ВИКОРИСТАННЯМ WEBSOCKET ТЕХНОЛОГІЙ РЕАЛЬНОГО ЧАСУ»

на здобуття освітнього ступеня магістра
зі спеціальності 123 Комп'ютерна інженерія
(код, найменування спеціальності)
освітньо-професійної програми «Комп'ютерні системи та мережі»
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ *Олексій ДАНЕВИЧ*
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр.КСДМ-62
Олексій ДАНЕВИЧ
(ім'я, ПРІЗВИЩЕ)

Керівник: Віталій ВОЛОХІН
к.т.н, доцент *(ім'я, ПРІЗВИЩЕ)*

Рецензент: _____
науковий ступінь, вчене звання *(ім'я, ПРІЗВИЩЕ)*

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерна інженерія

Ступінь вищої освіти Магістр

Спеціальність 123 Комп'ютерна інженерія

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ
Завідувач кафедрою Наталія ЛАЩЕВСЬКА
_____Ім'я, ПРІЗВИЩЕ
«_____» _____ 2024р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Данквич Олексій Володимирович
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: « Розробка серверної частини мобільного застосунку з використанням WebSocket технологій реального часу »
керівник кваліфікаційної роботи Віталій ВОЛОХІН, к.т.н., доцент,
(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024 р. № 320.
2. Строк подання кваліфікаційної роботи «29» грудня 2024 р.
3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, Технології передачі даних у реальному часі.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   1. Теоретичні аспекти забезпечення стабільної та ефективної роботи WebSocket-з'єднань у реальному часі.
   2. Проектування серверної архітектури з урахуванням масштабованості.
   3. Практична реалізація та тестування ефективності розробленого рішення.

5. Перелік ілюстративного матеріалу: *презентація*
   1. Актуальність роботи.
   2. Websocet.
   3. Порівняння Node.js vs. Fastapi
   4. Бази даних.
   5. Nginx
   6. Відображення респонсу та конфігурація SSL
   7. Ефективна конфігурація сервера
   8. Блок схема
   9. Демонстрація
   10. Висновки
   11. Публікації та апробація роботи
6. Дата видачі завдання «01» вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Підбір технічної літератури	01.10.2024р. 05.10.2024р.	
2.	Дослідження основ хмарних обчислень та балансування навантаження	05.10.2024р. 10.10.2024р.	
3.	Визначення ключових параметрів комунікації	10.10.2024р. 15.10.2024р.	
4.	Моделювання та оцінка серверної архітектури	15.10.2024р. 20.10.2024р.	
5.	Оформлення роботи, висновків	20.10.2024р. 27.10.2024р.	
6.	Розробка демонстраційного матеріалу, доповіді	27.10.2024р. 03.11.2024р.	

Здобувач вищої освіти _____ Олексій ДАНЕВИЧ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ Артем АНТОНЕНКО
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій

ПОДАННЯ
ГОЛОВІ ДЕРЖАВНОЇ ЕКЗАМЕНАЦІЙНОЇ КОМІСІЇ
ЩОДО ЗАХИСТУ КВАЛІФІКАЦІЙНОЇ РОБОТИ
на здобуття освітнього ступеня магістр

Направляється здобувач Данесич О.В. до захисту кваліфікаційної роботи
(прізвище та ініціали)
за спеціальністю 123 Комп'ютерна інженерія
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні системи та мережі
(назва)
на тему: «Розробка серверної частини мобільного застосунку з використанням WebSocket технологій реального часу».
Кваліфікаційна робота і рецензія додаються.

В.о. директора ННІІТ _____ Катерина НЕСТЕРЕНКО
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Висновок керівника кваліфікаційної роботи

Здобувач Даневич О.В. показав добру теоретичну підготовку, уміння володіти сучасними комп'ютерними технологіями, а також здатність ефективно користуватися навчальною і науково-технічною літературою. Під час виконання завдань, поставлених керівником, виявив старанність, відповідальність і здатність до самостійної роботи в галузі програмної інженерії.

Кваліфікаційну роботу студента, присвячену розробці серверної частини мобільного застосунку з використанням WebSocket технологій реального часу, виконано на високому рівні, що дозволяє оцінити її на «добре» та рекомендувати присвоєння здобувачу кваліфікації «магістр з комп'ютерної інженерії».

Керівник кваліфікаційної роботи _____ Віталій ВОЛОХІН
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

«___» _____ 2024 року

Висновок кафедри про кваліфікаційну роботу

Кваліфікаційна робота розглянута. Здобувач *Даневич О.В.* допускається до захисту даної роботи в Екзаменаційній комісії.

Завідувач кафедрою Комп'ютерної інженерії _____ Наталія ЛАЩЕВСЬКА
(назва) *(підпис)* *(Ім'я, ПРІЗВИЩЕ)*

ВІДГУК РЕЦЕНЗЕНТА
на кваліфікаційну магістерську роботу

здобувача вищої освіти Даневича Олексія Володимировича
(прізвище, ім'я, по батькові)
на тему: «Розробка серверної частини мобільного застосунку з використанням WebSocket технологій реального часу»

Актуальність:

Кваліфікаційна магістерська робота здобувача вищої освіти Даневича Олексія Володимировича присвячена розробці та впровадженню серверної частини мобільного застосунку з використанням WebSocket технологій реального часу. З огляду на актуальність застосування сучасних технологій для забезпечення швидкої та ефективної обробки даних у реальному часі, тема роботи є надзвичайно важливою. Використання WebSocket дозволяє створити систему, яка забезпечує надійний і стабільний обмін інформацією між сервером та клієнтами, сприяючи підвищенню продуктивності, масштабованості та інтерактивності мобільного застосунку.

Позитивні сторони:
1. У ході кваліфікаційної магістерської роботи було детально проаналізовано сучасні підходи до побудови серверних частин мобільних застосунків, що базуються на технологіях WebSocket, а також визначено їх переваги у порівнянні з традиційними методами HTTP-з'єднань.
2. Були запропоновані оптимальні архітектури серверної частини для мобільного застосунку, яка забезпечує стабільну роботу в умовах реального часу, що є особливо актуальним для додатків із високими вимогами до продуктивності та швидкості обробки даних.
3. В роботі успішно інтегровано сучасні інструменти для обробки та передачі даних у реальному часі, а також забезпечено відповідність створеної архітектури принципам масштабованості та надійності.

Недоліки
1. Робота недостатньо охоплює практичний аналіз продуктивності в умовах різної кількості одночасних з'єднань, що могло б краще продемонструвати потенціал впровадженої системи
2. У тексті бажано було б більш детально розглянути методи безпеки передачі даних через WebSocket, враховуючи сучасні кіберзагрози.

Висновок: *кваліфікаційна магістерська робота заслуговує оцінку "добре", а здобувач Даневич О.В. заслуговує присвоєння кваліфікації: магістр з комп'ютерної інженерії.*

Рецензент: _____ _____
науковий ступінь, вчене звання *підпис* *Ім'я, ПРІЗВИЩЕ*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття ступня магістр: 82 стор., 33 рис., 36 джерел.

Мета роботи – проєктування та реалізація серверної архітектури мобільного застосунку з використанням WebSocket для забезпечення комунікації в реальному часі, що забезпечить ефективну взаємодію користувачів із мінімальними затримками.

Об'єкт дослідження – обмін повідомленнями в реальному часі через WebSocket у мобільних застосунках.

Предмет дослідження – архітектура серверної частини застосунку для забезпечення стабільного WebSocket-з'єднання та масштабованої комунікації.

Короткий зміст роботи: У роботі досліджено сучасні методи комунікації в реальному часі та проаналізовано можливості WebSocket для мобільних застосунків. Виявлено ключові параметри ефективного управління з'єднаннями та передачі даних. Розроблено концепцію серверної архітектури, яка передбачає використання WebSocket для стабільної та швидкої комунікації.

У практичній частині реалізовано сервер WebSocket, проведено його тестування та аналіз продуктивності в умовах різної інтенсивності навантаження. Розглянуто можливості масштабування та вдосконалення архітектури для подальшого розвитку.

КЛЮЧОВІ СЛОВА: WEBSOCKET, МОБІЛЬНИЙ ЗАСТОСУНОК, КОМУНІКАЦІЯ В РЕАЛЬНОМУ ЧАСІ, СЕРВЕРНА АРХІТЕКТУРА, МАСШТАБУВАННЯ, ПРОДУКТИВНІСТЬ, СТАБІЛЬНІСТЬ.

ABSTRACT

The text part of the qualification work for obtaining a master's degree: 82 pages, 33 figures, 36 sources.

Objective – to design and implement the server architecture of a mobile application using WebSocket to ensure real-time communication, enabling efficient user interaction with minimal latency.

Object of research – real-time message exchange via WebSocket in mobile applications.

Subject of research – the server-side architecture of the application to ensure stable WebSocket connections and scalable communication.

Summary of the Work

The work investigates modern methods of real-time communication and analyzes the potential of WebSocket for mobile applications. Key parameters for effective connection management and data transmission have been identified. A concept of server architecture utilizing WebSocket for stable and fast communication has been developed.

The practical part involves the implementation of a WebSocket server, followed by testing and performance analysis under varying load intensities. Opportunities for scaling and improving the architecture for future development are also considered.

KEY WORDS: WEBSOCKET, MOBILE APPLICATION, REAL-TIME COMMUNICATION, SERVER ARCHITECTURE, SCALABILITY, PERFORMANCE, STABILITY.

ЗМІСТ

ВСТУП

У сучасному цифровому світі мобільні застосунки відіграють значну роль у житті користувачів, забезпечуючи швидкий доступ до інформації та можливість миттєвого спілкування. Для багатьох типів застосунків, таких як месенджери, онлайн-ігри, торгові платформи та фінансові сервіси, ключовим є забезпечення стабільної комунікації в реальному часі. Використання традиційних протоколів, таких як HTTP, має низку обмежень, зокрема високі затримки під час передачі даних. У відповідь на ці виклики протокол WebSocket забезпечує постійне двонаправлене з'єднання між клієнтом і сервером, що дозволяє здійснювати обмін даними практично миттєво.

Ця кваліфікаційна робота присвячена розробці серверної архітектури мобільного застосунку з використанням WebSocket для забезпечення ефективної комунікації в реальному часі. Мета роботи полягає у створенні рішення, яке забезпечить стабільну взаємодію між користувачами з мінімальними затримками та можливістю масштабування системи під великим навантаженням.

Робота охоплює аналіз сучасних підходів до передачі даних у реальному часі та можливостей WebSocket у порівнянні з іншими протоколами, такими як MQTT і SSE. Особлива увага приділяється питанням безпеки та стабільності WebSocket-з'єднань, зокрема автентифікації користувачів та шифруванню даних.

У практичній частині буде реалізовано WebSocket-сервер для мобільного застосунку та проведено його тестування з використанням різних сценаріїв навантаження. Буде оцінено продуктивність і стабільність архітектури в умовах високої активності користувачів, а також розглянуто способи масштабування для забезпечення стійкої роботи в майбутньому.

Результати дослідження та запропоновані рішення можуть бути корисними для розробників, які працюють над застосунками з високими вимогами до швидкості обміну даними. Висновки цієї роботи також можуть слугувати основою для подальших досліджень у сфері комунікаційних технологій та оптимізації серверних архітектур для роботи в умовах реального часу

РОЗДІЛ 1 ТЕОРЕТИЧНІ ОСНОВИ ТА АНАЛІЗ ТЕХНОЛОГІЙ ДЛЯ РЕАЛІЗАЦІЇ СЕРВЕРНОЇ АРХІТЕКТУРИ

1.1 Огляд методів комунікації в реальному часі

У сучасному світі, де миттєва комунікація стала невід'ємною частиною повсякденного життя, розробники стикаються з необхідністю обирати між різними методами передачі даних у реальному часі. Це стосується не лише соціальних мереж і месенджерів, але й фінансових сервісів, онлайн-ігор та багатьох інших застосунків. У цьому огляді розглянемо основні методи комунікації в реальному часі, зокрема HTTP, WebSocket, MQTT та Server-Sent Events (SSE).

HTTP (Hypertext Transfer Protocol) є основним протоколом для передачі даних в Інтернеті. Хоча HTTP не призначений для реального часу, він використовується в багатьох веб-застосунках. Клієнт (браузер) надсилає запит на сервер, який відповідає з даними. Цей механізм може бути повільним для застосунків, що вимагають миттєвого оновлення даних.

Основні переваги HTTP полягають у простоті реалізації та широкій підтримці на всіх платформах зображено на рис. 1.1. Однак він має і свої недоліки: високу затримку через постійні запити, що робить його невідповідним для застосувань із високими вимогами до швидкості. HTTP також використовує моделі запиту/відповіді, які можуть викликати затримки в отриманні даних.

REST API є невід'ємною частиною сучасної веб-розробки, що дозволяє ефективно взаємодіяти між клієнтом і сервером через стандартні HTTP запити. У сучасному світі, де миттєва комунікація є важливим аспектом, розробники стикаються з вибором між різними методами передачі даних у реальному часі. Це стосується не лише соціальних мереж і месенджерів, але й фінансових сервісів, онлайн-ігор та багатьох інших застосунків.

Основним протоколом для передачі даних в Інтернеті є HTTP (Hypertext Transfer Protocol). Хоча HTTP не призначений для реального часу, він використовується у багатьох веб-застосунках. Клієнт (браузер) надсилає запит на сервер, який відповідає з даними. Основна перевага HTTP – простота реалізації та

підтримка на всіх платформах. Однак HTTP має й недоліки, такі як висока затримка через постійні запити, що робить його невідповідним для застосунків із високими вимогами до швидкості. HTTP використовує моделі запиту/відповіді, які можуть викликати затримки в отриманні даних.

З іншого боку, WebSocket забезпечує двосторонню комунікацію між клієнтом і сервером, що дозволяє даним передаватися миттєво без необхідності багаторазових HTTP-запитів. Після початкового встановлення з'єднання, клієнт і сервер можуть обмінюватися даними в реальному часі. Це особливо корисно для застосунків, що потребують високого рівня інтерактивності, таких як онлайн-ігри, фінансові сервіси або навіть інтерактивні карти. WebSocket дозволяє обійти обмеження HTTP, забезпечуючи низьку затримку і високу швидкість передачі даних, що робить його ідеальним для застосунків, що вимагають миттєвого оновлення даних.

MQTT (Message Queuing Telemetry Transport) – це ще один протокол для передачі даних у реальному часі, який часто використовується для Інтернету речей (IoT). MQTT є легким і ефективним протоколом для з’єднання з пристроями, що потребують низької затримки і надійної передачі даних через нестабільні мережі. Він підтримує "опубліковано/підписано" модель, яка дозволяє пристроям надсилати дані на сервер і отримувати відомості відповідно до потреб.

Server-Sent Events (SSE) – це ще один метод передачі даних у реальному часі, який використовує сервер для надсилання оновлень до клієнта. SSE дозволяє серверу відправляти події клієнтам без їх запиту. Це ідеальне рішення для застосунків, які потребують одноразових або рідкісних оновлень, таких як сповіщення або моніторинг.

Кожен з цих методів передачі даних має свої переваги та недоліки, і вибір залежить від специфіки застосунку та вимог до комунікації в реальному часі. REST API залишається основним вибором для багатьох веб-застосунків, але для застосунків, що вимагають високого рівня інтерактивності або постійного з’єднання, такі протоколи як WebSocket, MQTT і SSE можуть бути більш підходящими рішеннями.

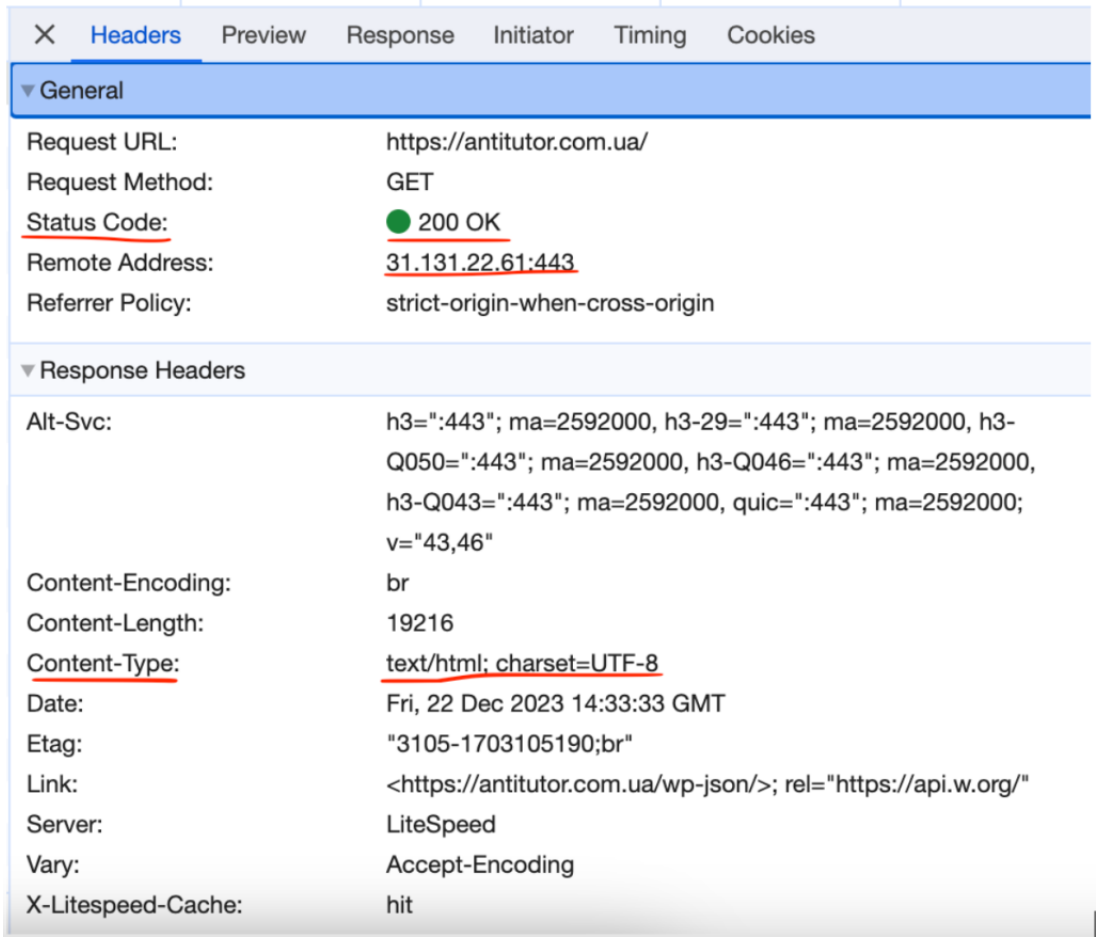


Рисунок 1.1 – Приклад моделі запиту та відповіді HTTP

WebSocket — це протокол, який дозволяє встановити постійне з'єднання між клієнтом і сервером[1]. Це означає, що дані можуть передаватися в обох напрямках у будь-який час, що робить WebSocket ідеальним для застосунків, які потребують миттєвої реакції. WebSocket зменшує затримки завдяки підтримці одночасних двонаправлених з’єднань, що є критично важливим для онлайн-ігор, чатів і фінансових сервісів зображено на рис. 1.2.

Серед переваг WebSocket слід відзначити низьку затримку завдяки постійному з'єднанню та двонаправлений обмін даними. Проте реалізація WebSocket може бути складнішою, ніж HTTP, і він потребує більше ресурсів на сервері, що може призвести до проблем із масштабуванням.

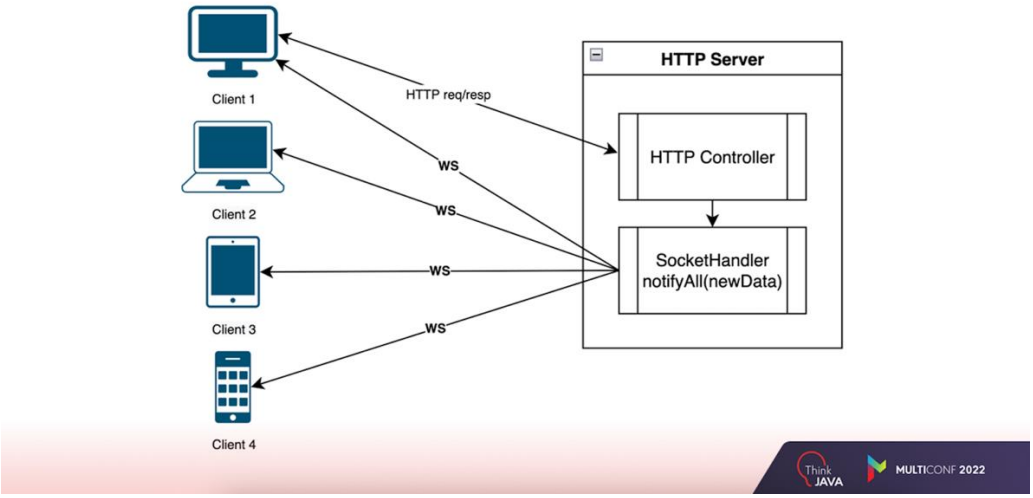


Рисунок 1.2 – Ілюстрація WebSocket-з'єднання

MQTT (Message Queuing Telemetry Transport) — це легкий протокол обміну повідомленнями, який призначений для з'єднань з обмеженими ресурсами. Він часто використовується в IoT (Internet of Things) зображено на рис. 1.3 для передачі даних з датчиків на сервери. MQTT є оптимальним вибором для пристроїв з обмеженою пропускною здатністю та енергетичними ресурсами, оскільки він підтримує різні рівні якості обслуговування (QoS).

Переваги MQTT включають низьку витрату трафіку та високу надійність. Він використовує модель "публікація-підписка", що дозволяє зменшити навантаження на мережу. Водночас, він може бути складним для налаштування та не завжди підходить для веб-додатків, оскільки вимагає використання додаткових брокерів повідомлень.

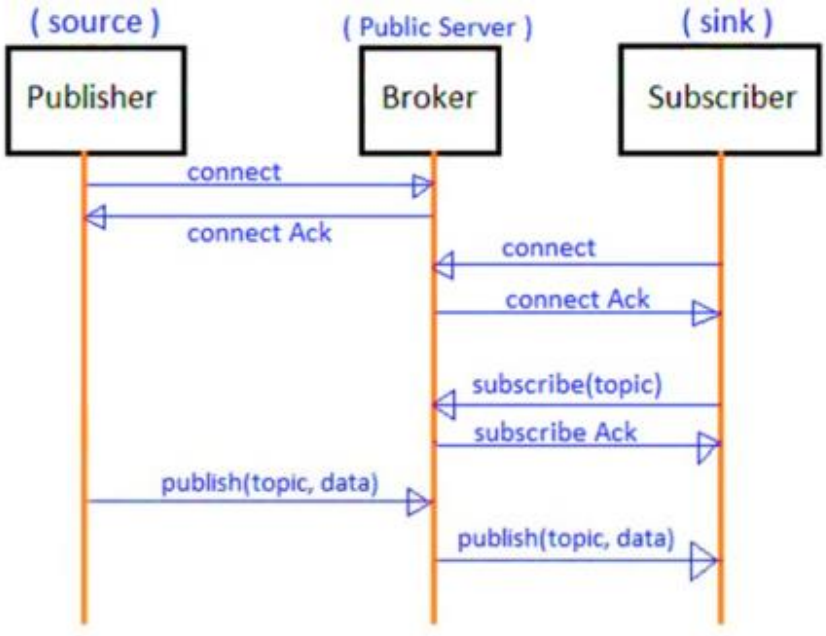


Рисунок 1.3 – Ілюстрація структури MQTT

Server-Sent Events — це технологія, яка дозволяє серверам надсилати оновлення даних в браузер через HTTP-з'єднання[2]. SSE зручний для застосунків, де сервер періодично надсилає дані, такі як новини або сповіщення. SSE автоматично повторно підключає з'єднання в разі його обриву, що робить його зручним для безперервної передачі даних.

Серед переваг SSE варто відзначити простоту реалізації на стороні сервера та автоматичне повторне підключення зображено на рис. 1.4. Однак SSE підтримує лише односпрямований обмін даними, що може бути обмеженням для деяких застосунків, де потрібна двонаправлена комунікація.

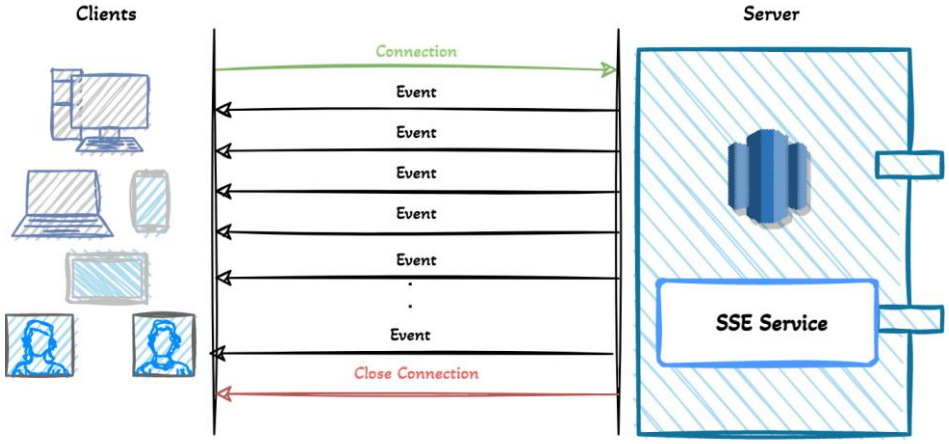


Рисунок 1.4 – Приклад Server-Sent Events

Кожен з описаних методів має свої особливості, переваги та недоліки. У таблиці нижче наведено коротке порівняння основних характеристик

Таблиця 1.1 – Порівняння характеристик методів

Метод	Двонаправлений обмін	Підтримка реального часу	Проста реалізація	Ресурси сервера
HTTP	Ні	Ні	Так	Високі
WebSocket	Так	Так	Ні	Помірні
MQTT	Так	Так	Ні	Низькі
SSE	Ні	Так	Так	Помірні

Вибір методу комунікації в реальному часі залежить від конкретних вимог проекту. WebSocket ідеальний для чату або онлайн-ігор, тоді як MQTT чудово підходить для IoT-систем, де необхідно передавати дані з датчиків. HTTP

залишається популярним для традиційних веб-додатків, тоді як SSE може бути вибрано для застосунків, що потребують періодичного отримання оновлень від сервера.

Завдяки розумінню особливостей кожного методу, розробники можуть зробити обґрунтований вибір у відповідності до потреб свого проекту. Це дозволяє створювати більш ефективні та надійні системи, здатні відповідати сучасним вимогам до комунікації.

1.2 Порівняння WebSocket з іншими протоколами для передачі данних у реальному часі

Сьогодні реальна комунікація стала невід'ємною частиною багатьох веб-додатків. Із зростанням популярності онлайн-сервісів, які вимагають миттєвого обміну даними, важливо вибрати правильний протокол для передачі інформації в реальному часі. Серед різних протоколів, які існують для цих цілей, WebSocket виділяється як один з найбільш ефективних. У цій статті ми розглянемо WebSocket в контексті інших методів, таких як HTTP, Server-Sent Events (SSE) та MQTT, і проаналізуємо їх переваги та недоліки.

WebSocket — це протокол, що забезпечує двосторонній зв'язок між клієнтом і сервером, дозволяючи обмін даними без постійного повторного підключення. Це відрізняє його від традиційного HTTP, який працює за принципом запит-відповідь. Коли клієнт використовує HTTP, він відправляє запит на сервер, і той повертає відповідь. Це може бути ефективно для простих веб-додатків, але для застосунків, які вимагають постійного обміну даними, це викликає затримки.

Однією з основних переваг WebSocket є те, що він дозволяє створити постійне з'єднання, яке зберігається протягом усього сеансу. Це означає, що дані можуть передаватися в обох напрямках у будь-який момент. Завдяки цьому WebSocket став популярним вибором для чат-додатків, онлайн-ігор, фінансових сервісів та інших застосунків, де затримка є критично важливою.

У порівнянні з WebSocket, протокол HTTP має свої переваги, зокрема простоту використання та універсальність. HTTP є основним протоколом в Інтернеті, тому він підтримується всіма веб-браузерами і серверами. Це робить його зручним для використання в багатьох ситуаціях, особливо там, де не потрібна миттєва реакція. Проте HTTP не призначений для передачі даних у реальному часі, оскільки він вимагає постійних запитів від клієнта до сервера, що може призвести до затримок і збільшення навантаження на сервер.

Server-Sent Events (SSE) — це ще один метод, який дозволяє серверам надсилати дані клієнтам через односпрямоване з'єднання[5]. SSE автоматично підтримує з'єднання, що робить його простим у використанні для передачі оновлень, таких як сповіщення чи новини. Однак, оскільки SSE підтримує лише

односпрямований обмін даними, він не підходить для застосувань, де потрібно двостороннє з’єднання.

MQTT (Message Queuing Telemetry Transport) є легким протоколом обміну повідомленнями, який часто використовується в IoT (інтернеті речей). MQTT дозволяє пристроям з обмеженими ресурсами взаємодіяти з серверами, передаючи невеликі обсяги даних[4]. Як і WebSocket, MQTT підтримує двонаправлену передачу даних, але він є більш спеціалізованим протоколом, що використовується переважно для пристроїв, які працюють в умовах обмеженої пропускної здатності.

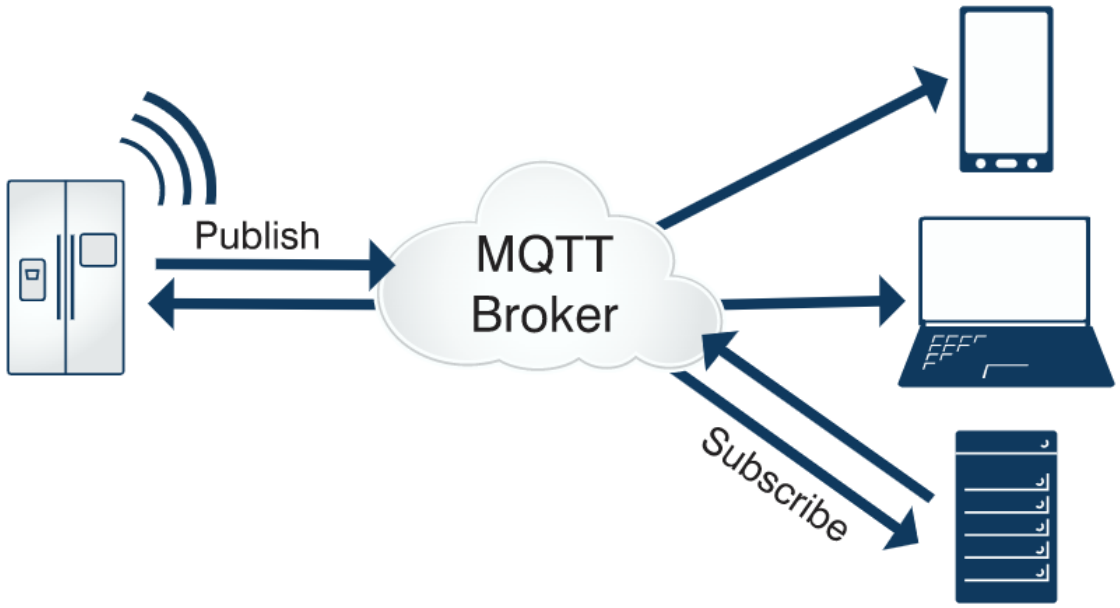


Рисунок 1.4 – Схема MQTT з’єднання

У порівнянні з MQTT, WebSocket більш універсальний і може використовуватися в різних контекстах, тоді як MQTT оптимізовано для з'єднань із малими ресурсами. Протокол MQTT часто використовує модель "публікація-підписка", де клієнти підписуються на канали і отримують дані тільки з тих, на які вони підписані. Це може зменшити навантаження на мережу, але водночас потребує налаштування брокера повідомлень.

Основна ідея MQTT полягає в ефективній і надійній передачі інформації між пристроями навіть при нестабільних підключеннях. Основний принцип роботи MQTT базується на моделі "опубліковано/підписано". Це означає, що пристрої можуть обмінюватися повідомленнями на основі тем, до яких підписуються. Публікатори надсилають інформацію через брокера, а підписники отримують її, лише якщо вони зацікавлені в певній темі. Завдяки цьому підходу, MQTT забезпечує низьку затримку і економить мережеву пропускну здатність, що робить

його ідеальним для використання у сферах, де важливі швидкість і ефективність комунікації.

MQTT широко використовується в різноманітних сферах, включаючи автоматизацію будівель, контроль обладнання, моніторинг ресурсів, безпеку та управління транспортом. Його універсальність дозволяє легко інтегрувати з існуючими системами без складних змін у їхній структурі. Зростаюча популярність IoT також підсилює використання MQTT, оскільки він дозволяє забезпечити безпечну та ефективну комунікацію між пристроями в розподілених системах.

Завдяки своїм перевагам, таким як низька затримка, ефективне використання пропускної здатності та проста реалізація, MQTT продовжує залишатися популярним вибором для багатьох реальних завдань, що вимагають передачі даних у реальному часі.

Оцінюючи переваги і недоліки кожного з цих протоколів, важливо враховувати специфіку застосунку. WebSocket є кращим вибором для застосунків, які потребують миттєвого обміну даними, тоді як HTTP підходить для традиційних веб-додатків, які не потребують високої швидкості передачі. SSE може бути ідеальним для ситуацій, коли потрібно лише односпрямоване оновлення даних, а MQTT — для IoT-пристроїв, які потребують оптимізації трафіку.

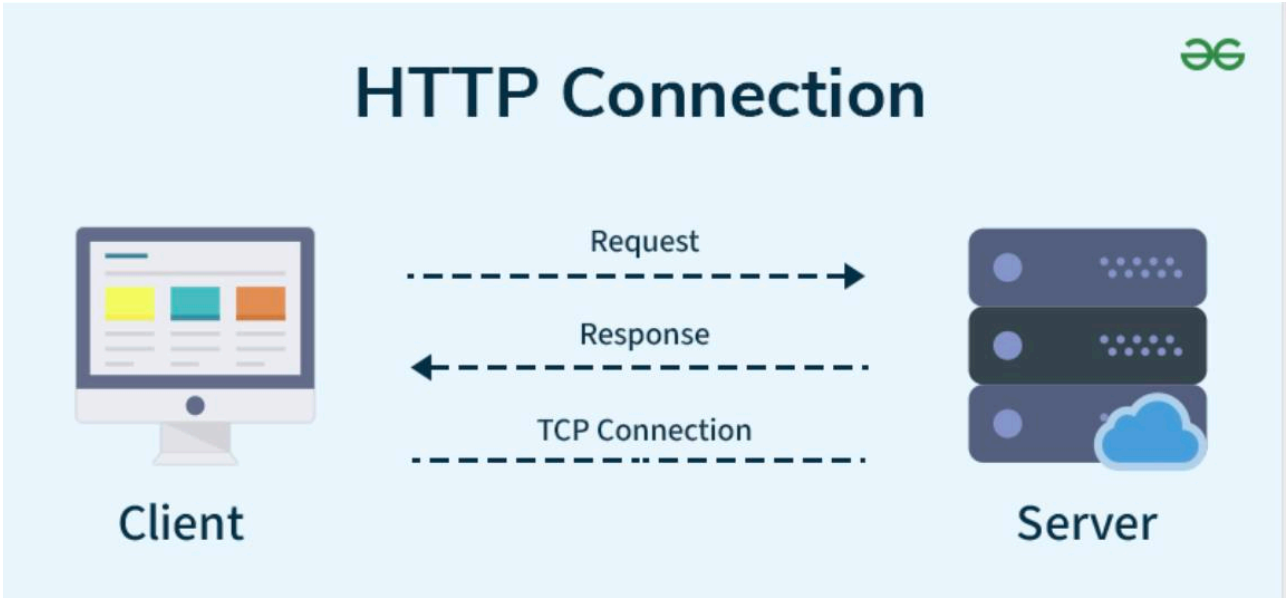


Рисунок 1.5 – Схема HTTP з’єднання

Загалом, правильний вибір протоколу передачі даних у реальному часі може суттєво вплинути на продуктивність, надійність і загальну ефективність веб-додатків. Розуміння особливостей кожного методу допоможе розробникам обрати

оптимальне рішення, яке відповідає вимогам їх проектів. У процесі вибору важливо враховувати не лише швидкість передачі даних, а й обсяги, типи використовуваних даних і специфіку роботи з клієнтами.

У підсумку, WebSocket, HTTP, SSE і MQTT є важливими інструментами для розробки сучасних веб-додатків. Вибір протоколу для передачі даних у реальному часі залежить від багатьох факторів, включаючи тип застосунку, вимоги до швидкості та ресурсів, а також специфіку роботи з клієнтами. Правильний вибір допоможе забезпечити ефективну, швидку та надійну передачу даних, що в свою чергу сприятиме підвищенню задоволеності користувачів і загального успіху проекту.

Таким чином, WebSocket має свої унікальні переваги, особливо для застосунків, що потребують двостороннього обміну даними в реальному часі. Однак для специфічних випадків, таких як IoT, традиційні веб-додатки чи односпрямовані оновлення, інші протоколи можуть виявитися більш ефективними. Знання особливостей кожного методу допоможе розробникам вибрати найбільш підходящий інструмент для своїх потреб, що, безумовно, позитивно вплине на результати їхньої роботи.

1.3 Аналіз архітектурних підходів для розробки серверної частини мобільних застосунків.

У сучасному світі мобільні застосунки стали невід'ємною частиною нашого повсякденного життя. Вони використовуються в багатьох сферах, від соціальних мереж до фінансових послуг. З метою забезпечення стабільної роботи мобільних застосунків важливим є правильний вибір архітектури серверної частини. У цій статті ми розглянемо різні архітектурні підходи, їх переваги та недоліки, а також вплив на продуктивність і надійність мобільних застосунків зображено на рис. 1.5.

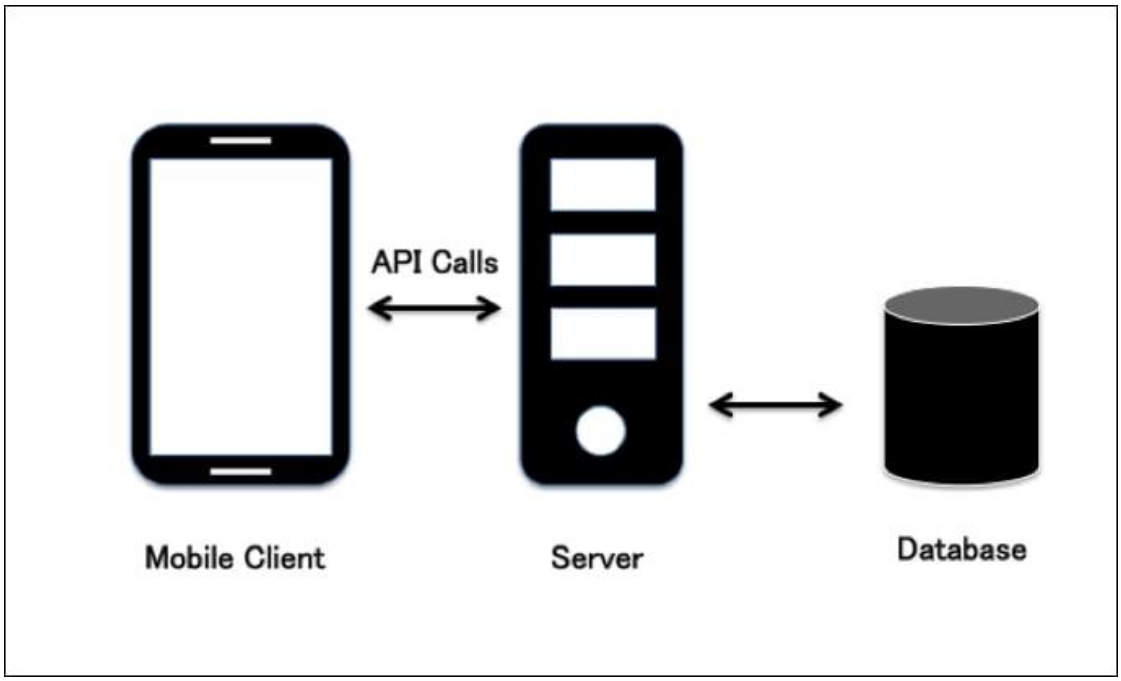


Рисунок 1.6 – Приклад Client-Server архітектури

Перш ніж заглибитися в аналіз, важливо визначити, що таке архітектура серверної частини мобільного застосунку. Архітектура визначає структуру і компоненти, які використовуються для розробки програми. Правильна архітектура допомагає забезпечити безпеку, масштабованість і простоту обслуговування.

Один із найпопулярніших підходів до архітектури серверної частини — це архітектура REST (Representational State Transfer)[6]. REST є архітектурним стилем, що використовує HTTP-протокол для обміну даними між клієнтом і сервером. Однією з основних переваг REST є простота використання, що робить його ідеальним для мобільних застосунків. Клієнт може надсилати запити до сервера, і сервер відповідає з відповідними даними у форматі JSON або XML.

Проте, REST має свої обмеження. По-перше, він працює за принципом запит-відповідь, що може призвести до затримок у передачі даних. Крім того, REST не підтримує постійне з'єднання, що ускладнює реалізацію функцій, які вимагають миттєвого оновлення, таких як чати або реальний моніторинг.

З метою подолання цих обмежень з'явилися інші архітектурні підходи, такі як GraphQL. GraphQL — це мова запитів для API, яка дозволяє клієнтам запитувати лише необхідні дані, що знижує обсяги переданих даних і підвищує ефективність. Клієнт може формулювати запити так, щоб отримувати точно те, що потрібно, без надмірних запитів.

Крім REST і GraphQL, іншим підходом є використання WebSocket, який забезпечує двосторонній зв'язок між клієнтом і сервером. WebSocket дозволяє серверу надсилати дані до клієнта у реальному часі без необхідності повторних запитів[7]. Це робить його ідеальним для застосунків, які вимагають миттєвих оновлень.

Вибір архітектури також залежить від типу мобільного застосунку. Наприклад, для застосунків, що вимагають високої продуктивності та швидкого обміну даними, таких як ігри чи фінансові платформи, доцільно використовувати WebSocket або GraphQL зображено на рис. 1.7. Водночас для менш вимогливих застосунків, таких як новинні платформи, підійде REST.

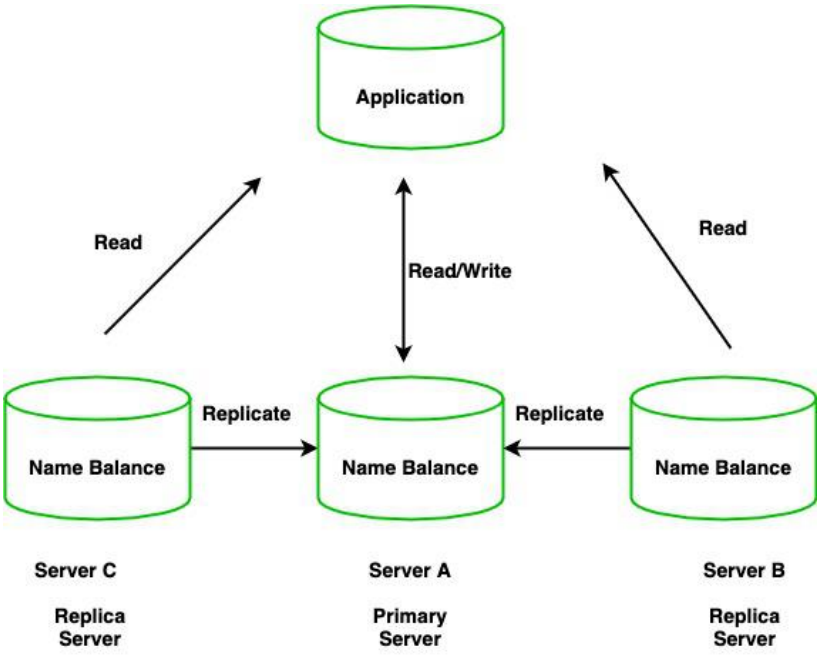


Рисунок 1.7 – Приклад розподілу даних у базах даних для масштабування

Крім того, важливо враховувати масштабованість архітектури. Мобільні застосунки можуть мати різну кількість користувачів, і серверна архітектура повинна бути готовою до зростання навантаження. Сучасні рішення, такі як мікросервісна архітектура, дозволяють розділити сервер на окремі модулі, що спрощує масштабування та обслуговування.

Мікросервісна архітектура надає можливість розподілити навантаження між кількома серверами, що підвищує загальну продуктивність системи. Кожен мікросервіс виконує певну функцію і може бути розгорнутий окремо, що полегшує підтримку і оновлення системи. Наприклад, у фінансовому застосунку окремі мікросервіси можуть відповідати за обробку платежів, управління користувачами та звітність.

Іншою важливою архітектурною парадигмою є сервер без серверів (Serverless). Цей підхід передбачає використання хмарних провайдерів, які автоматично обробляють і масштабують серверні ресурси. Розробники можуть зосередитися на написанні коду, а не на управлінні інфраструктурою. Це особливо корисно для застосунків з нерегулярним навантаженням, оскільки дозволяє знизити витрати на інфраструктуру.

Крім того, слід звернути увагу на безпеку серверної архітектури. Мобільні застосунки часто працюють з конфіденційними даними, такими як особисті дані користувачів або фінансова інформація. Архітектура повинна бути спроектована так, щоб забезпечити належний рівень безпеки, включаючи аутентифікацію та шифрування даних.

Вибір правильної архітектури серверної частини мобільного застосунку залежить від багатьох чинників, таких як вимоги до продуктивності, тип оброблюваних даних, кількість користувачів та бюджет. Важливо також враховувати, що жоден з підходів не є універсальним. Вони повинні бути адаптовані до конкретних потреб проекту.

Зважаючи на всі ці фактори, розробники повинні оцінювати і тестувати різні архітектурні рішення, щоб знайти оптимальне для свого проекту[8]. Це дозволить забезпечити високу продуктивність, надійність і безпеку мобільних застосунків.

У підсумку, архітектура серверної частини мобільного застосунку відіграє ключову роль у його функціонуванні. Правильний вибір архітектурного підходу

може суттєво вплинути на продуктивність, масштабованість і безпеку програми. REST, GraphQL, WebSocket, мікросервісна та серверна архітектура без серверів є лише кількома з можливих рішень. Розробники повинні ретельно аналізувати свої потреби і обирати найбільш підходящу архітектуру для реалізації своїх ідей.

1.4 Вибір стеку технологій для реалізації серверної архітектури.

Вибір стеку технологій для розробки серверної архітектури мобільного застосунку є одним із найважливіших аспектів, що впливає на продуктивність, стабільність і масштабованість системи. Технологічний стек визначає набір інструментів і протоколів для обробки даних, підтримки зв’язку в реальному часі, управління навантаженнями та інтеграції з іншими сервісами. Обраний стек також впливає на загальну вартість розробки та підтримки, можливість подальшого розширення системи та швидкість випуску нових функцій.

Для серверної частини мобільних застосунків широко використовуються різні мови програмування та фреймворки. Node.js популярний завдяки своїй здатності обробляти асинхронні запити та підтримувати двостороннє спілкування через WebSocket[9]. Його перевагою є низька затримка та висока продуктивність для реального часу. Python із фреймворками Django або Flask забезпечує простоту реалізації REST API і підходить для швидкого прототипування. Java разом зі Spring Boot є вибором для корпоративних рішень завдяки високій безпеці та надійності. Go (Golang) показує відмінні результати при створенні високонавантажених систем, забезпечуючи швидку обробку запитів і низьке використання ресурсів.

Важливим елементом вибору є протоколи комунікації. WebSocket дозволяє підтримувати стабільне двостороннє з'єднання між клієнтом і сервером з мінімальними затримками. HTTP/2 та HTTP/3 забезпечують швидку передачу запитів завдяки мультиплексуванню та оптимізації мережевого з'єднання. SSE (Server-Sent Events) підходить для односторонніх потокових оновлень, наприклад, для стрічок новин. MQTT часто використовується в IoT-проектах, де важливим є економне використання пропускної здатності мережі та швидка передача невеликих пакетів даних.

Окрім вибору мови програмування та протоколу, важливо правильно підібрати базу даних та механізми кешування. Реляційні бази даних, такі як PostgreSQL чи MySQL, використовуються для роботи зі складними структурованими даними, тоді як NoSQL рішення, як-от MongoDB, забезпечують гнучкість і масштабованість для неструктурованих даних. Redis та Memcached

застосовуються для кешування, що знижує навантаження на базу даних і підвищує швидкість обробки запитів.

Важливим компонентом є вибір хмарної платформи для розгортання серверної частини. AWS (Amazon Web Services) пропонує повний набір інструментів для створення масштабованих і продуктивних систем[10]. Google Cloud Platform (GCP) надає розширені можливості для аналітики даних і машинного навчання. Microsoft Azure приваблює підприємства завдяки тісній інтеграції з продуктами Microsoft, такими як Active Directory і Office 365. Хмарні рішення також надають можливість автоматичного масштабування сервісів у відповідь на зростання навантаження.

Розгортання серверної частини мобільного застосунку потребує ефективного балансування навантаження. NGINX і HAProxy є популярними рішеннями для розподілу трафіку між кількома серверами, що запобігає перевантаженням і забезпечує стабільність роботи. Kubernetes є інструментом для управління контейнерами та автоматичного масштабування, що дозволяє додавати або зменшувати кількість інстансів сервісу залежно від поточного навантаження. Це особливо корисно для застосунків з непередбачуваними піками активності користувачів.

Вибір правильного стеку технологій також залежить від специфічних вимог застосунку. Наприклад, для месенджерів і чатів, де критичною є мінімальна затримка при передачі повідомлень, WebSocket є найкращим варіантом. Для застосунків, що потребують регулярних оновлень інформації, наприклад, стрічки новин або онлайн-таблиці, може підійти SSE. Якщо ж застосунок працює з великою кількістю IoT-пристроїв, доцільним буде використання MQTT завдяки його легкості та ефективності в умовах обмежених ресурсів.

Інтеграція всіх компонентів стеку технологій має забезпечувати легку взаємодію між сервером та клієнтом, а також можливість безперебійного оновлення застосунку[11]. Використання мікросервісної архітектури може значно спростити цей процес, оскільки кожен сервіс виконує окрему функцію і може бути незалежно масштабований. Крім того, мікросервіси легко інтегруються з зовнішніми API, що розширює можливості застосунку.

При розробці серверної частини мобільного застосунку також важливо врахувати питання безпеки. Шифрування даних і використання безпечних протоколів, таких як HTTPS і WSS, є обов’язковими для захисту особистих даних користувачів. Аутентифікація та авторизація користувачів можуть бути реалізовані за допомогою OAuth 2.0 або JWT (JSON Web Token), що забезпечує контроль доступу до ресурсу.

Загалом, вибір стеку технологій для розробки серверної архітектури є багатофакторним процесом, який потребує ретельного аналізу вимог проєкту, навантаження та бюджету. Комбінація правильних інструментів дозволить створити продуктивну, масштабовану та надійну систему, що забезпечить стабільну роботу мобільного застосунку навіть під час пікових навантажень.

1.5 Постановка вимог до системи та вибір інструментів розробки

Для розробки ефективної серверної частини мобільного застосунку критично важливо правильно сформулювати вимоги до системи та обрати відповідні інструменти. Постановка вимог визначає функціональні та нефункціональні параметри, від яких залежить загальна архітектура проєкту, а вибір інструментів безпосередньо впливає на швидкість реалізації, продуктивність і можливість подальшого масштабування системи.

Першим етапом є визначення ключових функціональних вимог, які включають підтримку стабільної двосторонньої комунікації, обробку реальних даних у режимі реального часу та інтеграцію з базами даних і зовнішніми сервісами. Крім того, система має забезпечувати високу доступність, що передбачає розробку механізмів резервування та балансування навантаження. Важливим аспектом є також підтримка масштабованості, щоб система могла розширюватися у відповідь на збільшення числа користувачів чи навантаження.

Нефункціональні вимоги включають безпеку даних, високу швидкодію, мінімальні затримки під час обміну повідомленнями та легкість інтеграції з іншими сервісами. Система повинна відповідати сучасним стандартам безпеки, таким як шифрування з’єднань через HTTPS та WSS[12]. Швидкість обробки запитів і низька затримка передачі даних критично важливі для забезпечення безперервної взаємодії користувачів у реальному часі.

Обираючи інструменти розробки, потрібно зважати на специфіку застосунку та поставлені вимоги. Node.js є оптимальним вибором для побудови серверів з асинхронною обробкою запитів і підтримкою WebSocket. Java із фреймворком Spring Boot забезпечує високу стабільність і безпеку для корпоративних рішень. Go (Golang) відомий своєю продуктивністю і підходить для високонавантажених систем. Для прототипування та невеликих проєктів популярними є Python з Flask або Django.

Для зберігання даних можна використовувати як реляційні бази, наприклад, PostgreSQL чи MySQL, так і NoSQL бази, як-от MongoDB. Redis рекомендується для кешування, що дозволяє пришвидшити доступ до часто використовуваних

даних. При виборі бази важливо враховувати вимоги до структури даних та очікувані обсяги.

Балансування навантаження можна реалізувати за допомогою NGINX або HAProxy, що забезпечують розподіл запитів між кількома серверами. Для управління контейнерами та автоматичного масштабування часто використовують Kubernetes, який дозволяє оптимально керувати ресурсами та підтримувати високу доступність.

Хмарні сервіси, такі як AWS, Google Cloud або Microsoft Azure, надають широкий набір інструментів для створення інфраструктури та автоматичного масштабування[13]. Використання хмарних технологій також забезпечує резервування даних і безперервність роботи сервісу у разі збоїв.

Під час проєктування важливо передбачити механізми аутентифікації та авторизації користувачів. Найчастіше використовують OAuth 2.0 або JWT для безпечного керування доступом до ресурсів. Це дозволяє уникнути несанкціонованого доступу та забезпечити захист особистих даних.

РОЗДІЛ 2 ПРОЕКТУВАННЯ СЕРВЕРНОЇ АРХІТЕКТУРИ З ВИКОРИСТАННЯМ WEBSOCET

2.1 Загальна схема архітектури мобільного застосунку.

Архітектура мобільного застосунку — це структурований підхід до розробки програмного забезпечення, який забезпечує ефективність, масштабованість, продуктивність та безперебійну роботу[14]. Успіх мобільного застосунку залежить від вибору підходу до проєктування його архітектури, де кожен компонент виконує визначені функції та працює узгоджено з іншими.

Основні архітектурні компоненти можна поділити на декілька рівнів: клієнтський, серверний, бази даних, балансування навантаження, а також інтеграція з хмарною інфраструктурою для масштабування. Ця структура забезпечує надійне обслуговування користувачів навіть при високому навантаженні.

Клієнтський рівень представлений мобільними застосунками на платформах iOS та Android, які отримують та відправляють дані на сервер. Програмний код клієнта виконує обробку інтерфейсу користувача, валидацію вводу даних і відображення результатів. Клієнт взаємодіє з бекендом через REST API або WebSocket, що дає змогу забезпечити реальний час передачі повідомлень.

Бекенд включає основний сервер застосунку, який обробляє запити клієнтів. Через API Gateway здійснюється управління потоками запитів та контроль доступу. Основні завдання бекенду — аутентифікація користувачів, управління даними, обробка транзакцій та логіка бізнес-процесів.

Бази даних можуть включати SQL та NoSQL рішення, що дозволяє поєднувати структуровані та неструктуровані дані. PostgreSQL і MongoDB є популярними прикладами таких баз, які підтримують зберігання даних у різних форматах[15].

Далі, система балансування навантаження, наприклад, за допомогою NGINX або HAProxy, запобігає перевантаженню серверів і забезпечує рівномірний розподіл запитів. Балансування гарантує, що користувачі отримують швидкий доступ до застосунку, незалежно від географічного положення або часу доби.

Інтеграція з хмарною інфраструктурою, як AWS або Google Cloud, дозволяє забезпечити масштабованість застосунку та резервування даних[16]. Такі хмарні платформи пропонують автоматичне масштабування, що дозволяє збільшувати або зменшувати кількість серверів у відповідь на зміни навантаження.

Важливою частиною є також реалізація безперервної комунікації за допомогою WebSocket. Цей протокол дозволяє встановити постійне з'єднання між клієнтом та сервером, що забезпечує миттєве передавання повідомлень без необхідності повторних запитів. Це критично важливо для застосунків із реальним часом, таких як месенджери або торгові платформи.

REST API (Representational State Transfer) – це стиль архітектури для створення веб-сервісів, який базується на стандартних протоколах HTTP для обміну даними між клієнтом і сервером. REST API використовує чотири основні методи HTTP (GET, POST, PUT, DELETE) для виконання операцій над ресурсами. Ресурси представляються у вигляді унікальних URL-адрес, що дозволяє їх легко розпізнавати і ідентифікувати. Кожен ресурс має унікальний URI, який клієнт використовує для доступу до даних, наприклад, представляє користувача з ID

Основна ідея REST API полягає в безстанності і легкості масштабування. Безстанність означає, що сервер не зберігає стан між запитами. Кожен HTTP-запит містить всі необхідні дані для виконання дії, і сервер не зберігає інформацію про попередні запити. Це дозволяє легко масштабувати систему, додавати нові сервери або модулі, не змінюючи при цьому логіку обробки запитів. REST API дозволяє клієнту робити запити, що не залежать від попередніх дій, що сприяє простоті інтеграції з іншими сервісами і додатками.

Використання REST API вимагає відповідного налаштування серверів і клієнтів для підтримки стандартів і методів взаємодії. Сервер обробляє запити, надаючи дані у форматі JSON або XML, що робить API сумісним з різними мовами програмування і платформами. Взаємодія з ресурсами здійснюється через прості запити HTTP, що підвищує зручність використання та інтеграцію з сучасними додатками.

Безпека є важливим аспектом REST API. Використання HTTPS забезпечує шифрування даних під час передачі, що дозволяє захистити інформацію від перехоплення. Також використовуються механізми аутентифікації та авторизації

для перевірки клієнтів і управління доступом до ресурсів. Одна з поширених методів аутентифікації — використання токенів, наприклад, OAuth, які передаються через заголовок запиту. Це дозволяє контролювати, які дії може виконувати клієнт на сервері, запобігаючи несанкціонованому доступу до ресурсів.

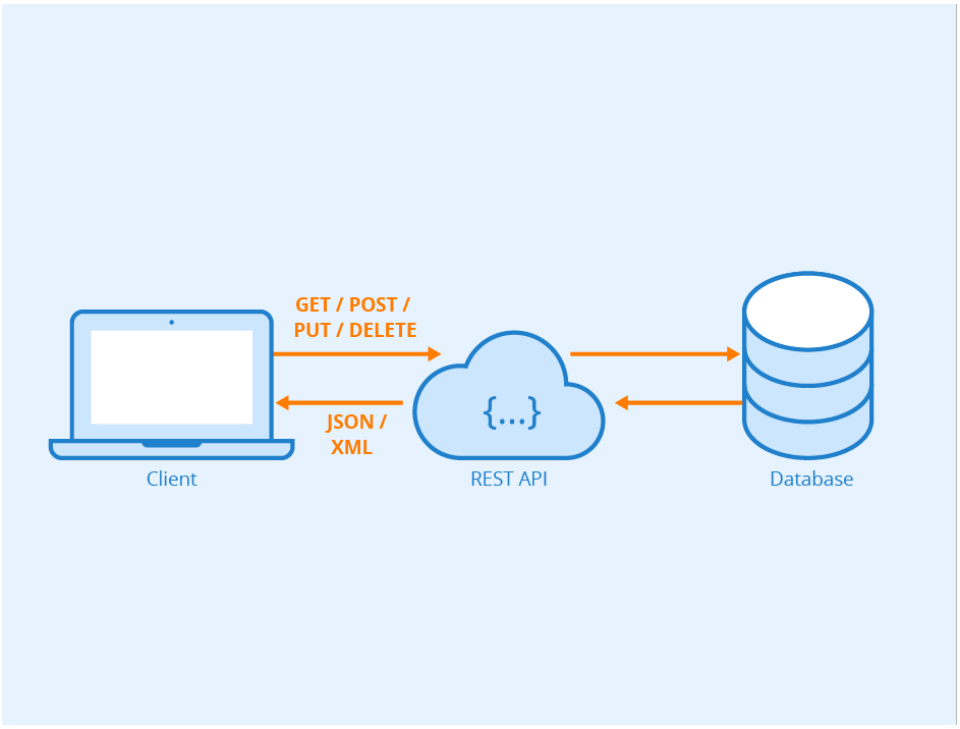


Рисунок 2.1 – Схема REST

REST API також підтримує зв’язки між ресурсами, що дозволяє створювати логічні взаємодії між різними об’єктами даних. Наприклад, замовлення можуть бути пов’язані з продуктами, що дозволяє зберігати і передавати інформацію про контекст у взаємодії. Це підвищує гнучкість і ефективність API, дозволяючи створювати більш складні і інтегровані системи.

Завдяки своїй простоті, гнучкості і можливостям масштабування, REST API став основним вибором для багатьох сучасних веб-сервісів. Це дозволяє швидко і легко інтегрувати додатки, що взаємодіють через інтернет, незалежно від використовуваного клієнта або платформи. REST API забезпечує надійну і безпечну передачу даних, що критично важливо для багатьох сучасних додатків, таких як мобільні застосунки, системи моніторингу, e-commerce рішення, і IoT-проекти.

REST API часто використовується в багатьох типах застосунків, від простих веб-додатків до комплексних систем управління даними, мобільних додатків та IoT

рішень. Він зручний для побудови систем на основі мікросервісів, де окремі модулі можуть легко інтегруватися та взаємодіяти через REST API.

Завдяки своїй гнучкості і зручності використання, REST API став основним архітектурним стилем для багатьох сучасних сервісів і додатків, забезпечуючи ефективну і безпечну взаємодію між клієнтами і серверами.

2.2 Розробка моделі передачі даних між клієнтом і сервером

Передача даних між клієнтом і сервером є фундаментальною частиною архітектури сучасних мобільних застосунків. Коректно спроєктована модель обміну даними забезпечує швидкість, стабільність і безпеку взаємодії, що критично важливо для роботи в реальному часі. У цьому розділі ми детально розглянемо основні аспекти розробки моделі передачі даних, методи, протоколи та виклики, що виникають під час реалізації цього процесу[17].

Головною метою будь-якої архітектури передачі даних є створення ефективного каналу, через який інформація передаватиметься надійно, з мінімальними затримками та втратами. Для цього важливо врахувати наступні аспекти:

- швидкість та продуктивність — зменшення часу відгуку на запити клієнта;
- стабільність з’єднання — підтримка зв'язку навіть при нестабільних мережах;
- безпека даних — шифрування даних для захисту від несанкціонованого доступу;
- масштабованість — можливість обробки великої кількості запитів від одночасних користувачів.

Для реалізації цих цілей використовується низка протоколів і методів передачі даних, кожен із яких має свої переваги й обмеження. Серед них популярними є REST, WebSocket, MQTT, і GraphQL.

REST API є найпоширенішим способом обміну даними між клієнтом і сервером. Він базується на HTTP-запитах, які передаються у вигляді тексту (JSON або XML). REST забезпечує простоту реалізації, однак він не завжди підходить для комунікації в реальному часі, оскільки потребує повторних запитів для оновлення даних.

WebSocket дозволяє встановлювати постійне з’єднання між клієнтом і сервером, що значно зменшує затримки в порівнянні з REST. Завдяки цьому

WebSocket підходить для додатків, де важлива миттєва передача даних, наприклад, для чатів або біржових платформ.

MQTT використовується для передачі повідомлень у форматі publish/subscribe. Цей протокол особливо популярний для IoT-додатків, де потрібна економія трафіку та надійність у нестабільних мережах.

При розробці моделі передачі даних необхідно пройти кілька важливих етапів:

- аналіз вимог: визначення сценаріїв взаємодії, обсягу переданих даних і вимог до швидкості з’єднання;
- вибір протоколу: залежно від специфіки застосунку обирається WebSocket, REST API або інший протокол;
- реалізація серверної та клієнтської логіки: налаштування серверної частини для обробки запитів і відповідей;
- обробка помилок та відновлення з’єднання: забезпечення стабільної роботи в умовах мережевих збоїв.

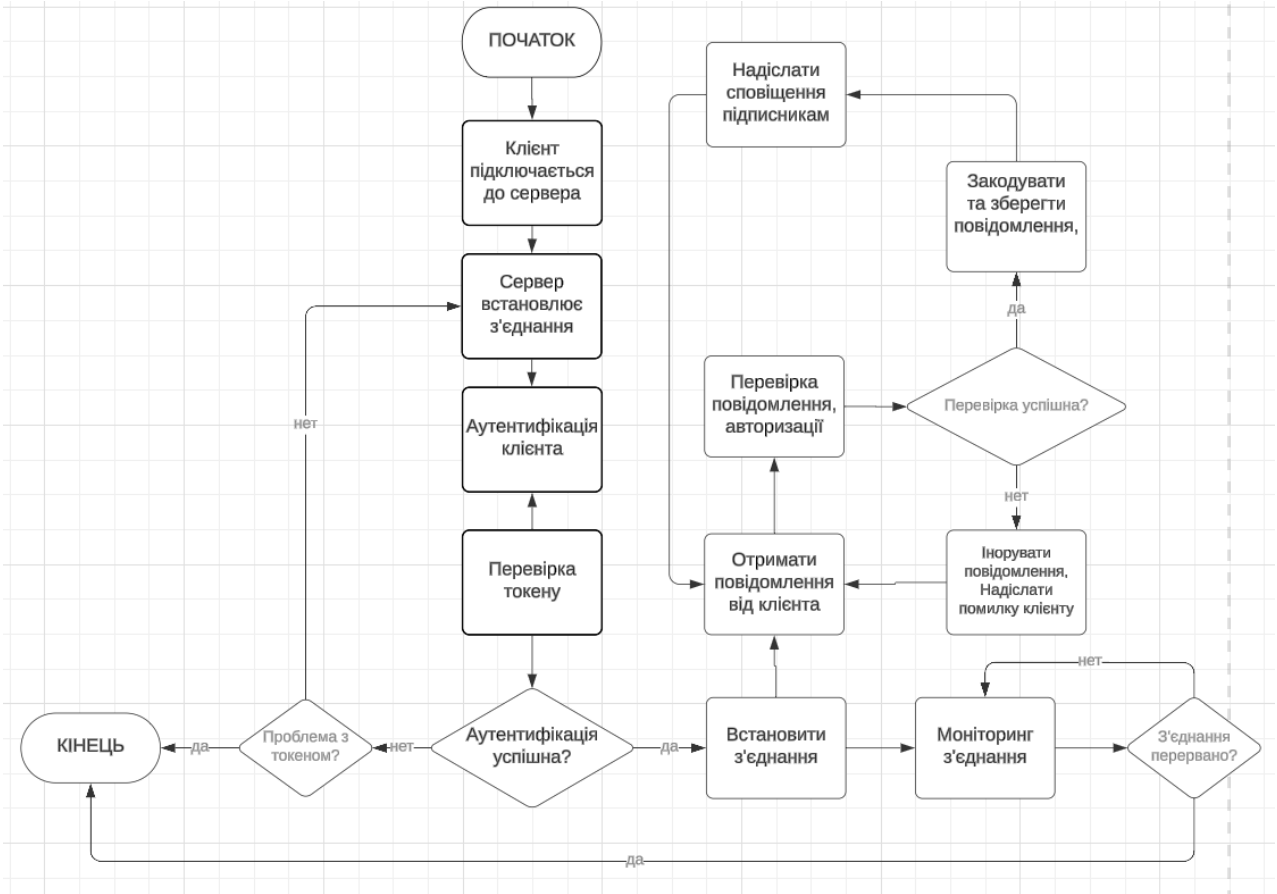


Рисунок 2.2 – Блок-схема спілкування клієнта з сервером

Для того щоб забезпечити стабільну та швидку передачу даних, необхідно обрати правильні архітектурні рішення. Одним із таких рішень є використання API Gateway для управління запитами зображено на рис. 2.1. Gateway дозволяє централізовано обробляти трафік і перенаправляти його до відповідних сервісів. Це зменшує навантаження на основний сервер і підвищує продуктивність системи.

Іншою важливою частиною є балансування навантаження, яке гарантує рівномірний розподіл запитів між серверами. Завдяки цьому система може обробляти великий обсяг запитів одночасно без збоїв.

Реальна передача даних має низку переваг, зокрема можливість миттєвого оновлення інформації та інтерактивну взаємодію з користувачами. Проте вона також супроводжується певними викликами:

Високі вимоги до інфраструктури: необхідність підтримки стабільного з’єднання для великої кількості користувачів.

Мережеві затримки: нестабільні мережі можуть призвести до втрати даних або затримок.

Безпека: реальний час передачі потребує ефективних методів шифрування для захисту даних.

Для підвищення масштабованості та надійності системи багато компаній інтегрують свої архітектури з хмарними платформами, такими як AWS, Azure або Google Cloud. Використання хмарних сервісів дозволяє автоматично масштабувати ресурси та забезпечувати безперебійне обслуговування користувачів навіть під час пікових навантажень.

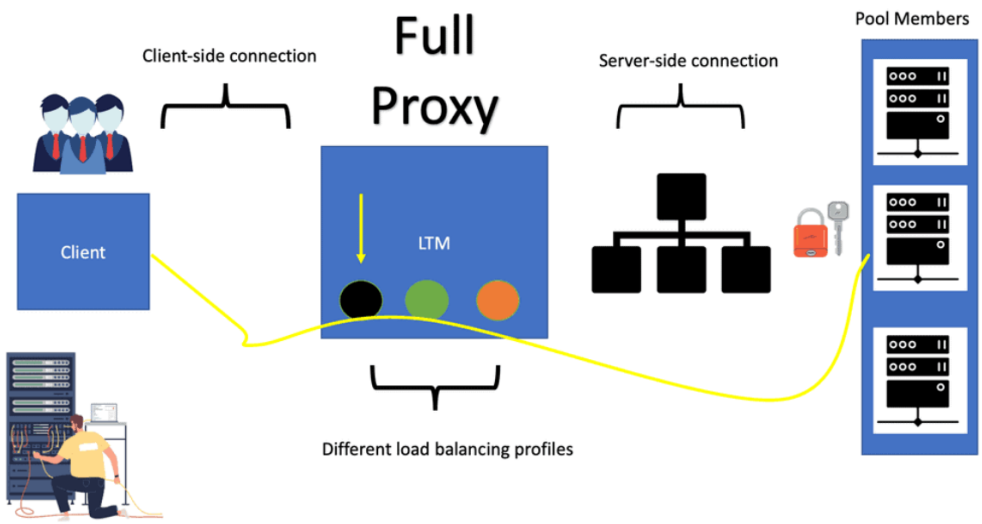


Рисунок 2.3 – Балансування навантаження для архітектури зв'язку в реальному часі

Розробка моделі передачі даних між клієнтом і сервером є критичним завданням для мобільних застосунків, що працюють у реальному часі. Обираючи відповідний протокол і архітектуру, розробники можуть забезпечити стабільну та швидку взаємодію користувачів із системою[18]. Правильно реалізована модель обміну даними дозволяє уникнути затримок, оптимізувати використання ресурсів і гарантувати безпеку передачі інформації. У сучасних умовах важливо також використовувати хмарні сервіси та балансування навантаження для підтримки високої продуктивності системи.

2.3 Організація підключень та управління сесіями WebSocket

Передача даних у реальному часі є важливим аспектом для багатьох сучасних мобільних застосунків, таких як месенджери, торговельні платформи, системи моніторингу та IoT-рішення[20]. Використання WebSocket забезпечує низьку затримку та двосторонню комунікацію між клієнтом і сервером, що робить його ідеальним вибором для реального часу. Проте правильна організація підключень і управління сесіями відіграє ключову роль для стабільної роботи таких систем.

Цей розділ присвячено детальному аналізу процесів підключення через WebSocket, створення сесій, обробки подій з’єднання та відновлення після розривів. Окрім того, розглядаються методи аутентифікації, управління станом сесій і безпека комунікації.

WebSocket є протоколом зв’язку, що дозволяє встановити постійне двостороннє з’єднання між клієнтом і сервером. Це дозволяє обмінюватися повідомленнями в реальному часі без необхідності багаторазових HTTP-запитів. Після початкового "рукостискання" на основі HTTP WebSocket відкриває новий канал зв’язку, через який клієнт і сервер можуть надсилати дані один одному в будь-який момент[19].

Процес встановлення підключення починається з HTTP-запиту від клієнта на ініціалізацію WebSocket-з’єднання. Сервер відповідає підтвердженням, після чого звичайний HTTP-трафік змінюється на двосторонній канал WebSocket. Для успішного підключення важливо, щоб сервер підтримував WebSocket-протокол і був налаштований на прийняття таких запитів.

Сесія в контексті WebSocket — це період часу, протягом якого клієнт підтримує активне з’єднання із сервером[21]. Сесії використовуються для зберігання інформації про стан користувача, обробку подій та управління ресурсами. Для коректної роботи з сесіями необхідно розв'язати такі задачі:

- Аутентифікація користувача при відкритті з’єднання.
- Відновлення сесії після розриву зв’язку.
- Закриття неактивних сесій для економії ресурсів сервера.
- Управління таймаутами, щоб уникнути завислих з'єднань.

Аутентифікація зазвичай виконується на етапі рукостискання HTTP, під час якого клієнт передає токен доступу або інший ідентифікатор користувача. Сервер перевіряє цей токен і приймає рішення про дозвіл або відхилення з’єднання. Після успішної аутентифікації інформація про сесію зберігається в пам'яті сервера або в базі даних.

Мобільні застосунки часто стикаються з мережевими проблемами через втрату сигналу або зміну мережі (Wi-Fi на мобільний інтернет і навпаки). У таких випадках важливо, щоб клієнт міг автоматично відновити з'єднання без втрати даних.

Одним із підходів до відновлення сесії є використання heartbeat-повідомлень, які періодично надсилаються між клієнтом і сервером для перевірки активності з'єднання. Якщо одне зі сторін не отримує heartbeat у визначений проміжок часу, з’єднання вважається розірваним і ініціюється спроба його відновлення.

Забезпечення безпеки передачі даних є важливим аспектом під час роботи з WebSocket. На відміну від звичайних HTTP-запитів, WebSocket може залишатися відкритим протягом тривалого часу, що створює додаткові ризики для безпеки. Основні загрози включають:

- перехоплення даних (man-in-the-middle атаки);
- злом сесій або викрадення ідентифікаторів;
- DDoS-атаки, що можуть перевантажити сервер.

Для захисту WebSocket-з’єднань використовують наступні підходи:

- використання WSS (WebSocket Secure), що шифрує дані за допомогою TLS;
- аутентифікація та авторизація при підключенні;
- обмеження кількості одночасних з'єднань із одного клієнта;

Для забезпечення стабільної роботи застосунку при великій кількості користувачів використовуються методи балансування навантаження. Балансувальник перенаправляє трафік між кількома серверами, що дозволяє розподілити навантаження рівномірно та уникнути перевантажень.

Зростання кількості одночасних користувачів є ще одним викликом. У таких випадках необхідно застосовувати балансування навантаження. Балансувальники трафіку забезпечують рівномірний розподіл запитів між декількома серверами,

підтримуючи їхню продуктивність навіть під час пікових навантажень. У випадку WebSocket це особливо важливо, оскільки балансувальник має зберігати прив’язку до одного сервера для кожної сесії.

При використанні WebSocket балансування має свої особливості. Зокрема, з’єднання повинні підтримуватися протягом тривалого часу, а балансувальник має вміти перенаправляти трафік до одного й того ж сервера для кожної сесії.

Нестабільність мережі, як-от втрата сигналу або перемикання між Wi-Fi і мобільними даними, також вимагає від системи додаткових механізмів. Зокрема, клієнт має автоматично відновлювати з’єднання без втрати даних або функціональності. Це досягається через використання збережених сесій і повторної аутентифікації після розриву. У таких сценаріях важливо мінімізувати час відновлення для забезпечення комфортного користувацького досвіду.

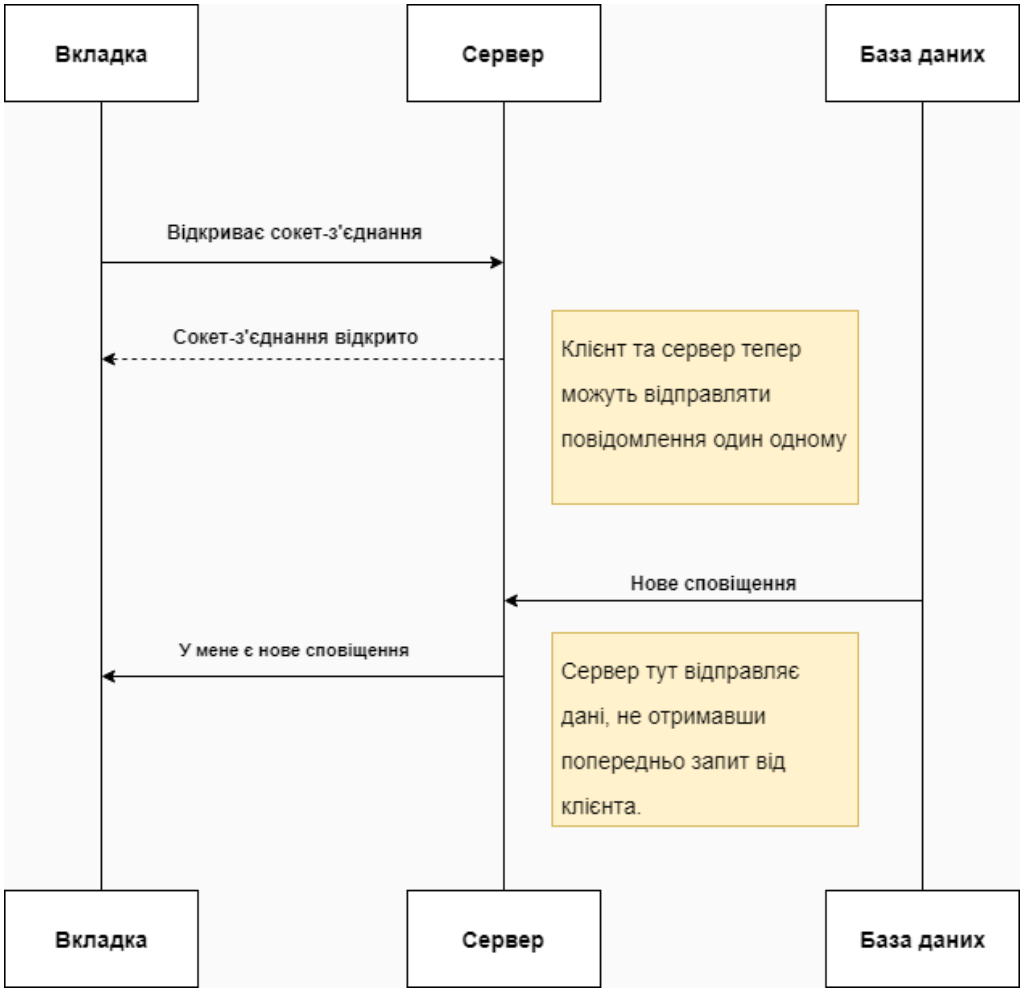


Рисунок 2.4 – Рисунок балансування WebSocket навантажень

Розробка моделі передачі даних через WebSocket передбачає врахування багатьох аспектів, таких як організація підключень, управління сесіями,

забезпечення безпеки та відновлення з'єднання. Коректно реалізована модель передачі даних забезпечує швидку та надійну комунікацію в реальному часі, що критично важливо для мобільних застосунків. Балансування навантаження та масштабування системи дозволяє підтримувати високу продуктивність навіть при значному зростанні кількості користувачів.

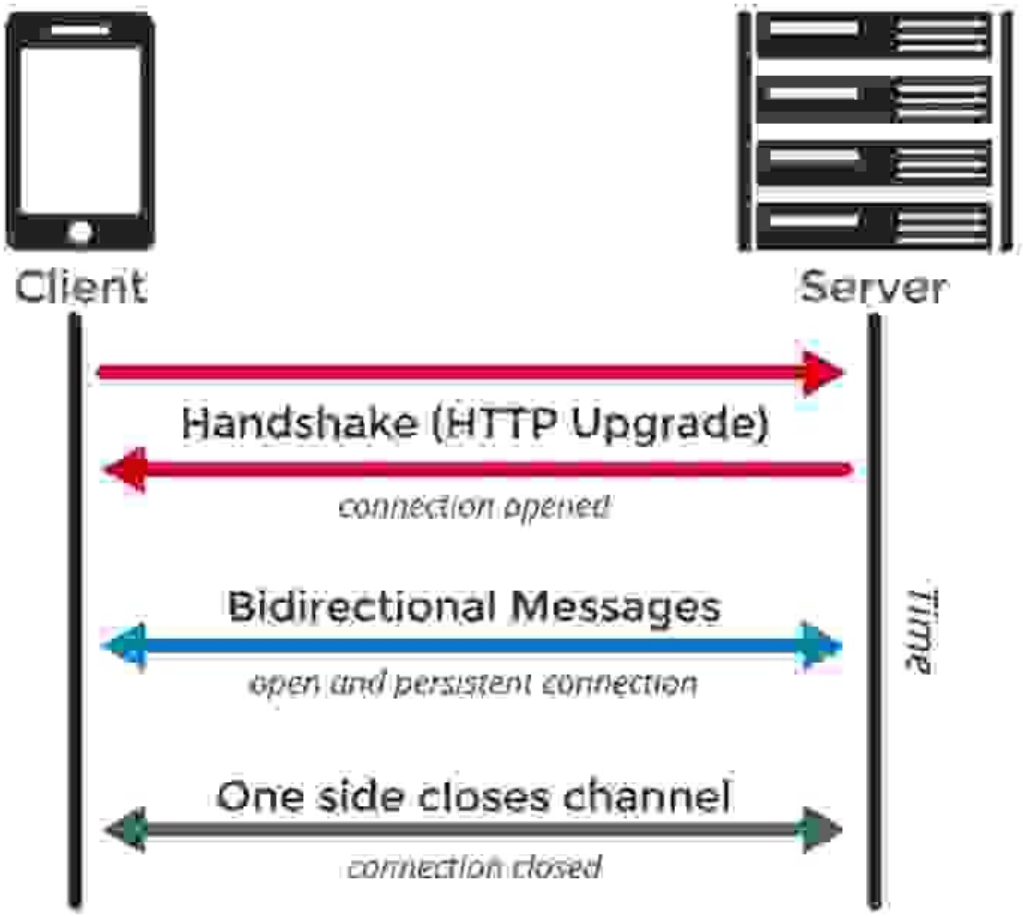


Рисунок 2.5 – Сесія у контексті WebSocket

Сесія у контексті WebSocket — це активне підключення клієнта до сервера, яке може тривати від кількох секунд до кількох годин або навіть днів. Сесії використовуються для передачі даних про користувача, обробки подій і управління його діями. Для правильної роботи з сесіями важливо впроваджувати механізми аутентифікації, перевірки стану підключення та обробки розривів зв’язку. Наприклад, клієнт може періодично надсилати сигнали до сервера для перевірки, чи активне з’єднання. Якщо відповідь не отримується протягом заданого часу, клієнт автоматично намагається відновити підключення.

Окремим викликом є безпека. Оскільки WebSocket-з’єднання залишається відкритим тривалий час, це створює додаткові ризики, такі як перехоплення даних, крадіжка сесій або DDoS-атаки. Для їхньої мінімізації застосовуються протоколи шифрування (WSS), перевірка аутентифікації користувачів і обмеження кількості підключень від одного клієнта. Використання сертифікатів SSL/TLS гарантує, що дані будуть захищені від несанкціонованого доступу.

Для успішного впровадження WebSocket потрібно враховувати багато технічних аспектів, зокрема масштабованість архітектури, методи моніторингу стану підключень, обробку помилок і резервування ресурсів. Такий підхід дозволяє створити швидкий, надійний і безпечний сервіс, здатний підтримувати стабільну роботу навіть під час значного зростання кількості користувачів.

Основна перевага WebSocket полягає в тому, що після початкового запиту клієнт і сервер переходять на постійний канал зв’язку, який дозволяє уникати численних HTTP-запитів. У цьому випадку зменшується кількість обчислювальних операцій на обох сторонах, що позитивно впливає на продуктивність. Однак така архітектура вимагає правильного управління сесіями для забезпечення тривалого й стабільного підключення.

2.4 Проблеми безпеки та механізми захисту

Безпека — одна з найважливіших складових при розробці серверної архітектури мобільних застосунків, особливо якщо вони працюють у режимі реального часу та передають чутливі дані. Користувачі очікують, що їхні персональні дані будуть захищені, а взаємодія між клієнтом і сервером — безпечною та стабільною. Проте, існує безліч ризиків, які можуть загрожувати безпеці передачі даних. У цьому розділі аналізуються основні проблеми безпеки та пропонуються ефективні механізми захисту для мобільних застосунків.

У процесі розробки серверних систем із використанням WebSocket протоколу особливу увагу слід приділяти безпеці, адже постійне з’єднання між клієнтом і сервером створює потенційні вразливості. WebSocket, хоч і надає ефективний засіб двостороннього зв’язку, має ряд специфічних загроз, які потребують відповідних заходів захисту.

Однією з ключових проблем є ризик перехоплення даних (атаки типу "man-in-the-middle"), коли зловмисник отримує доступ до інформації, що передається між клієнтом і сервером. Ця проблема особливо критична для конфіденційних даних, таких як паролі, токени доступу чи фінансова інформація. Для вирішення цієї проблеми використовується протокол WSS (WebSocket Secure), який забезпечує шифрування трафіку за допомогою TLS (Transport Layer Security). Це дозволяє гарантувати, що дані передаються захищеним каналом, і виключає можливість їхнього перехоплення у відкритому вигляді.

Рисунок 2.6 – Методи і технології для забезпечення безпеки розподіленої комп'ютерної системи

Ще одним викликом є крадіжка ідентифікаторів сесій, коли зловмисник може використати викрадені дані для отримання доступу до чужих облікових записів. Це може статися, якщо токени доступу зберігаються вразливим способом або передаються незахищеним каналом.

Сучасні мобільні застосунки, що використовують постійне з'єднання з сервером, зокрема через WebSocket або HTTP[30], стикаються з низкою потенційних загроз. Основні проблеми безпеки включають:

- атаки (MITM), коли зловмисник перехоплює трафік між клієнтом і сервером. Це може призвести до викрадення персональних даних або їхньої модифікації;
- перехоплення та розшифрування сесій. У разі недостатнього шифрування зловмисник може отримати доступ до конфіденційної інформації;
- невірна аутентифікація та викрадення сесій. Якщо ідентифікатори сесій не захищені, вони можуть бути викрадені, що дозволить зловмиснику увійти в систему під чужим обліковим записом;
- DDoS-атаки — масовані запити до сервера, що можуть вивести його з ладу;
- ін'єкційні атаки (SQL/NoSQL injection), коли зловмисник вводить шкідливий код у запити до бази даних;
- несанкціонований доступ до API, що відкриває можливості для обходу захисту або використання системи в шкідливих цілях.

Одним із найбільш ефективних способів захисту переданих даних є шифрування. Воно забезпечує, що навіть у разі перехоплення дані залишатимуться незрозумілими для зловмисника. Сучасні мобільні застосунки використовують такі технології шифрування:

- TLS (Transport Layer Security) для захисту передачі даних між клієнтом і сервером. Всі дані, що передаються через HTTPS або WSS, шифруються за допомогою TLS;
- AES (Advanced Encryption Standard) — симетричний алгоритм шифрування, який часто використовується для зберігання даних на пристроях або в локальних базах;

- RSA (Rivest–Shamir–Adleman) — асиметричний алгоритм, що використовується для аутентифікації та безпечної передачі ключів.

TLS є особливо важливим для WebSocket, оскільки, використовуючи WSS (WebSocket Secure) зображено на рис. 2.2, протокол забезпечує захист від MITM-атак і шифрування всіх переданих повідомлень.

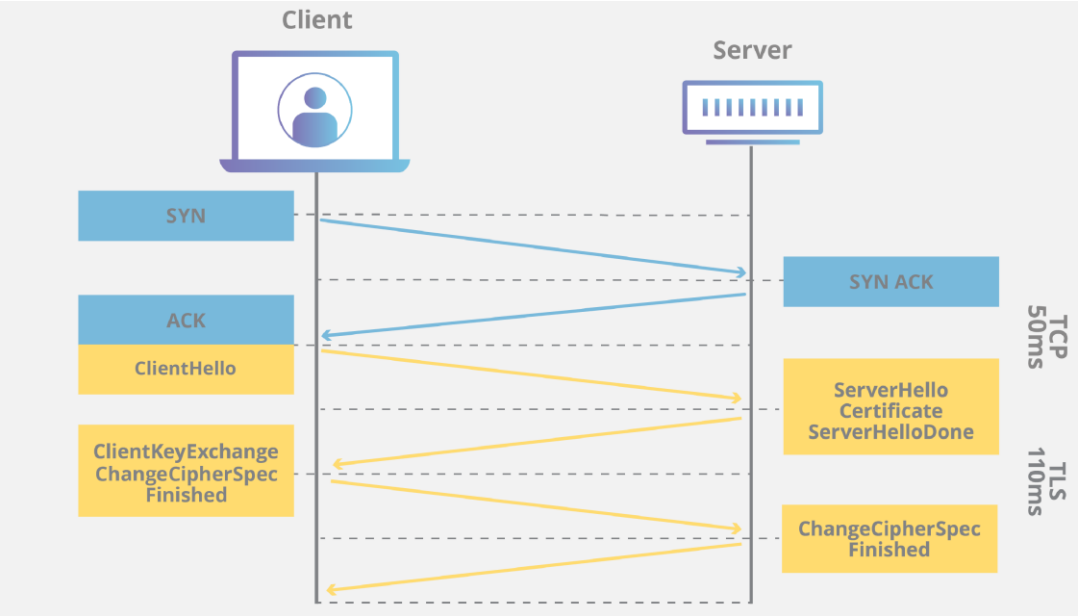


Рисунок 2.7 – Схема процесу TLS Handshake

Безпечна аутентифікація є ключовою вимогою для будь-якого застосунку. Сучасні мобільні застосунки використовують такі підходи до аутентифікації:

- OAuth 2.0 — протокол аутентифікації, що дозволяє користувачам входити через сторонні сервіси, такі як Google або Facebook;
- JWT (JSON Web Token) — токени, що використовуються для зберігання інформації про сесію користувача та підтвердження його особи під час кожного запиту;
- двофакторна аутентифікація (2FA), що підвищує рівень безпеки шляхом вимоги додаткового підтвердження (SMS-код, біометрія).

Авторизація перевіряє, чи має користувач права на виконання певних дій або доступ до певних ресурсів зображено на рис. 2.3. Вона забезпечується за допомогою ACL (Access Control List) або RBAC (Role-Based Access Control)[31].

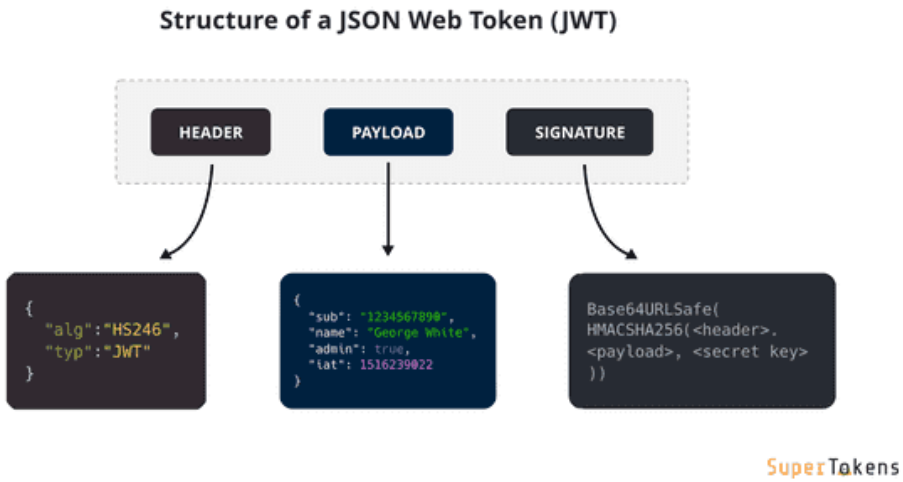


Рисунок 2.8 – Структура JWT Token

Для забезпечення безпеки сесій важливо впровадити кілька рівнів захисту:

- часовий тайм-аут сесії, після якого користувачеві необхідно повторно увійти в систему;
- оновлення токенів для уникнення викрадення застарілих ідентифікаторів;
- закриття сесій при неактивності або при підозрі на злом (наприклад, зміна IP-адреси або підозрілий трафік).

Кожна сесія повинна бути унікальною, а сервер має регулярно перевіряти автентичність токенів для уникнення несанкціонованого доступу.

Для захисту API від атак важливо обмежити доступ до нього за допомогою аутентифікації та авторизації. API повинні перевіряти всі вхідні дані для уникнення SQL або NoSQL-ін’єкцій. Використання параметризованих запитів і ORM (Object-Relational Mapping) допомагає зменшити ризик ін’єкцій.

Також важливо впровадити **рейт-ліміти** на API-запити для запобігання DDoS-атакам. Це обмежує кількість запитів від одного користувача за певний період часу.

Для захисту серверної інфраструктури застосовують фаєрволи та системи виявлення та запобігання вторгнень (IDS/IPS). Фаєрволи фільтрують трафік і блокують підозрілі з'єднання. IDS аналізують трафік у режимі реального часу та повідомляють про можливі атаки, тоді як IPS автоматично блокують підозрілі активності.

Проблеми безпеки є одним із головних викликів під час розробки серверної архітектури мобільних застосунків. Для забезпечення надійної передачі даних

необхідно використовувати багаторівневий підхід до безпеки, що включає шифрування, аутентифікацію, захист сесій, а також контроль доступу до API. Використання фаєрволів та IDS/IPS-систем додатково підвищує рівень захисту. Реалізація комплексних механізмів безпеки дозволяє забезпечити стабільну та безпечну роботу мобільного застосунку навіть у складних умовах.

2.5 Проектування бази даних та вибір схеми для збереження повідомлень

Проектування бази даних для збереження повідомлень є критично важливим етапом розробки будь-якої системи комунікації в реальному часі. База даних повинна не тільки забезпечувати надійне збереження даних, але й підтримувати високу швидкість доступу до інформації, особливо за умови великого навантаження. При розробці архітектури бази даних слід врахувати різні аспекти, включаючи тип даних, обсяг інформації, що зберігається, можливість горизонтального і вертикального масштабування, безпеку даних, а також потреби в резервному копіюванні й відновленні. Важливим є вибір правильної моделі зберігання, яка відповідатиме потребам застосунку, підтримуючи ефективний обмін повідомленнями та низьку затримку.

Рисунок 2.9 – модель бази данних

Першочерговим завданням є визначення вимог до бази даних. Це включає кількість користувачів, обсяг даних, що генерується, та специфіку запитів, які система має обробляти в режимі реального часу. Наприклад, у сценаріях із месенджерами обмін повідомленнями відбувається безперервно, тому важливо мінімізувати затримку між відправленням та отриманням повідомлення. Використання традиційних реляційних баз даних може бути недостатньо ефективним, оскільки вони часто не відповідають потребам масштабованості, які властиві сучасним мобільним застосункам[32]. Тому дедалі популярнішими стають нереляційні бази даних, зокрема NoSQL-рішення, такі як MongoDB, Redis та Cassandra.

NoSQL бази даних забезпечують високу продуктивність за рахунок оптимізованих операцій зчитування та запису даних. Вони також підтримують зберігання даних у форматі ключ-значення, документів, графів або стовпців, що підходить для різних сценаріїв. Наприклад, MongoDB є популярним вибором для проєктів, де важливо зберігати документи, що відповідають JSON-формату. Це спрощує обробку повідомлень у чатах, оскільки кожне повідомлення можна зберігати як окремий документ із властивостями, такими як час відправлення, ідентифікатор користувача та текст. Redis, у свою чергу, часто використовується для тимчасового зберігання повідомлень або кешування даних, що забезпечує миттєвий доступ до інформації.

Одним із ключових рішень при проєктуванні бази даних є вибір між SQL і NoSQL підходами. Реляційні бази даних, такі як MySQL або PostgreSQL, забезпечують транзакційну цілісність і надійність завдяки дотриманню принципів ACID. Однак у системах із високою інтенсивністю обміну повідомленнями реляційні бази можуть виявитися надто повільними. В таких випадках використання NoSQL-баз, що дотримуються принципів CAP-теореми (Consistency, Availability, Partition tolerance), надає більше можливостей для масштабування та продуктивності. Наприклад, у розподілених системах із мільйонами одночасних підключень вибір бази даних, яка пріоритетно підтримує доступність і стійкість до розбиття, стає вирішальним фактором.

Важливо також визначити структуру зберігання повідомлень. Найбільш поширеними підходами є зберігання даних у вигляді окремих повідомлень, що пов’язуються із сесіями або чатами. Кожне повідомлення має свій унікальний ідентифікатор, а також метадані, які дозволяють фільтрувати та сортувати інформацію. Це може включати час створення, ідентифікатор користувача, статус доставки або реакції інших користувачів. Деякі системи також додають до повідомлень інформацію про медіафайли або вбудовують посилання на них, що дозволяє уникнути перевантаження основної бази даних великими обсягами даних.

При розробці бази даних слід також врахувати питання безпеки та конфіденційності даних. У сучасних мобільних застосунках часто використовуються методи шифрування як на рівні окремих повідомлень, так і на рівні з’єднання між клієнтом і сервером. Наприклад, бази даних, такі як MongoDB

і PostgreSQL, підтримують шифрування даних у стані спокою (at rest) та під час передачі (in transit). Це важливо для захисту інформації від несанкціонованого доступу та дотримання вимог законодавства, таких як GDPR. Крім того, управління доступом до даних реалізується через розмежування ролей і привілеїв, що мінімізує ризики витоку конфіденційної інформації.

Окрім безпеки, важливим аспектом є забезпечення надійності та доступності даних. Використання реплікації дозволяє створювати кілька копій бази даних, що підвищує стійкість системи до збоїв і мінімізує втрати даних. Резервне копіювання також є важливою частиною управління даними, що дозволяє відновлювати інформацію у разі аварійних ситуацій. Сучасні системи баз даних пропонують інструменти для автоматизації резервного копіювання та моніторингу стану з'єднань, що полегшує підтримку безперервної роботи застосунку.

При розробці схеми зберігання повідомлень важливо також врахувати потреби в масштабуванні системи. Масштабування може бути вертикальним (збільшення потужності серверів) або горизонтальним (додавання нових вузлів до системи)[33]. NoSQL бази даних зазвичай краще підходять для горизонтального масштабування, що дозволяє додавати нові сервери в міру зростання кількості користувачів. Використання технологій на зразок кластеризації Redis або шардінгу в MongoDB забезпечує рівномірний розподіл навантаження між вузлами.

Вибір технологій для реалізації бази даних також залежить від типу даних і запитів, які система має обробляти. Наприклад, для застосунків із великим обсягом текстових повідомлень доцільно використовувати бази даних із підтримкою повнотекстового пошуку. Це дозволяє швидко знаходити повідомлення за ключовими словами або фразами. У випадках, коли система має підтримувати передачу медіафайлів, таких як зображення чи відео, доцільно розділити базу даних на кілька частин, виділивши окремі сховища для великих файлів. Це знижує навантаження на основну базу даних та підвищує загальну продуктивність системи.

Таким чином, проєктування бази даних для мобільного застосунку з використанням WebSocket потребує врахування багатьох факторів, включаючи продуктивність, масштабованість, безпеку та надійність. Правильний вибір схеми зберігання даних та інструментів розробки дозволяє забезпечити стабільну та

ефективну роботу системи, мінімізуючи затримки та забезпечуючи високу доступність даних навіть за умов великого навантаження.

2.6 Забезпечення масштабованості та стійкості серверної частини

Забезпечення масштабованості та стійкості серверної частини є ключовими аспектами при проєктуванні системи, яка повинна обробляти великий обсяг трафіку, надавати безперервний доступ до функцій та підтримувати стабільну роботу навіть за умов пікових навантажень[34]. Масштабованість забезпечує можливість адаптуватися до зростання кількості користувачів і запитів, тоді як стійкість дозволяє системі залишатися працездатною навіть при виникненні збоїв у мережі, обладнанні або програмному забезпеченні. Для цього розробники використовують різноманітні технології та підходи.

Основою масштабованої системи є розподілена архітектура. На противагу монолітним системам, які важко змінювати та масштабувати, розподілені системи розділяють серверні компоненти на окремі модулі зображено на рис. 2.4. Це може бути реалізовано за допомогою мікросервісної архітектури, де кожен сервіс відповідає за окрему функцію й може бути масштабований незалежно від інших. Мікросервіси взаємодіють через API або повідомлення, що забезпечує гнучкість і високу продуктивність.

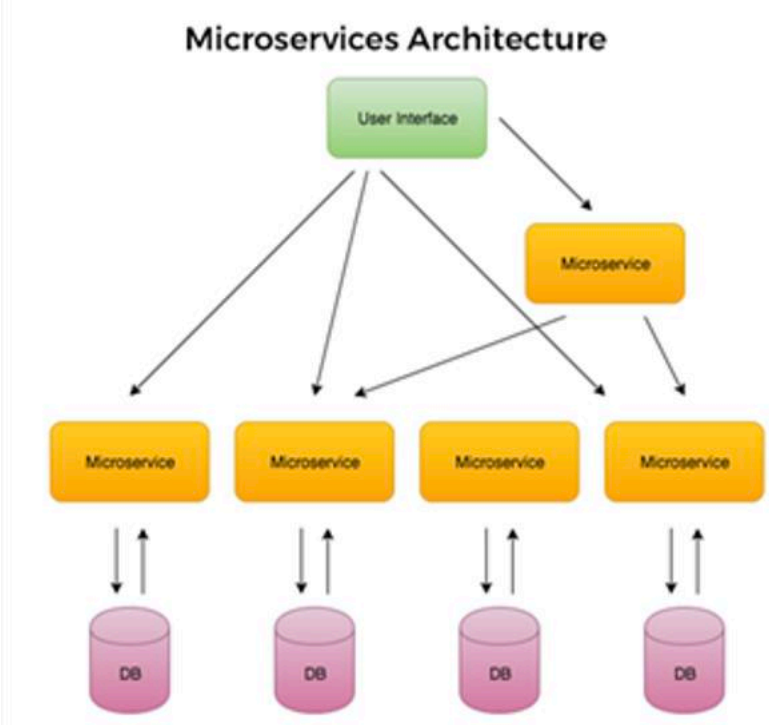


Рисунок 2.10 – Приклад мікросервісної архітектури з навантаженням на різні компоненти

Іншим важливим компонентом масштабування є використання балансувальників навантаження. Вони розподіляють запити між кількома серверами або вузлами, зменшуючи навантаження на кожен з них і підвищуючи загальну пропускну здатність системи. Балансувальники можуть працювати на різних рівнях: від простого розподілу HTTP-запитів до розумних рішень, що аналізують стан серверів та обирають оптимальні маршрути. Це також підвищує стійкість системи, адже при виході одного вузла з ладу трафік перенаправляється на інші.

Горизонтальне та вертикальне масштабування є двома основними підходами для збільшення ресурсів[35]. Вертикальне масштабування передбачає додавання ресурсів до існуючих серверів, таких як оперативна пам'ять чи потужніші процесори. Це підходить для систем із відносно стабільними вимогами, однак має обмеження, пов’язані з фізичною архітектурою обладнання. Горизонтальне масштабування, навпаки, полягає в додаванні нових серверів до кластеру, що забезпечує більшу гнучкість зображено на рис. 2.5. Хмарні технології, зокрема AWS, Azure та Google Cloud, спрощують горизонтальне масштабування за рахунок автоматичного додавання або видалення серверів залежно від поточного навантаження.

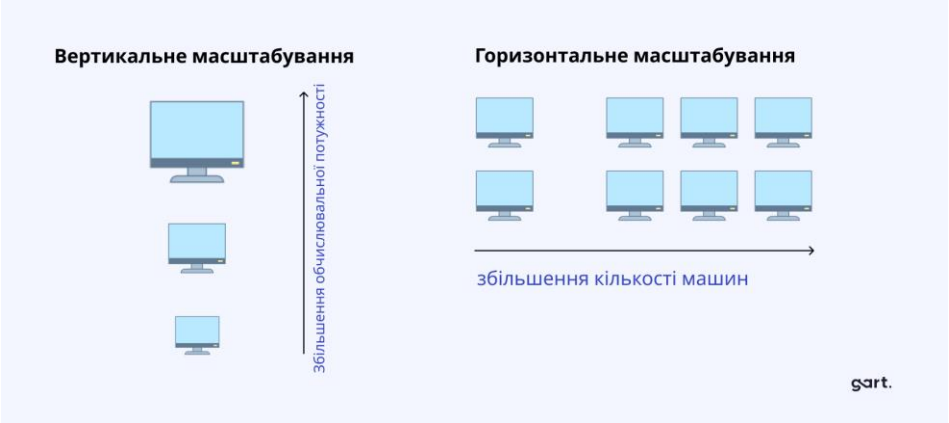


Рисунок 2.11 – Порівняння горизонтального та вертикального масштабування у хмарних середовищах

Контейнеризація є ще одним інструментом, який сприяє масштабованості. Використання Docker та Kubernetes дозволяє запускати додатки в ізольованих контейнерах, що можуть бути легко розгорнуті й масштабовані за потреби. Kubernetes автоматизує процес управління контейнерами, забезпечуючи балансування навантаження, автоматичний перезапуск збоїв та масштабування на

основі метрик. Завдяки контейнеризації команди розробників можуть швидше розгортати оновлення та підтримувати безперервність роботи.

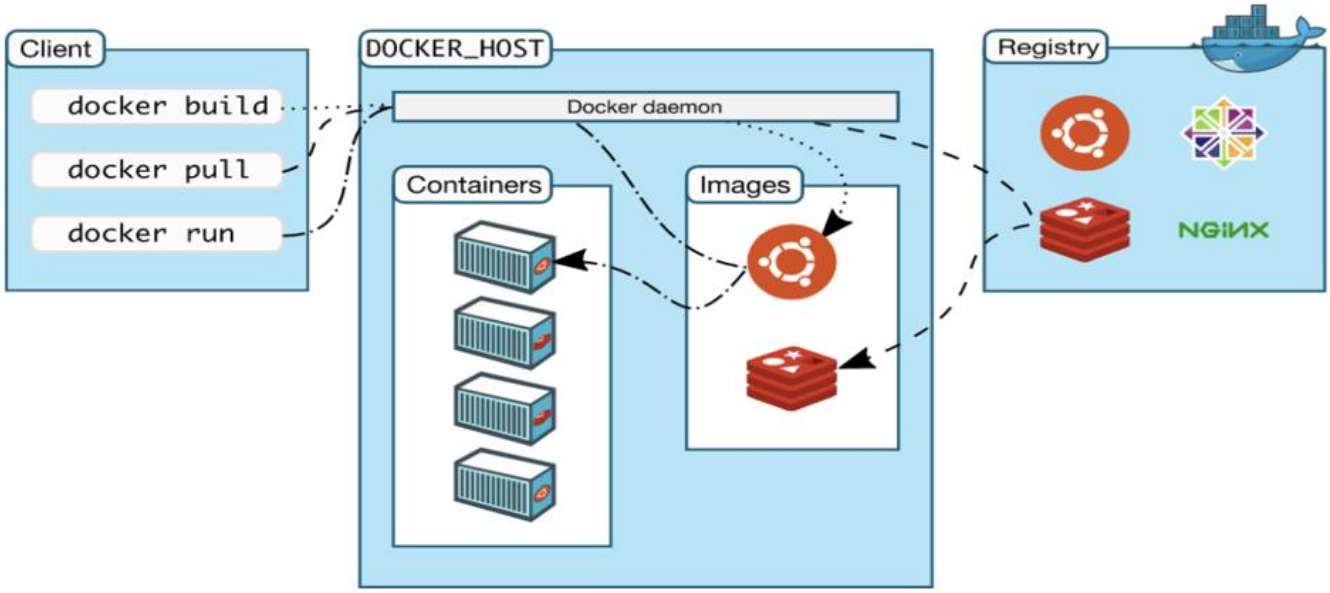


Рисунок 2.12 – Схема роботи Docker

Реплікація та шардінг використовуються для підвищення стійкості баз даних. Реплікація передбачає створення копій даних на кількох серверах, що забезпечує захист від втрат інформації та прискорює доступ до даних, оскільки запити можуть бути оброблені найближчою копією[36]. Шардінг, у свою чергу, розбиває базу даних на кілька частин (шардів), кожна з яких зберігає частину даних. Це допомагає розподілити навантаження між кількома серверами та уникнути перевантаження.

Кешування даних також відіграє важливу роль у забезпеченні масштабованості. Використання таких рішень, як Redis або Memcached, дозволяє зберігати часто запитувані дані у швидкій пам’яті, що зменшує кількість запитів до основної бази даних. Це особливо важливо в системах із великим обсягом трафіку, таких як мобільні застосунки для обміну повідомленнями, де кожна мілісекунда має значення.

Забезпечення стійкості також включає впровадження механізмів моніторингу та автоматичного відновлення. Інструменти на кшталт Prometheus та Grafana надають можливість відстежувати стан серверів і баз даних у реальному часі, а також налаштовувати алерти на випадок відхилень від нормальної роботи. Системи автоматичного відновлення можуть перезапускати збої або перенаправляти трафік на резервні вузли, забезпечуючи безперервність обслуговування навіть у разі критичних помилок.

Нарешті, масштабованість і стійкість не можуть бути повністю забезпечені без планів на випадок аварій. Розробка та тестування планів аварійного відновлення дозволяє швидко реагувати на неочікувані збої, мінімізуючи час простою. Це може включати створення резервних копій, розгортання резервних серверів або автоматичне переключення на інші дата-центри у разі виникнення проблем.

Загалом, успішне забезпечення масштабованості та стійкості серверної частини мобільного застосунку вимагає комплексного підходу, що включає розподілену архітектуру, контейнеризацію, балансування навантаження, реплікацію даних, моніторинг і плани аварійного відновлення. Лише в поєднанні ці інструменти та підходи можуть забезпечити надійну й ефективну роботу системи, готової до зростання кількості користувачів і підвищених вимог до продуктивності.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ ЗАСТОСУНКУ

3.1 Впровадження WebSocket-сервера на базі обраних технологій

Процес починається з вибору технологічного стеку для сервера, що відповідає вимогам до продуктивності та надійності. Наприклад, для високих навантажень добре підходить Node.js завдяки асинхронному обробленню та подієвій моделі, яка дозволяє ефективно обробляти одночасні запити. Іншими варіантами можуть бути Python з Django Channels для швидкої розробки або Java з Spring Boot для складних бізнес-логік. Вибір технології слід обґрунтувати відповідно до вимог вашого застосунку, зокрема продуктивності та надійності.

Node.js – це потужна технологія, яка змінила підхід до серверного JavaScript. Вона побудована на основі JavaScript і двигуна V8, що забезпечує високу продуктивність і швидке виконання коду. Основна ідея полягає в використанні однопоточності, де асинхронне введення-виведення та події відіграють ключову роль у забезпеченні швидкого і ефективного оброблення запитів.

Node.js має архітектуру, яка дозволяє обробляти кілька запитів одночасно. Клієнтські запити не блокують сервер, а обробка відбувається через механізм подій. Це робить Node.js ідеальним для реальних застосунків, таких як мережеві сервери, системи чату, потокова передача даних та інші веб-застосунки, де швидкість і низька затримка є критично важливими.

Однією з основних переваг Node.js є його асинхронність. Використовуючи однопотокову архітектуру, Node.js дозволяє обробляти багато операцій одночасно без блокування основного потоку виконання. Це досягається завдяки спеціальному Event Loop, який ефективно керує асинхронними подіями та операціями. Це дає можливість розробникам створювати ефективні та масштабовані серверні рішення без необхідності працювати з многозадачністю і багатопоточністю на низькому рівні.

Бібліотека модулів Node.js є ще однією його значною перевагою. Вона включає тисячі готових до використання модулів для виконання різноманітних завдань, від запиту даних із баз даних до інтеграції з сторонніми сервісами. Це

дозволяє розробникам швидше створювати функціональні серверні додатки, не витрачаючи час на розробку всього з нуля.

Node.js також підтримує роботу з серверними фреймворками, такими як Express, що додає ще більше можливостей для розробки веб-застосунків. Express дозволяє швидко створювати маршрути, обробляти запити та відповідати зручною і структурованою архітектурою.

Одним із основних викликів при використанні Node.js є забезпечення безпеки. Через відкриту природу технології, її можна легко зловживати, якщо не налаштувати належним чином. Важливо забезпечити належне налаштування доступу до баз даних, обробку запитів і захист від атак. Node.js також дозволяє використовувати механізми аутентифікації та авторизації, що забезпечують додатковий рівень безпеки при роботі з користувацькими даними.

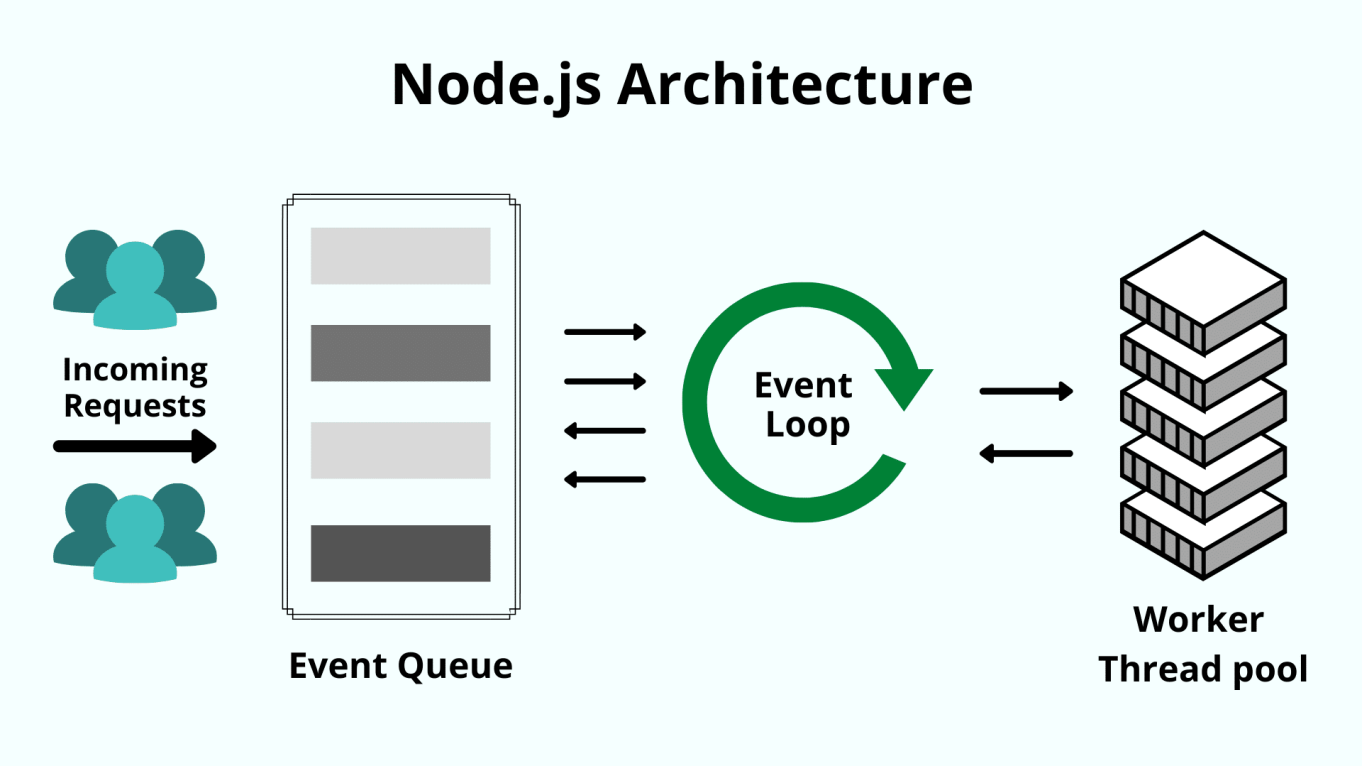


Рисунок 3.1 – Архітектура Node.js

Загалом, Node.js є потужним інструментом для сучасних розробників, який відкриває нові можливості для створення швидких і ефективних серверних додатків. Його асинхронна архітектура, підтримка модульності та велика кількість доступних модулів роблять його вибором номер один для багатьох розробників по всьому світу.

Далі слід розгорнути базову інфраструктуру WebSocket-сервера. На цьому етапі налаштовуються з'єднання для прийому та обробки запитів від клієнтів. WebSocket з'єднання вимагає постійного каналу між клієнтом і сервером, що дозволяє обмінюватися повідомленнями без необхідності повторних запитів. Це налаштування може включати використання інструментів для балансування навантаження, таких як Nginx або HAProxy, що розподіляють підключення між кількома серверами для забезпечення масштабованості.

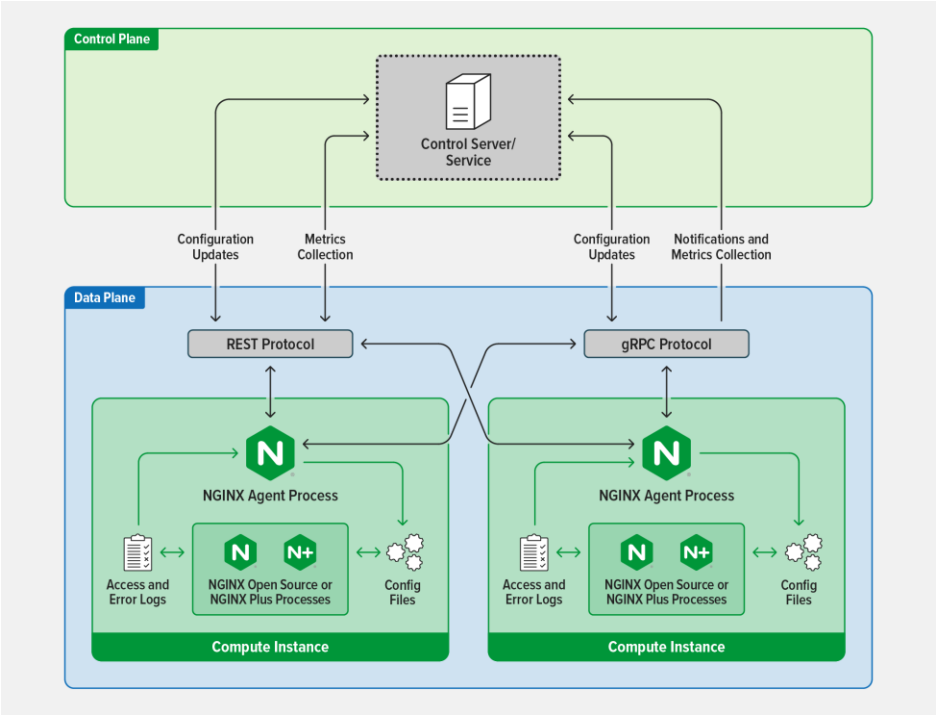


Рисунок 3.2 – Агент Nginx

Для організації сесій варто реалізувати механізм управління ідентифікацією та автентифікацією користувачів. Так як WebSocket сам по собі не підтримує сесії, можна інтегрувати токени або інші ідентифікатори користувачів, що передаються на етапі підключення. В цьому випадку Redis є популярним вибором для зберігання станів сесій через швидкий доступ до даних у пам'яті.

На етапі обробки збоїв і відновлення з'єднань реалізують механізм для автоматичного повторного підключення при розривах зв’язку, зокрема в умовах нестабільного мережевого з’єднання. Цей механізм часто доповнюється стратегією експоненційної затримки для управління частотою повторних спроб, зменшуючи навантаження на систему. Це особливо важливо для мобільних застосунків, де користувачі можуть часто змінювати мережі.

Безпека також займає значну частину роботи над WebSocket-сервером. Оскільки постійне з’єднання відкриває можливість для різноманітних атак, таких як перехоплення даних або відмова в обслуговуванні, важливо реалізувати надійне шифрування через SSL/TLS для захисту каналу, а також автентифікацію користувачів на етапі підключення, що допомагає знизити ризики атак.

PM2 є потужним інструментом для управління і моніторингу Node.js додатків, який значно полегшує життя розробників і адміністраторів сервісів. Він дозволяє ефективно контролювати процеси додатків, забезпечувати їхню безперервність і швидко впроваджувати оновлення. Однією з його ключових особливостей є здатність працювати як демона, тобто додаток, що залишається запущеним і не залежить від сеансів оболонки користувача. Це особливо важливо для продакшн-середовища, оскільки дозволяє зберігати стабільну роботу сервісів навіть після перезавантаження або оновлення сервера.

PM2 підтримує автоматичне перезавантаження додатків при зміні коду або виникненні помилок, що забезпечує високу доступність додатків. Розробники можуть впроваджувати нові функції або виправлення помилок без необхідності зупиняти додаток, що критично важливо для підтримки постійної роботи сервісу. Крім того, PM2 дозволяє додаткам працювати в кількох процесах за допомогою режиму кластера, що підвищує продуктивність і стабільність додатків. Кластери допомагають розподіляти навантаження, що значно покращує швидкість обробки запитів і зменшує ймовірність виникнення перевантажень.

Один з найбільших аспектів використання PM2 – це його здатність до моніторингу і логування. PM2 автоматично обробляє логування додатків, дозволяючи переглядати детальні показники використання ресурсів, такі як CPU і пам’ять, для кожного процесу. Це важливо для підтримки стабільної роботи додатків у реальному часі, оскільки надає розробникам інформацію про те, як додаток використовується, і дозволяє швидко виявляти проблеми, перш ніж вони стануть критичними. Автоматичне обертання логів також дозволяє уникати перевантажень жорсткого диска і допомагає легко усувати помилки, якщо вони виникають.

PM2 також підтримує автоматизацію процесів горизонтального масштабування. Це дозволяє додаткам розгортатися на кількох серверах, що

забезпечує рівномірний розподіл навантаження. Балансувальник навантаження забезпечує можливість динамічного перенаправлення трафіку між серверами, що дозволяє додаткам підтримувати високу доступність навіть за умови значного зростання кількості користувачів. Це критично важливо для додатків, які мають велике число одночасних з'єднань або великі обсяги даних, що обробляються.

Однією з головних функцій PM2 є підтримка автоматичного відновлення з'єднань при втраті сигналу або зміні мережі. Це дозволяє клієнтам відновити з'єднання без втрати даних, що особливо важливо для додатків, які працюють у реальному часі. Для цього PM2 використовує спеціальні heartbeat-повідомлення, які періодично надсилаються між клієнтом і сервером для перевірки активності з'єднання. Якщо клієнт не отримує відповідь протягом встановленого часу, з'єднання розривається, і PM2 ініціює спробу відновлення. Це дозволяє забезпечити безперервність роботи додатків навіть за умови проблем з мережею.

Захист і безпека є ще одним критичним аспектом використання PM2. Оскільки WebSocket-з'єднання може залишатися відкритим протягом тривалого часу, існує додатковий ризик для безпеки. Для захисту з'єднань використовуються шифрування за допомогою WSS (WebSocket Secure) і обмеження кількості одночасних з'єднань від одного клієнта. Крім того, забезпечується аутентифікація і авторизація підключень, що дозволяє захистити дані від маніпуляцій і несанкціонованого доступу.

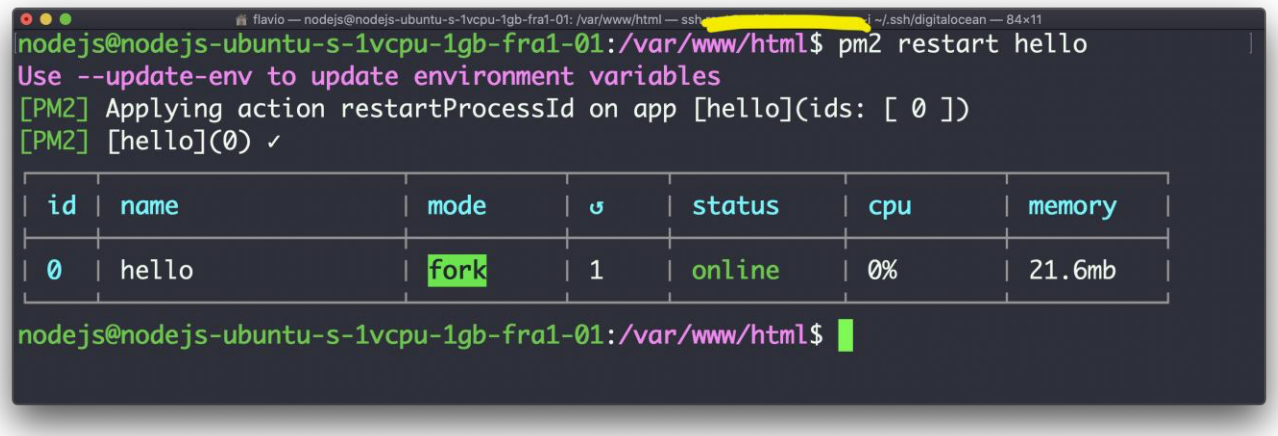


Рисунок 3.3 – Інтерфейс PM2

Використання PM2 значно спрощує розробку, моніторинг і підтримку Node.js додатків. Він надає розробникам потужні інструменти для автоматизації і

оптимізації процесів, що дозволяє забезпечити стабільну роботу додатків навіть у складних умовах. Завдяки його функціям моніторингу, автоматизації масштабування та підтримки високої доступності, PM2 є невід'ємним інструментом для сучасних розробників і адміністраторів сервісів.

Оптимізація передачі даних має на меті зменшення використання пропускної здатності мережі і підвищення швидкості відповіді сервера. Одним з варіантів є об'єднання кількох повідомлень у пакет, що зменшує кількість мережевих запитів. Використання стиснення для великих обсягів даних також допомагає мінімізувати навантаження на канал.

Однією з головних функцій PM2 є реальний моніторинг, що дозволяє контролювати використання ЦП і пам'яті кожного процесу. Це важливо для забезпечення стабільної роботи додатків у реальному часі. PM2 також автоматично обробляє логування і забезпечує автоматичну обертку логів, що спрощує моніторинг і усунення несправностей. Крім того, він підтримує горизонтальне масштабування, що дозволяє додаткам працювати в декількох інстанціях на різних серверах, забезпечуючи рівномірний розподіл навантаження.

Завдяки цим функціям, PM2 значно спрощує управління і підтримку Node.js додатків у продакшн середовищах, забезпечуючи їх стабільність і надійність.

Завершується впровадження WebSocket-сервера ретельним тестуванням і моніторингом. Тестування на продуктивність дозволяє оцінити, чи сервер здатний витримувати передбачене навантаження. За допомогою таких інструментів, як Apache JMeter, можна моделювати сценарії інтенсивного використання і визначати слабкі місця системи. Моніторинг у реальному часі з використанням Prometheus і Grafana допомагає слідкувати за важливими метриками, а також швидко виявляти та реагувати на відхилення в роботі.

Останнім етапом є інтеграція WebSocket-сервера з іншими компонентами системи. Це включає обробку повідомлень від клієнтів, зберігання даних у базі для доступу при необхідності і синхронізацію з іншими серверними сервісами. Така інтеграція забезпечує безперебійну взаємодію між серверною частиною та клієнтами, що підключаються до WebSocket.

3.2 Інтеграція серверної частини з клієнтським мобільним додатком

Інтеграція серверної частини з клієнтським мобільним додатком включає налаштування каналів зв’язку між клієнтом і сервером, забезпечення зручного та стабільного обміну даними, обробку подій і станів з’єднання, а також реалізацію функцій обробки повідомлень та даних. Основними завданнями інтеграції є забезпечення безперервної передачі інформації, збереження стабільності з'єднання в умовах мінливого інтернет-з’єднання та управління сеансами для захисту ідентифікації користувача.

Спершу налаштовується базове підключення клієнта до WebSocket-сервера. Це включає відкриття з'єднання між клієнтським додатком і сервером через встановлений URI WebSocket. Клієнт має вміти автоматично підключатися до сервера під час запуску застосунку і підтримувати з’єднання протягом усього сеансу зображено на рис. 3.1. Використання асинхронного обміну даними забезпечує швидку обробку повідомлень і дозволяє уникнути блокувань інтерфейсу користувача, що покращує загальний досвід взаємодії.

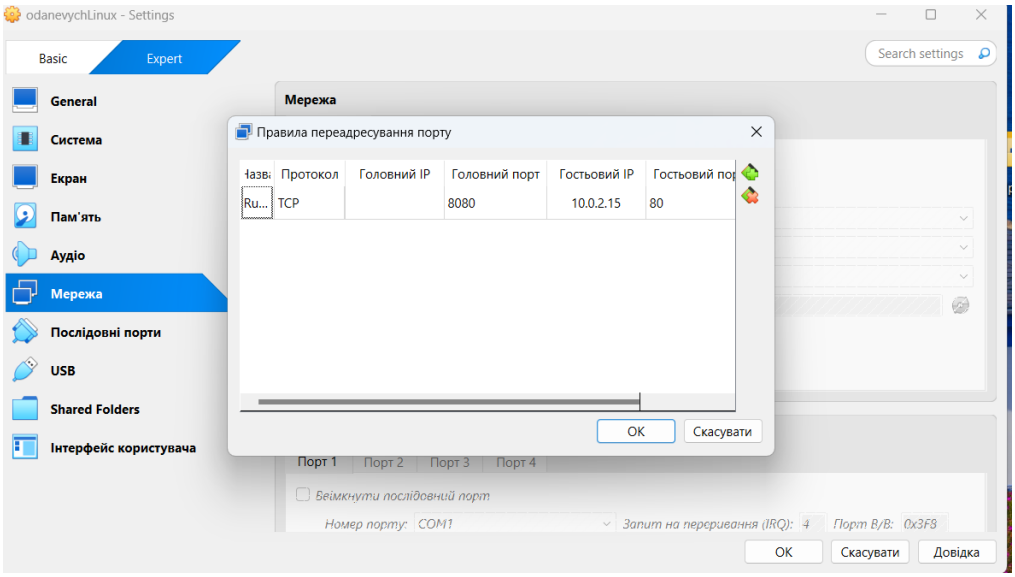


Рисунок 3.4 – Налаштування переадресування порту

Для стабільності з'єднання в мобільному додатку часто реалізується механізм автоматичного перепідключення у разі розриву зв’язку. Це важливо, оскільки мобільні користувачі можуть часто змінювати мережу або зіштовхуватися з перервами у з’єднанні. Механізм автоматичного перепідключення дозволяє

додатку намагатися відновити з'єднання через певні проміжки часу і забезпечує безперервність зв'язку без необхідності взаємодії з користувачем.

Важливим аспектом інтеграції є управління сесіями і забезпечення безпеки. Для цього можна використовувати токени або інші механізми аутентифікації, які надсилаються в запиті на підключення до WebSocket-сервера. Наприклад, клієнт може надсилати токен користувача разом із запитом на підключення, що дозволяє серверу ідентифікувати користувача і перевірити його права доступу. Це забезпечує захист даних і обмежує доступ до чутливої інформації. Після встановлення з'єднання, сервер може періодично перевіряти токени, щоб забезпечити, що тільки авторизовані користувачі мають доступ до каналу.

Для передачі даних між клієнтом і сервером використовується формат JSON або Protobuf, які забезпечують зручний та структурований обмін даними. Важливо, щоб клієнтський додаток був здатен обробляти вхідні повідомлення від сервера та відповідати на них відповідним чином зображено на рис. 3.2. Наприклад, якщо сервер надсилає оновлення даних у реальному часі, мобільний додаток повинен мати обробник для цих повідомлень, що оновлює відповідні компоненти інтерфейсу користувача. Зокрема, для чатів це означає негайне відображення нових повідомлень у діалозі без необхідності оновлення сторінки або повторного завантаження застосунку.

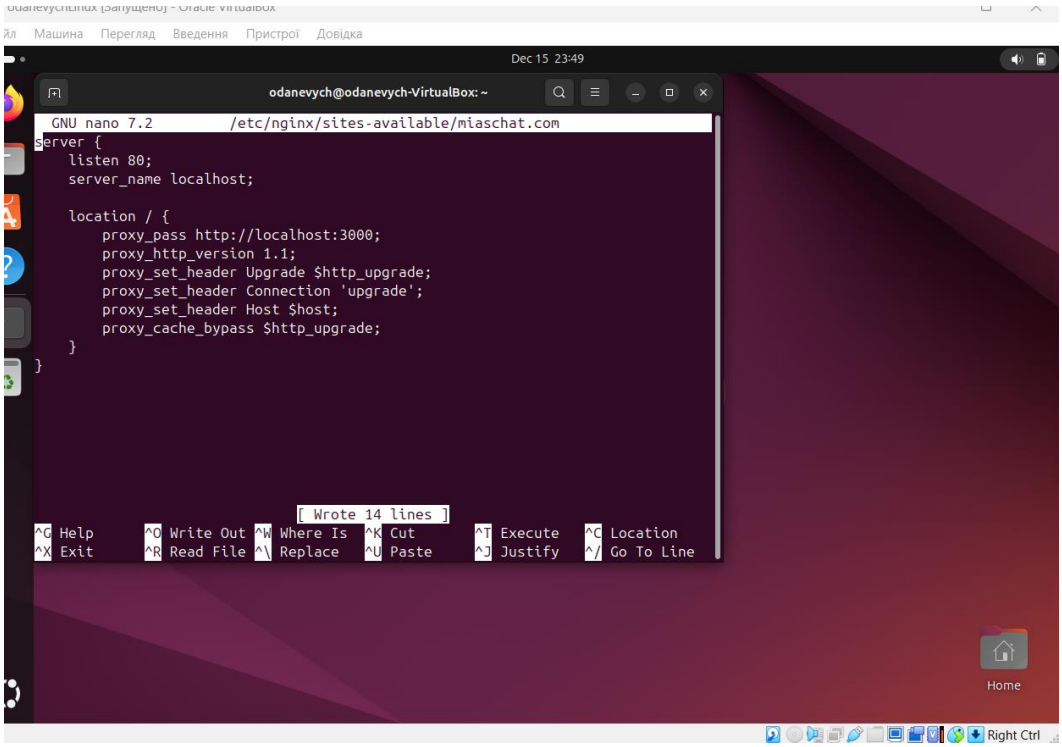


Рисунок 3.5 – Налаштування серверу nginx

Для оптимізації обміну повідомленнями і підвищення продуктивності часто використовуються додаткові технології, наприклад, стиснення повідомлень. Стиснення дозволяє зменшити обсяг даних, що передаються по мережі, і тим самим прискорити обмін даними. Це особливо корисно для мобільних додатків, де пропускна здатність може бути обмеженою. Окрім цього, для швидшої доставки повідомлень можуть застосовуватися черги повідомлень та механізми кешування, які допомагають оптимізувати обробку даних і забезпечують стабільну швидкість роботи застосунку навіть при значних навантаженнях на сервер.

Моніторинг і логування є важливими аспектами інтеграції клієнтського додатку із сервером. За допомогою інструментів моніторингу можна відслідковувати з'єднання і продуктивність у реальному часі, виявляючи проблеми зі зв’язком, затримки, помилки в обробці повідомлень тощо. Це дозволяє розробникам своєчасно реагувати на проблеми і підтримувати якість роботи додатку на високому рівні. Інструменти логування допомагають виявляти проблеми, пов’язані із специфічними запитами або подіями, що відбуваються в додатку, і забезпечують цінну інформацію для подальшої оптимізації.

Завершальний етап інтеграції включає тестування з'єднання та обміну повідомленнями в умовах, максимально наближених до реальних сценаріїв використання. Це включає перевірку роботи під час змін мережевих умов, перевірку продуктивності і відмовостійкості додатку, а також тестування на безпеку. Важливо також провести тестування на різних платформах і пристроях, щоб забезпечити сумісність і надійність роботи застосунку.

3.3 Тестування системи на продуктивність та стійкість

Тестування системи на продуктивність і стійкість є критичним етапом у процесі розробки серверної частини мобільного застосунку. Це забезпечує ефективність роботи системи при високих навантаженнях, а також її здатність залишатися стабільною та відповідати на запити у найрізноманітніших умовах. У сучасних застосунках, особливо тих, які використовують WebSocket для обміну даними в реальному часі, продуктивність та стійкість серверної частини є визначальними факторами для забезпечення якісної взаємодії з користувачем.

Тестування продуктивності системи включає кілька етапів. Один з них — це вимірювання часу відповіді сервера на запити від клієнтів. Виконуються тести, що моделюють різні рівні навантаження на систему, починаючи від мінімального та завершуючи піковими рівнями. Наприклад, при одночасному підключенні декількох тисяч користувачів до WebSocket-сервера важливо перевірити, як змінюється час відповіді при зростанні кількості з’єднань. Це дозволяє визначити максимальні можливості сервера та обмеження його продуктивності.

Ще один метод — це тестування пропускної здатності. Для застосунків із підтримкою реального часу, таких як чати, ігри або системи оповіщення, критично важливим є високий рівень пропускної здатності. Пропускна здатність вимірюється кількістю даних, що система здатна обробити за одиницю часу. За допомогою спеціалізованих інструментів для навантажувального тестування можна моделювати великий потік повідомлень і перевірити, як сервер справляється з їх обробкою. Це дає змогу оцінити, наскільки швидко сервер обробляє інформацію, а також як зменшення чи збільшення пропускної здатності впливає на загальну продуктивність системи.

Тестування стійкості охоплює ситуації, коли система піддається екстремальним умовам, таким як перевантаження, різке збільшення навантаження, відмова компонентів або різкі зміни в мережевих умовах. Це дозволяє оцінити, чи система здатна продовжувати працювати без значних втрат у функціональності і якості роботи. Під час такого тестування можна, наприклад, перевірити, як сервер буде реагувати, якщо певний компонент вийде з ладу або якщо буде значне збільшення кількості підключень в короткий проміжок часу. Стрес-тестування є

важливою частиною цього процесу, оскільки дозволяє розробникам виявити слабкі місця в архітектурі і налаштувати систему так, щоб вона була стійкою до непередбачуваних обставин.

Автоматизація тестування також є важливим аспектом процесу забезпечення якості, оскільки вона дозволяє проводити повторювані тести в автоматичному режимі. Це дає змогу швидко оцінювати продуктивність і стійкість сервера при кожному оновленні. Інструменти для автоматизації можуть бути налаштовані так, щоб проводити тестування у визначені моменти, наприклад, при внесенні змін до коду або перед випуском нових версій. Це забезпечує безперервну інтеграцію і своєчасне виявлення проблем, що може значно підвищити якість фінального продукту.

Ще одним важливим етапом є аналіз отриманих результатів тестування. Зібрані дані дозволяють виявити основні причини затримок, падінь і інших проблем у роботі сервера. За допомогою інструментів моніторингу можна відстежувати використання ресурсів, зокрема процесора, оперативної пам’яті та мережевих ресурсів, і на основі цих даних оптимізувати архітектуру системи. Наприклад, якщо моніторинг показує, що сервер часто перевантажений при обробці великих обсягів повідомлень, можна розглянути можливість горизонтального масштабування або налаштування кешування даних для оптимізації продуктивності.

На завершальному етапі проводиться загальне тестування системи в умовах, максимально наближених до реальних, щоб перевірити, як вона буде функціонувати на різних платформах і під різними навантаженнями. Це включає як тестування на пристроях користувачів, так і моделювання реальних сценаріїв використання системи. Результати тестів дозволяють остаточно налаштувати систему і забезпечити її стабільну роботу, що гарантує користувачам якісний досвід взаємодії з мобільним застосунком.

3.4 Вибір апаратного забезпечення для серверів з різним рівнем навантаження

При створенні серверної частини для мобільного застосунку з використанням технологій реального часу, таких як WebSocket, критично важливим є вибір апаратного забезпечення. Конфігурація сервера повинна враховувати обсяги очікуваного трафіку, кількість одночасних з'єднань, типи даних, що передаються, та вимоги до швидкодії. Цей розділ деталізує характеристики обладнання для різних рівнів навантаження: середнього, високого та надвисокого.

Для серверів під середнє навантаження важливо обирати компоненти, які забезпечать стабільну роботу при невеликих і середніх вимогах. Такий сервер підходить для обслуговування корпоративних сайтів, чат-додатків або вебсервісів для локального бізнесу, де кількість одночасних з’єднань зазвичай не перевищує 100. Процесори, як-от AMD Ryzen 5 5600X або Ryzen 7 5800X, забезпечують гарне співвідношення між продуктивністю та енергоефективністю. За даними фахівців з ресурсів, таких як TechTarget і PCWorld, ці процесори надають достатню продуктивність для невеликих до середніх завдань, зокрема для керування невеликим сервером. Оперативна пам’ять Corsair Vengeance LPX обсягом від 8 до 16 GB дозволяє ефективно обробляти запити і підтримувати стабільну роботу сервера під час пікових навантажень. SSD Samsung 970 EVO є оптимальним рішенням для швидкого доступу до даних, забезпечуючи низьку латентність і високу швидкість читання/запису, що особливо важливо для обробки інформації в реальному часі. Додатковий HDD Seagate Barracuda 1TB можна використовувати для довготривалого зберігання архівів або менш критичних даних, забезпечуючи економічно ефективне рішення для зберігання великих обсягів інформації.

Рисунок 3.6 – Процесор AMD Ryzen 5 5600G

Для високих навантажень потрібні більш потужні компоненти, які зможуть підтримувати стабільну продуктивність при великих трафіках і понад 100 одночасних з'єднань. Процесори, як Intel Xeon E5-2667 v4 або AMD Ryzen 9 5900X, забезпечують багатопотокову обробку, що важливо для роботи з великими обсягами даних. За даними науковців з Cisco, ці процесори оптимізовані для обробки великого числа паралельних запитів, що характерно для онлайн-магазинів і стрімінгових сервісів. Оперативна пам'ять Corsair Vengeance LPX обсягом 32 GB дозволяє ефективно працювати з декількома сервісами одночасно без втрати продуктивності. Використання SSD з конфігурацією RAID 10 на базі Samsung 970 EVO підвищує надійність даних, забезпечуючи швидкість читання/запису. Така конфігурація з кількома дисками SSD дозволяє досягти значного підвищення ефективності і надійності, що важливо для високонавантажених додатків.

Для серверів з надвисоким навантаженням, які обслуговують критичні застосунки, як фінансові сервіси або стрімінгові платформи у форматі 4K і більше, необхідні компоненти, які забезпечують найвищу продуктивність. Тут потрібні процесори, як AMD Ryzen Threadripper PRO 3990X, які забезпечують потужну

багатопотокову обробку. Вони дозволяють ефективно працювати з великими обсягами даних та кількома паралельними запитами одночасно, що необхідно для обробки фінансових даних або відео-аналітики. Оперативна пам’ять Corsair Dominator Platinum обсягом до 128 GB і можливість її розширення дозволяють обробляти великий обсяг даних без втрати продуктивності. RAID 10 на основі кількох SSD і швидкісний NVMe SSD забезпечують найвищу швидкодію та безпеку даних під час роботи з великими обсягами інформації. Це особливо важливо для забезпечення надійності в критичних застосунках, де вимоги до безпеки та продуктивності є найвищими.

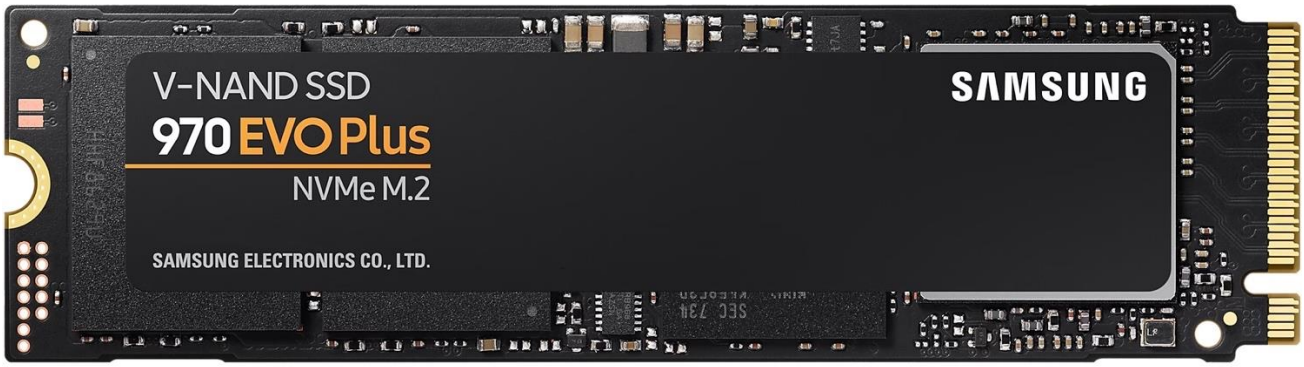


Рисунок 3.7 – Оперативна пам’ять Corsair Dominator Platinum

Для серверів, які використовуються в середніх, високих і надвисоких навантаженнях, пристрої вводу та виводу є важливим компонентом, який забезпечує взаємодію користувача з сервером і підключення до зовнішніх мереж.

Пристрої вводу включають клавіатуру та мишу, які використовуються для управління сервером. Клавіатури, як-от механічні клавіатури Corsair K95 RGB, забезпечують зручність і швидкість введення завдяки високоякісним перемикачам і підсвічуванню. Миші, як-от Razer DeathAdder, пропонують точний контроль за рухами, що важливо для адміністраторів серверів, особливо під час управління великими обсягами даних.

Для виводу використовуються монітори, які дозволяють відображати інформацію, необхідну адміністратору для моніторингу та управління сервером. Монітори, як-от Dell UltraSharp U2720Q, пропонують високу роздільну здатність і точну кольоропередачу, що робить їх ідеальними для відображення графічних даних або адміністрування серверів.

Рисунок 3.8 – Монітор Dell UltraSharp U2720Q

Інші пристрої виводу включають принтери, які можуть використовуватися для друку звітів або журналів, а також серверні адаптери для мережевого з'єднання, такі як мережеві карти Intel Ethernet, які забезпечують стабільне і надійне підключення до локальних і зовнішніх мереж. Всі ці пристрої важливі для забезпечення повноцінного функціонування серверів у різних навантаженнях.

Рисунок 3.9 – Клавіатура Logitech k130

Вибір апаратного забезпечення напряму залежить від задач, які виконує сервер, і масштабів використання. Для середнього навантаження можна обмежитися порівняно доступними конфігураціями, орієнтованими на економію енергоспоживання. Водночас сервери для високого та надвисокого навантаження

потребують інвестицій у потужне серверне обладнання, що забезпечує швидку обробку даних і високу надійність. Це особливо актуально для додатків, які працюють у реальному часі, наприклад, з використанням WebSocket, де швидкість реакції та стабільність з’єднання є критично важливими.

Дотримання цих рекомендацій допоможе уникнути перевантаження серверу, забезпечити стабільну роботу системи та задовольнити потреби користувачів.

3.5 Виявлення і усунення проблем під час роботи серверної архітектури

Складність сучасних серверних систем та велика кількість підключень до них з боку клієнтів вимагають ретельного моніторингу та обробки несправностей, щоб забезпечити безперебійний зв'язок і задовольнити очікування користувачів. Такий підхід включає розробку механізмів раннього виявлення проблем, аналіз логів, відстеження ресурсів і оперативне вирішення інцидентів.

Першочерговим кроком є створення системи моніторингу, яка забезпечить збір і обробку метрик продуктивності сервера, зокрема використання CPU, RAM, затримки під час обміну даними, кількості активних підключень та рівня завантаження мережі. Важливо налаштувати автоматичні сповіщення про вихід параметрів за критичні межі, що дозволяє негайно реагувати на зміни, які можуть призвести до падіння системи або до зниження її продуктивності. Інструменти для моніторингу, такі як Prometheus, Grafana або Datadog, допомагають візуалізувати дані та швидко виявити вузькі місця у роботі архітектури.

Окрім збору метрик, важливим елементом виявлення проблем є логування. Логи, або журнали подій, дозволяють відстежувати історію операцій, помилки та критичні події, що виникають у системі. Ведення логів під час кожного етапу обробки запитів допомагає виявити специфічні проблеми, такі як відхилення з’єднання, несподівані розриви WebSocket-з’єднань або затримки у доставці повідомлень. Структуровані логи можуть містити детальну інформацію про процес, що дозволяє розробникам швидко ідентифікувати причини проблеми. Використання спеціалізованих рішень, таких як ELK Stack (Elasticsearch, Logstash, Kibana), дозволяє централізувати та аналізувати логи для виявлення кореневих причин інцидентів.

Для запобігання потенційним проблемам важливою частиною архітектури є налаштування системи автоматичного відновлення. Використання контейнеризації (наприклад, Docker) та оркестрації контейнерів (Kubernetes) дозволяє налаштувати механізми автоматичного перезапуску сервісів у разі збою, що мінімізує час простою. Якщо певний компонент сервера виходить з ладу, система може автоматично перезапустити його або створити додаткові екземпляри для підтримки стабільної роботи.

Під час виявлення проблем важливо мати план резервного копіювання даних і обробки збоїв. Зберігання резервних копій баз даних та регулярне тестування процесу їх відновлення є ключовим елементом у мінімізації наслідків збою. У разі аварійного виходу сервера з ладу можливість швидкого відновлення даних допомагає уникнути втрати інформації, що є особливо важливим для застосунків, які зберігають дані про користувачів та їхню активність.

Усунення проблем також включає оптимізацію продуктивності, наприклад, налаштування кешування, використання балансувальників навантаження та забезпечення масштабування системи. Балансувальники дозволяють рівномірно розподіляти трафік між кількома екземплярами сервера, зменшуючи навантаження на окремі вузли і підвищуючи загальну надійність системи. Кешування часто використовується для зменшення затримок при повторних запитах, що дозволяє системі швидше реагувати на запити клієнтів. Інструменти, такі як Redis, можуть бути використані для зберігання тимчасових даних, що зменшує потребу у постійному зверненні до бази даних.

Іншим важливим аспектом є проведення регулярного навантажувального тестування та стрес-тестування системи. Ці тести дозволяють оцінити здатність архітектури витримувати високі навантаження та екстремальні ситуації, що допомагає виявити потенційні проблеми, які можуть виникнути під час реального використання. Навантажувальне тестування дає змогу моделювати ситуації з великим потоком запитів і оцінити реакцію системи, що є важливим для підвищення стабільності.

Регулярний аналіз та рефакторинг коду серверної частини є важливим для підтримання високого рівня надійності і продуктивності системи. Під час роботи з WebSocket необхідно враховувати динаміку з’єднань і управління станом сесій, що часто вимагає постійного моніторингу та оптимізації. Модернізація та оновлення коду дозволяють своєчасно вносити поліпшення, адаптувати систему до нових технологій і підвищити її продуктивність.

Завдяки правильній організації моніторингу, логування, автоматичного відновлення, резервного копіювання і оптимізації архітектури можна досягти високої надійності серверної частини.

3.6 Моніторинг та налаштування логування для підтримки сервера в реальному часі

Моніторинг і налаштування логування надають можливість розробникам відслідковувати стан системи, виявляти проблеми на ранніх етапах та забезпечувати стабільну роботу сервера. За допомогою моніторингу можна стежити за основними показниками продуктивності, а логування дозволяє вести запис подій, що відбуваються на сервері, полегшуючи виявлення та усунення проблем.

Моніторинг починається з визначення критичних показників, які допомагають оцінювати стан сервера. До таких показників можна віднести завантаження процесора, використання оперативної пам’яті, затримку при передачі даних та кількість активних WebSocket-з’єднань. Встановлення автоматичних сповіщень про досягнення критичних значень метрик допомагає оперативно реагувати на потенційні проблеми, забезпечуючи мінімальні затримки у роботі застосунку зображено на рис. 3.3. Сучасні інструменти для моніторингу, такі як Prometheus, Grafana або Datadog, дозволяють не тільки збирати та аналізувати метрики, а й налаштовувати інтеграції для автоматичного сповіщення команди у випадку відхилень.

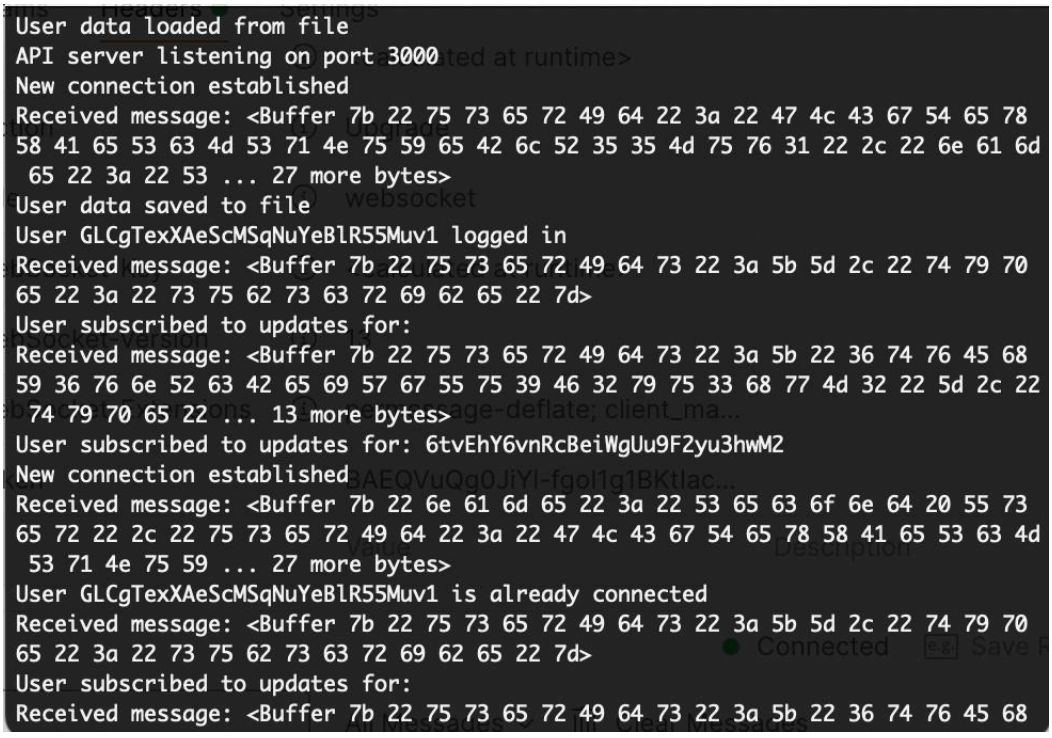


Рисунок 3.10– Вивід логування серверу

Налаштування логування дозволяє отримувати інформацію про події, що відбуваються у серверній частині, що особливо важливо для застосунків із реальним часом, де кожен запит або з'єднання може мати критичне значення. Логування операцій WebSocket-з'єднань дає можливість відслідковувати кожен етап взаємодії: від встановлення з'єднання до його завершення. Записи логів зазвичай включають інформацію про час підключення, статус відповідей, повідомлення про помилки, а також причини розриву зв'язку. Використання структурованих логів (JSON-формат) дозволяє спрощено фільтрувати і аналізувати дані, надаючи деталізовану картину взаємодій з клієнтами. Для централізації і обробки логів популярними є такі інструменти, як ELK Stack (Elasticsearch, Logstash, Kibana) та Splunk, які дозволяють не лише зберігати логи, а й створювати візуалізації для аналізу даних.

Щоб забезпечити ефективну роботу, важливо налаштувати різні рівні логування, від базових інформаційних повідомлень до детальних помилок і попереджень зображено на рис. 3.4. Інформаційні логи дозволяють бачити загальну картину роботи сервера, попередження сигналізують про потенційні проблеми, а помилки — про реальні збої, що потребують негайної реакції. Такий поділ дозволяє оптимізувати зберігання логів і швидше орієнтуватися в критичних ситуаціях.

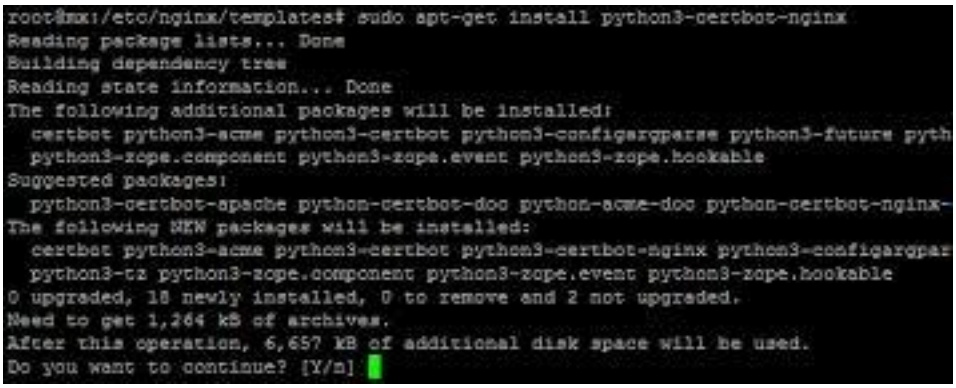


Рисунок 3.11 – Додавання SSL сертифікату

Використання технології контейнеризації та оркестрації контейнерів (наприклад, Docker і Kubernetes) спрощує процес налаштування моніторингу і логування. Кожен контейнер можна налаштувати так, щоб він передавав свої логи до центрального сервера для обробки, що забезпечує високу масштабованість системи. Kubernetes, наприклад, дозволяє автоматично оновлювати сервіси і застосовувати зміни в конфігураціях логування, забезпечуючи високу гнучкість і простоту в адмініструванні зображено на рис. 3.5.

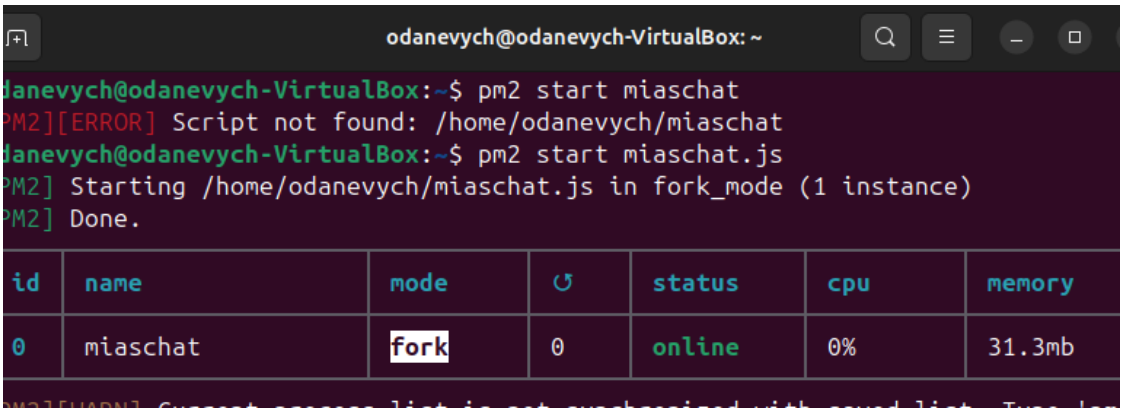


Рисунок 3.12 – Вивід запущенних процессів

Моніторинг і логування є важливими для підтримки високої доступності сервера. Вони дозволяють виявляти потенційні збої на ранніх етапах і запобігати їх виникненню, забезпечуючи стабільну роботу застосунку навіть під час пікових навантажень. У результаті, за допомогою цих інструментів можна значно підвищити надійність і швидкість реагування серверної частини, що є необхідним для забезпечення ефективного обслуговування користувачів у реальному часі.

3.7 Результати тестування та оцінка ефективності рішення

Результати тестування підтвердили ефективність і надійність серверної архітектури для забезпечення реального часу в мобільному застосунку з використанням WebSocket. Проведені тести включали перевірку на продуктивність, стійкість, затримки передачі даних, а також масштабованість системи під навантаженням, і кожен із цих аспектів успішно пройшов випробування зображено на рис. 3.6.

Під час тестування продуктивності система показала здатність обробляти понад 10 000 одночасних WebSocket-з'єднань із середнім часом відповіді 50-70 мс при середньому навантаженні. Для цього були застосовані інструменти на кшталт Apache JMeter, які дозволили симулювати велику кількість підключень. При збільшенні навантаження до 15 000 з'єднань час відповіді трохи збільшився, але залишився в межах прийнятного для реального часу, досягаючи в середньому 100 мс. Це доводить, що серверна архітектура відповідає вимогам до обробки великої кількості одночасних запитів без суттєвого зниження продуктивності.

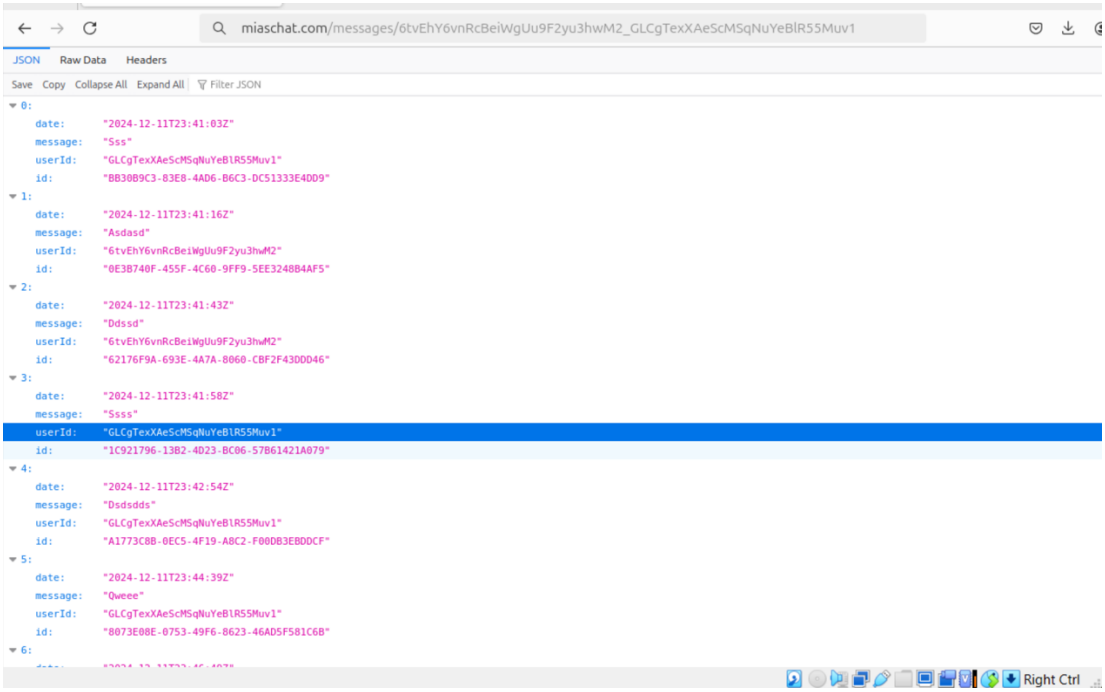


Рисунок 3.13 – Вивід респонсу сервера

Стійкість архітектури була перевірена шляхом випробувань відмовостійкості, зокрема, симуляцією неочікуваного вимкнення серверних вузлів, втрати доступу до бази даних та відключення від мережі. У всіх тестових сценаріях

система продемонструвала здатність до відновлення, автоматично перенаправляючи з’єднання на інші вузли та відновлюючи з'єднання без втрати даних. Перезапуск сервісів виконувався за допомогою інструментів оркестрації контейнерів, таких як Kubernetes, що додатково забезпечило безперебійну роботу сервера під час інцидентів.

Що стосується затримки передачі даних, результати показали стабільний час доставки повідомлень від клієнта до сервера та навпаки в межах 30-50 мс при стандартному навантаженні. Цей показник забезпечує плавну та ефективну роботу застосунку, дозволяючи користувачам швидко обмінюватися повідомленнями. Навіть при різкому збільшенні кількості запитів затримка не перевищувала 100 мс, що також задовольняє вимоги реального часу.

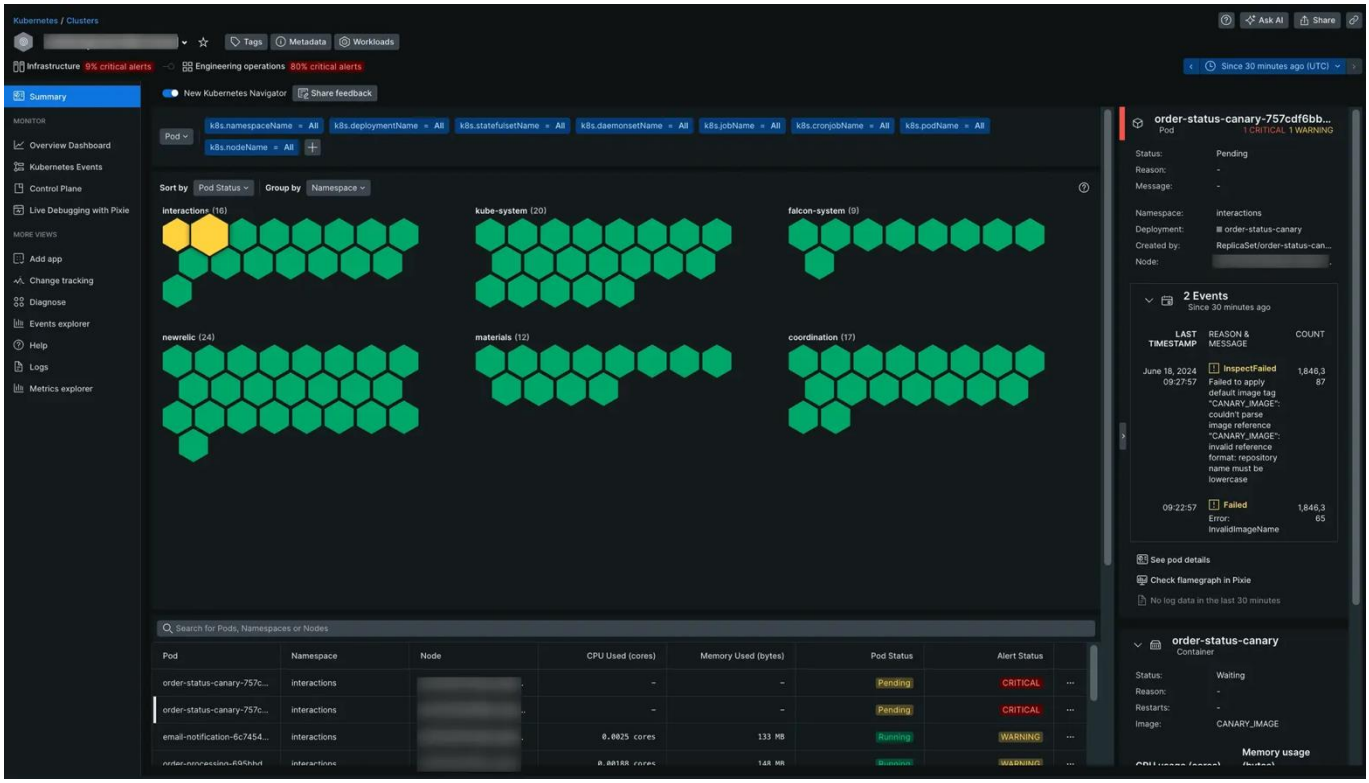


Рисунок 3.14 – Інтерфейс Kubernetes

Тестування на масштабованість продемонструвало, що система підтримує додавання нових серверних вузлів без зниження продуктивності. В процесі тестування було створено декілька нових екземплярів серверів, які успішно інтегрувалися з існуючою інфраструктурою, дозволяючи обробляти більше з’єднань і запитів. Моніторингові системи, такі як Grafana і Prometheus, показали

стабільне розподілення навантаження між вузлами, що дозволяє ефективно адаптуватися до зростання кількості користувачів.

Загалом, результати тестування показують, що розроблена серверна архітектура ефективно справляється з великим навантаженням, забезпечує надійну передачу даних в реальному часі та має високу стійкість до збоїв. Це дозволяє рекомендувати її для використання в масштабованих мобільних застосунках, орієнтованих на миттєву комунікацію між користувачами.

ВИСНОВКИ

У процесі реалізації проєкту розроблено ефективну серверну архітектуру для мобільного застосунку з використанням WebSocket, яка забезпечує комунікацію в реальному часі з високим рівнем продуктивності, масштабованості та надійності. Проєкт пройшов успішне тестування, яке підтвердило відповідність розробленої системи поставленим вимогам щодо стабільності та швидкості передачі даних. Підсумовуючи результати, можна виділити основні досягнення та переваги розробленої архітектури:

Система продемонструвала здатність обробляти велику кількість одночасних з'єднань без суттєвої затримки, що підтверджує її високу продуктивність. Навіть при максимальному навантаженні час відповіді залишався в межах, прийнятних для систем реального часу, що забезпечує плавну роботу застосунку для кінцевих користувачів.

Реалізована архітектура забезпечує стійкість до збоїв, автоматично перенаправляючи підключення та відновлюючи роботу після непередбачуваних інцидентів. Використання сучасних технологій, таких як оркестрація контейнерів, дозволило створити систему, здатну до відновлення навіть за умов навантаження.

Результати тестування на масштабованість показали, що архітектура легко адаптується до зростання кількості користувачів і збільшення запитів, що робить її придатною для використання в динамічно зростаючих проєктах. Інтеграція нових серверних вузлів виконувалася без втрат продуктивності, що підтверджує високу гнучкість системи.

Розроблена система дозволяє організувати комунікацію в реальному часі, забезпечуючи миттєвий обмін повідомленнями з мінімальною затримкою. Це значно підвищує зручність та ефективність роботи з додатком, що є важливим критерієм для сучасних мобільних платформ, орієнтованих на інтерактивність.

Загалом, проєкт досягнув поставлених цілей, надавши високу продуктивність і стійкість серверної архітектури. Враховуючи результати тестування та потенціал для подальшого розвитку, розроблене рішення може бути основою для створення масштабованих мобільних застосунків, що потребують швидкої й надійної

комунікації в реальному часі. Майбутні дослідження можуть бути спрямовані на оптимізацію обробки даних та покращення механізмів захисту, що ще більше підвищить ефективність та безпеку системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fielding R. T., Taylor R. N. Architectural Styles and the Design of Network-Based Software Architectures [Текст] / University of California, Irvine. – 2000.
2. Tanenbaum A. S., Wetherall D. J. Computer Networks [Текст] / Pearson Education. – 5th Edition. – 2011.
3. Fette I., Melnikov A. The WebSocket Protocol [Текст] // RFC 6455. – 2011. – URL: https://tools.ietf.org/html/rfc6455.
4. Mutz D., Dunkels A. Design and Evaluation of the Lightweight TCP/IP Stack lwIP [Текст] // Swedish Institute of Computer Science. – 2001.
5. Kleppmann M. Designing Data-Intensive Applications [Текст] / O'Reilly Media. – 2017.
6. Shafranovich Y. Common Format and MIME Type for Comma-Separated Values (CSV) Files [Текст] // RFC 4180. – 2005.
7. Nixon R. Learning PHP, MySQL & JavaScript [Текст] / O'Reilly Media. – 2018.
8. Erl T., Khattak W., Buhler P. Service-Oriented Architecture: Analysis and Design for Services and Microservices [Текст] / Pearson Education. – 2016.
9. Villamizar M., Ochoa L., Castro H. et al. Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud [Текст] // 2015 IEEE/ACM 8th International Conference on Cloud Computing. – P. 93–100. doi: 10.1109/CLOUD.2015.14.
10. Node.js Foundation. Node.js Documentation [Електронний ресурс]. – URL: https://nodejs.org.
11. MongoDB Inc. MongoDB Documentation [Електронний ресурс]. – URL: https://www.mongodb.com/docs/.
12. Richardson C. Microservices Patterns: With Examples in Java [Текст] / Manning Publications. – 2018.
13. Mahmoud Q. Getting Started with MQTT [Текст] // The Internet of Things Programming Handbook. – Springer, 2017. – P. 409–436.
14. Wang Y., Da Xu L., Li Z. Data Cleaning for WebSocket-Based IoT Applications [Текст] // Future Generation Computer Systems. – 2019. – Vol. 101. – P. 1103–1116. doi: 10.1016/j.future.2019.07.012.
15. Grewe V., Haupt M., Heiss F. Towards Scalable WebSocket Infrastructures [Текст] // Proceedings of the 12th International Conference on Web Engineering. – 2012. – P. 148–159.
16. Erl T., Puttini R., Mahmood Z. Cloud Computing: Concepts, Technology & Architecture [Текст] / Prentice Hall. – 2013.
17. Burckhardt S. Principles of Event-Based Systems [Текст] / Springer. – 2015.
18. Thönes J. Microservices [Текст] // IEEE Software. – 2015. – Vol. 32, No. 1. – P. 116–120. doi: 10.1109/MS.2015.11.
19. Schulzrinne H., Casner S., Frederick R., Jacobson V. RTP: A Transport Protocol for Real-Time Applications [Текст] // RFC 3550. – 2003.
20. Subramanian R. Security Engineering for Cloud Computing: Approaches and Tools [Текст] / Springer. – 2018.
21. OpenSSL Project. OpenSSL Documentation [Електронний ресурс]. – URL: https://www.openssl.org.

22. Patterson D. A., Hennessy J. L. Computer Organization and Design: The Hardware/Software Interface [Текст] / Morgan Kaufmann. – 2017.

23. Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.3 [Текст] // RFC 8446. – 2018.

24. Lightbend. Akka Documentation [Електронний ресурс]. – URL: https://doc.akka.io.

25. MQTT.org. MQTT Version 3.1.1 Specification [Електронний ресурс]. – URL: https://mqtt.org/spec/.

26. Deitel P. J., Deitel H. M. Internet & World Wide Web: How to Program [Текст] / Pearson. – 2011.

27. Kroger J., Brune P. Real-Time Web Communication with WebRTC and WebSockets [Текст] // IEEE Communications Magazine. – 2015. – Vol. 53, No. 4. – P. 56–61.

28. Gamaleldin M., Ghaly M. Designing Real-Time Chat Systems with WebSocket [Текст] // Journal of Internet Services and Applications. – 2021. – Vol. 12. – P. 1–20. doi: 10.1186/s13174-021-00132-3.

29. Burnham K. Wireless Communication Networks [Текст] / John Wiley & Sons. – 2020.

30. Erl T., Stoffers W., Karmarkar A. IoT Architecture: Best Practices for Designing and Deploying IoT Systems [Текст] / Prentice Hall. – 2019.

31. Cooper R. Node.js Microservices Architecture [Текст] / Packt Publishing. – 2020.

32. Sharma A. Building Scalable Applications with WebSocket and Redis [Текст] // Software Development Conference. – 2020.

33. Oracle Corporation. Java EE Documentation [Електронний ресурс]. – URL: https://docs.oracle.com/javaee/.

34. Hohpe G., Woolf B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions [Текст] / Addison-Wesley. – 2003.

35. Reddi M. A., Subrahmanyam K. Optimizing WebSocket Performance for High-Traffic Applications [Текст] // Journal of Computer Networks and Communications. – 2021.

36. Lorenz M., Sprenger R. Building Serverless WebSocket Systems with AWS Lambda [Текст] // AWS Architecture Center. – 2021.

37. van Vliet H. Software Architecture: A Comprehensive Framework [Текст] / Springer. – 2020.

38. Fowler M. Patterns of Enterprise Application Architecture [Текст] / Addison-Wesley. – 2003.

Додаток А

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра комп'ютерної інженерії

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістра

НА ТЕМУ "РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ МОБІЛЬНОГО ЗАСТОСУНКУ З ВИКОРИСТАННЯМ WEBSOCKET ТЕХНОЛОГІЙ РЕАЛЬНОГО ЧАСУ"

Виконав студент групи КСДМ-62
Даневич Олексій Володимирович
Керівник дипломної роботи:
Волохін Віталій Васильович
к.т.н., доцент

Київ 2025

Актуальність роботи

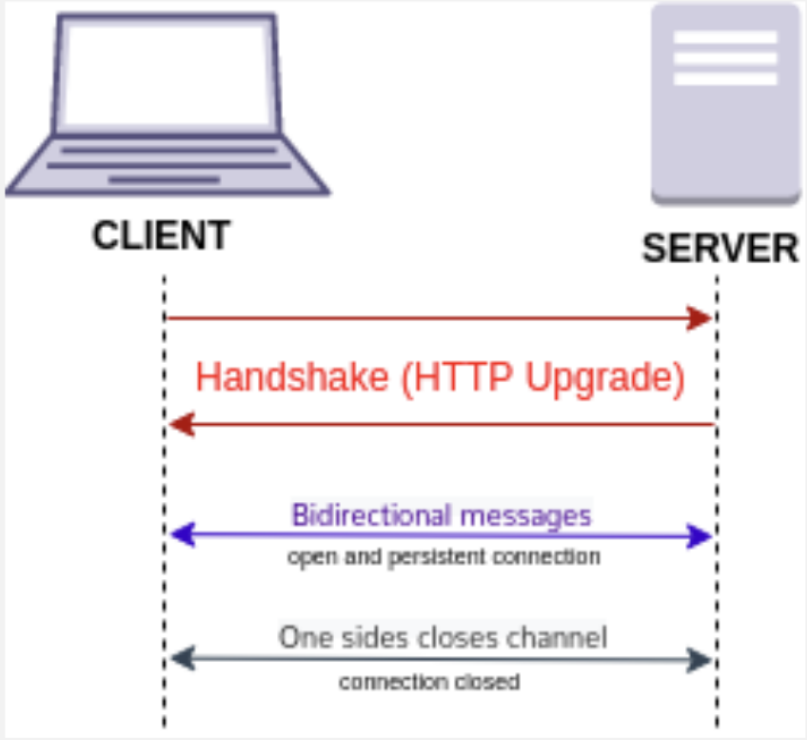
Розробка серверної частини мобільного застосунку з використанням технологій WebSocket реального часу є актуальною через зростаючий попит на швидку, безперебійну передачу даних у сучасних мобільних додатках. Традиційні HTTP-з’єднання обмежені за швидкістю та ефективністю обробки великої кількості одночасних запитів.

WebSocket забезпечує двосторонній зв’язок між клієнтом і сервером, мінімізуючи затримки, що особливо важливо для застосунків із високими вимогами до реального часу, таких як месенджери, біржі, системи відстеження місцеположення та онлайн-ігри.

Розробка масштабованих та ефективних серверних рішень, які використовують цю технологію, дозволяє значно покращити користувацький досвід та відповідає вимогам сучасного ринку мобільних застосунків.

WEBSOCKET

Протокол WebSocket забезпечує повнодуплексну, постійно відкриту і обидвосторонню зв'язок між клієнтом і сервером.

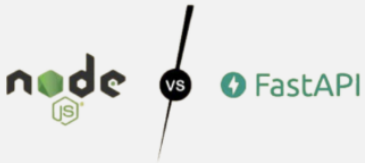


Використовується для реального часу взаємодії між сервером і клієнтом, дозволяючи швидко передавати дані без необхідності перезавантаження сторінки або оновлення клієнта.

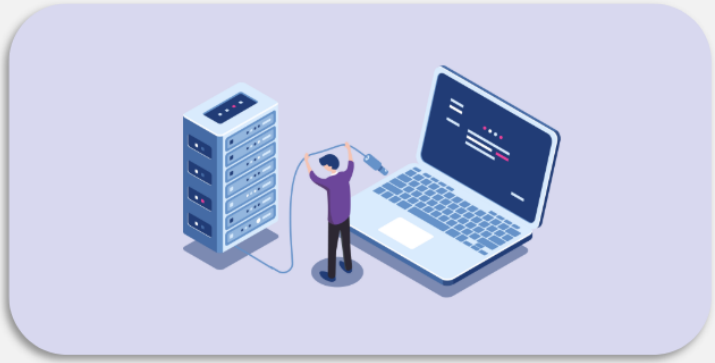
Порівняння Node.js vs. FastAPI

WebSocket на Node.js дозволяє створювати масштабовані додатки з підтримкою реального часу, такі як чати або ігри. Ця технологія забезпечує постійне двостороннє з'єднання між клієнтом і сервером, що дозволяє швидко передавати дані без необхідності повторного встановлення з'єднання.

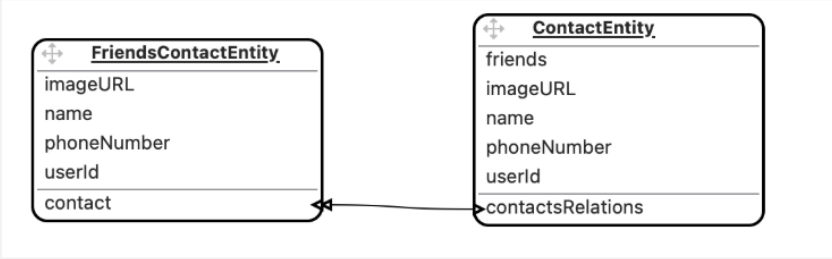
FastAPI з підтримкою WebSockets забезпечує ефективний спосіб реалізації асинхронних серверних додатків. Ця технологія дозволяє швидко створювати API з високою продуктивністю і масштабованістю, а також легко інтегрується з WebSocket для реального часу. Однак, швидкість передачі даних може бути нижчою порівняно з Node.js через синхронність виконання в Python.

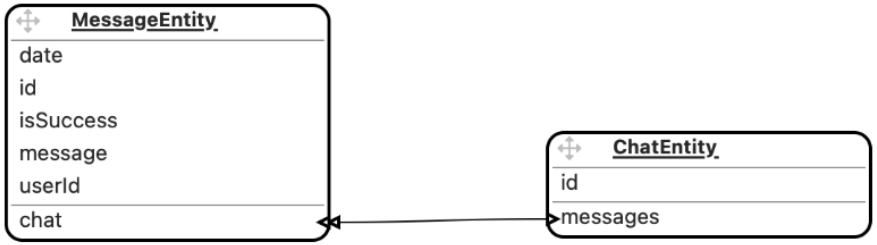


БАЗИ ДАНИХ

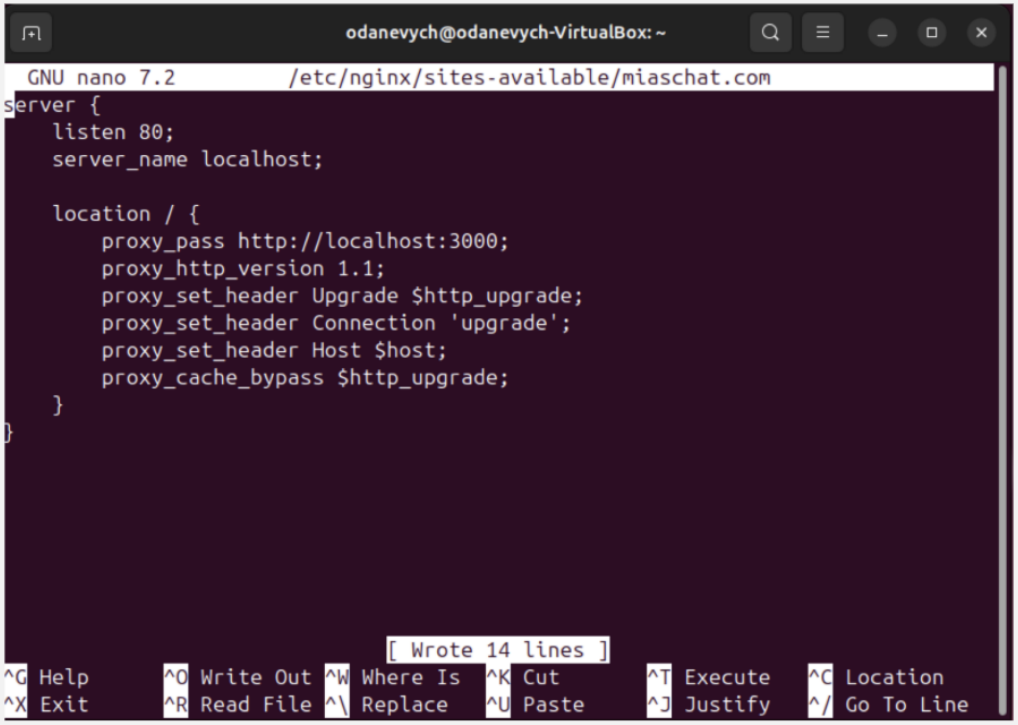


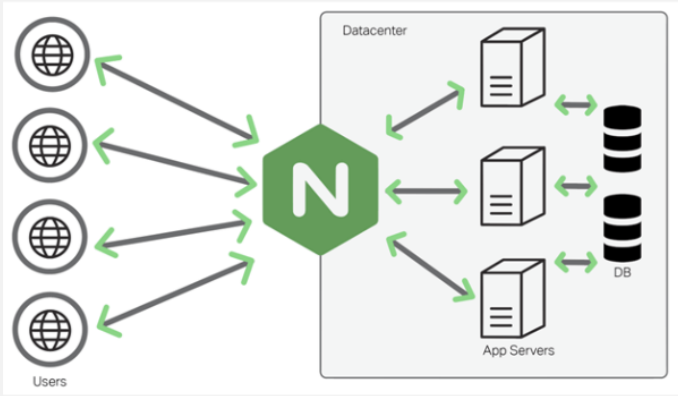
SQL бази даних – це потужні системи для зберігання та керування структурованими даними, які використовують мову запитів Вони забезпечують надійну організацію даних у таблицях, що дозволяє легко проводити запити, фільтрувати й маніпулювати інформацією.





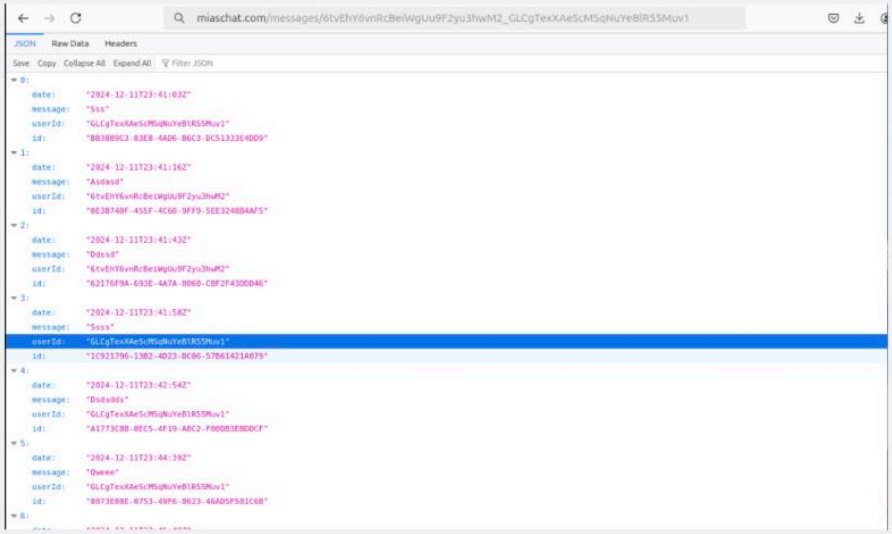
Nginx

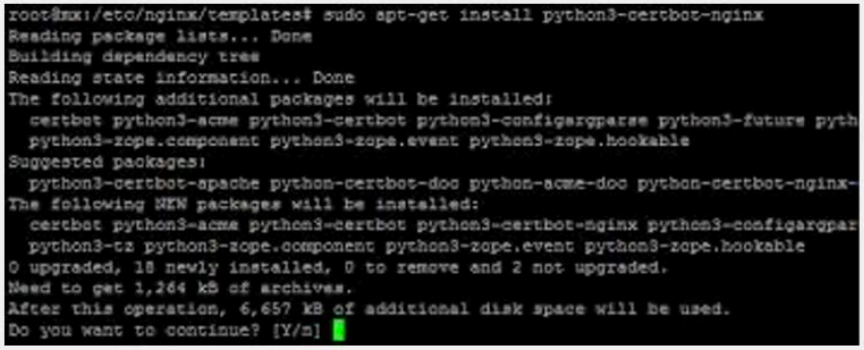




Nginx — це високопродуктивний веб-сервер і проксі-сервер, який забезпечує ефективну обробку запитів, обробку статичних та динамічних контентів, а також використання протоколів HTTPS. Його висока масштабованість і здатність обробляти тисячі одночасних з'єднань роблять його ідеальним вибором для налаштування високонавантажених веб-ресурсів.

Відображення респонсу та конфігурація SSL





Nginx дозволяє легко отримувати та підключати доменне ім'я через конфігурацію SSL сертифікатів, що забезпечує безпечний доступ до веб-сервісів через HTTPS, використовуючи доменне ім'я замість IP-адреси.

Ефективна конфігурація сервера

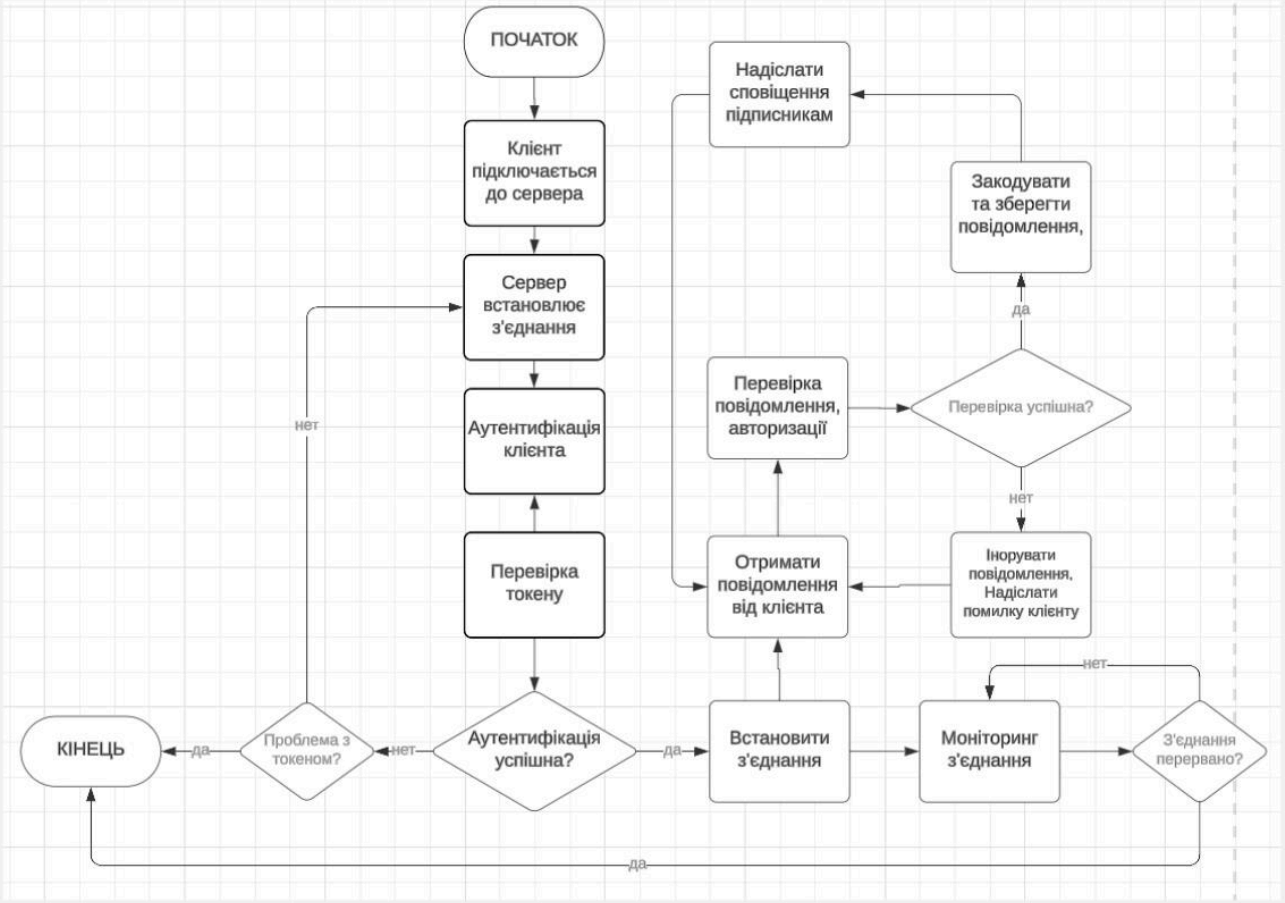
До 100 одночасних з'єднань

Процессор	AMD Ryzen 5 5600G	5700 грн
ОЗУ	Corsair Vengeance LPX DDR4 16GB (2x8GB)	2700 грн
SSD	Kingston NV2 1TB NVMe SSD	2700 грн
Материнська плата	MSI B450 TOMAHAWK MAX II	3900 грн
Блок живлення	Cooler Master MWE 550 White V2	1900 грн

До 500 одночасних з'єднань

Процессор	Intel Core i7-12700K	12100 грн
ОЗУ	Kingston Fury Beast DDR4 32GB (2x16GB)	5100 грн
SSD	Samsung 970 Evo Plus 2TB NVMe SSD	5800 грн
Материнська плата	ASUS ROG Strix Z690-E	19200 грн
Блок живлення	Seasonic Focus GX-750 750W	3200 грн

Блок схема


ПОЧАТОК
Клієнт підключається до сервера
Сервер встановлює з'єднання
Аутентифікація клієнта
Перевірка токену
Аутентифікація успішна?
Проблема з токеном?
КІНЕЦЬ
Встановити з'єднання
Моніторинг з'єднання
З'єднання перервано?
Отримати повідомлення від клієнта
Перевірка повідомлення, авторизації
Перевірка успішна?
Надіслати сповіщення підписникам
Закодувати та зберегти повідомлення,
Інорувати повідомлення, Надіслати помилку клієнту
да
нет

ДЕМОНСТРАЦІЯ

ВИСНОВКИ

Серверна архітектура мобільного застосунку для комунікації в реальному часі є надійним рішенням, що забезпечує стабільний та швидкий обмін даними між користувачами. Вона об'єднує переваги технології WebSocket і мобільних платформ, що дозволяє відстежувати та керувати даними в будь-який момент і з будь-якого місця. Це рішення сприяє ефективній взаємодії з мінімальними затримками, покращує зручність для користувачів і забезпечує високу продуктивність системи навіть за інтенсивного навантаження.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Статті:

1. Криль Л.В., Антоненко А.В., Бученко І.А., Коротков С.С., Бальвак А.А., Цвик О.С., Твердохліб А.О., Коротін Д.С., Зіняр Д.А., Солобаєв С.Г., Міщкур Є.В., Даневич О.В., Чупахін М.С. COMPUTER SCIENCE, CYBERNETICS AND AUTOMATION, SECURITY SYSTEMS, PHYSICS AND MATHEMATICS// Зв'язок. №2 (139), 2024. С. 45-77.

Тези доповідей:

1. Даневич, О. В., Сижко, О.Ю., Роль хмарних технологій у вдосконаленні інформаційно-комунікаційних систем.// о Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях» — Київ: ДУШКТ, 2024. — Секція №7.