МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІМ. Ю.М. ПОТЕБНІ ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Використання ReactJs і NodeJs як єдиної платформи для розробки розподілених додатків**

Виконав: студент 2 курсу, групи 8.1212-іпз-2
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
А.В. Сахарова
(ініціали та прізвище)
Керівник доцент, к.т.н. В. І. Заяц
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Алтер Віжн Груп»

В.С. Тряпічко
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІМ. Ю.М. ПОТЕБНІ
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра Електроніки, інформаційних систем та програмного забезпечення

Рівень вищої освіти _____другий (магістерський)_____

Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)

Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_____Т. В. Критська
“_01_” __вересня_____2023_року

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

Сахаровій Аліні Валеріївні
(прізвище, ім’я, по батькові)

1.Тема роботи __Використання ReactJs і NodeJs як єдиної платформи для розробки розподілених додатків

керівник роботи Заєць Валерій Іванович, доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від ___02.06.2023 р. №597-с

2. Строк подання студентом кваліфікаційної роботи 1 грудня 2023 р.

3. Вихідні дані магістерської роботи

- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- дослідження проблеми створення розподілених додатків;
- створення програмного продукту та його опис;
- перелік вимог для роботи програми;
- дослідження поставленої проблеми та розробка висновків та пропозицій.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

10 слайдів презентації

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 01.09.2023

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Аналіз предметної області. Огляд існуючих рішень у сфері розподілених додатків	02.09-10.09.23	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	11.09-12.09.23	виконано
3	Дослідження фреймворку ReactJs та бібліотек для роботи з ним	13.09-14.09.23	виконано
4	Розробка загальної архітектури системи велосипедних змагань	15.09-20.09.23	виконано
5	Реалізація основних функцій системи велосипедних змагань	21.09-26.09.23	виконано
6	Тестування системи на швидкість реагування з RFID зчитувачами та Arduino	27.09-28.09.23	виконано
7	Реалізація клієнтської і серверної частини веб-застосунку	29.09-13.10.23	виконано
8	Реалізація мобільного та десктопного застосунку	14.10-16.10.23	виконано
9	Тестування загальної роботи системи	17.10-19.10.23	виконано
10	Демонстрація результатів реалізації системи науковому керівнику та узгодження етапів дослідження	20.10-09.11.23	виконано
11	Проведення дослідження системи на реальних велосипедних змаганнях	10.11-17.11.23	виконано
12	Опис результатів дослідження	18.11-22.11.23	виконано
13	Оформлення звіту	23.11-27.11.23	виконано

Студент _____ А.В. Сахарова
(підпис) (прізвище та ініціали)

Керівник роботи _____ В.І. Заяц
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ І.А. Скрипник
(підпис) (прізвище та ініціал)

АНОТАЦІЯ

Сторінок: 80

Рисунків: 17

Джерел: 36

Сахарова А. В. Використання ReactJs і NodeJs як єдиної платформи для розробки розподілених додатків : кваліфікаційна робота магістра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник В. І. Заєць. Запоріжжя : ЗНУ, 2023. 80 с.

Мета і завдання дослідження полягають у вивченні вивчення можливостей та переваг використання фреймворків ReactJS і NodeJS як єдиної платформи для розробки розподілених додатків. Основною метою є аналіз інтеграції цих технологій для забезпечення ефективного взаємодії між клієнтською та серверною частинами, а також для підтримки масштабованості та гнучкості додатків. Дослідження охоплює аспекти розробки інтерфейсу користувача, логіки бізнес-процесів і взаємодії з базою даних на основі цих технологій. В результаті вивчення буде розроблено та проаналізовано розподілену систему, де ReactJS та NodeJS використовуються як центральна основа для ефективної та модульної розробки.

Ключові слова: ReactJS, NodeJS, розподілені додатки, веб-розробка, інтеграція технологій.

SUMMARY

Pages: 80

Figures: 17

Sources: 36

Sakharova A.V. Using ReactJs and NodeJs as a single development platform: qualification work of the master of specialty 121 "Software Engineering" / Science head V.I. Zaiatz. Zaporizhzhia : ZNU, 2023. 80 p.

The aim and objectives of the research are to explore the possibilities and advantages of using ReactJS and NodeJS frameworks as a unified platform for developing distributed applications. The main goal is to analyze the integration of these technologies to ensure effective interaction between the client and server components, as well as to support scalability and flexibility of applications. The research covers aspects of user interface development, business logic, and database interaction based on these technologies. As a result of the study, a distributed system will be developed and analyzed, where ReactJS and NodeJS serve as a central foundation for efficient and modular development.

Keywords: ReactJS, NodeJS, distributed applications, web development, technology integration.

ЗМІСТ

ВСТУП

Актуальність теми

У сучасному світі, розробка розподілених додатків стає все важливішою завданням, оскільки додатки повинні бути швидкими, масштабованими та здатними до ефективної взаємодії з користувачами. Використання ReactJS та Node.js як єдиної платформи надає уніфіковану архітектуру, що сприяє розробці розподілених додатків, використовуючи єдину мову програмування та ефективний обмін даними між клієнтом та сервером. У даній дослідженні розглядаються переваги такого підходу та його вплив на розробку розподілених додатків.

Використання ReactJS та Node.js дозволяє розробляти клієнтську та серверну частини додатків, використовуючи однакові мови програмування (JavaScript/TypeScript), спрощуючи комунікацію та підтримку проекту. Однакова мова програмування спрощує впровадження змін та розширення функціональності додатку, оскільки розробники можуть легко розуміти і модифікувати як клієнтську, так і серверну частини, що сприяє швидкому виправленню помилок та розвитку проекту. Крім того, використання ReactJS та Node.js дозволяє легко обмінювати компоненти та бібліотеки між клієнтською та серверною частинами, що сприяє створенню модульних та перевикористовуваних рішень для розподілених додатків. Уніфікований стек спрощує підтримку єдиного кодового стилю та стандартів розробки на всьому проекті, полегшуючи розуміння коду та знижуючи ймовірність виникнення конфліктів та невідповідностей у проекті.

Розподілені додатки стають стандартом у світі сучасної розробки, оскільки вони підвищують масштабованість та надійність систем. Використання ReactJS та Node.js як ключових компонентів для розробки розподілених додатків відкриває нові можливості для створення мікросервісної архітектури. ReactJS надає зручні інструменти для створення інтерфейсів користувача та клієнтської логіки, дозволяючи розробникам створювати інтерактивні та ефективні інтерфейси. Одночасно, Node.js забезпечує швидкість та надійність серверної частини, дозволяючи обробляти запити та здійснювати обмін даними з іншими сервісами. Ця

комбінація робить ReactJS та Node.js ідеальними для створення розподілених додатків, де кожен сервіс може бути розгляданий як незалежний модуль. Ця архітектурна модель дозволяє командам розробників працювати над окремими сервісами паралельно, поліпшуючи ефективність розробки та масштабованість.

Використання ReactJS та Node.js як єдиної платформи для розробки розподілених додатків відкриває безліч можливостей для розробників. Уніфікована технологічна стекова архітектура дозволяє створювати компоненти як на клієнтській, так і на серверній стороні з використанням однієї мови програмування, спрощуючи розробку та підтримку проекту. Реактивний підхід ReactJS та швидкість Node.js допомагають розробникам створювати продуктивні та масштабовані додатки. Інтеграція з іншими технологіями та підтримка великої спільноти розробників роблять ReactJS та Node.js цікавими інструментами для розробки розподілених додатків.

Мета і завдання дослідження

Мета і завдання дослідження полягають у вивченні та оцінці можливостей використання React.js і Node.js як єдиної платформи для розробки розподілених додатків. Головною метою є визначення ефективності цих технологій у створенні розподілених додатків, а також встановлення практичних рекомендацій і кращих практик для їх використання. Завдання включають аналіз переваг та обмежень, дослідження впливу на продуктивність та безпеку, порівняльний аналіз з іншими технологіями та надання конкретних прикладів використання React.js і Node.js для розробки розподілених додатків.

Об’єкт дослідження

Об’єктом дослідження є розробка розподілених додатків.

Предмет дослідження

Предметом дослідження є використання React.js і Node.js як єдиної платформи для розробки розподілених додатків.

Методи дослідження

Для вирішення поставленої задачі використовуються наступні методи дослідження:

1. Дослідження і аналіз наукових статей, книг, документації і онлайн-ресурсів, що стосуються використання React.js і Node.js в розподіленому програмуванні.
2. Опитування або інтерв'ювання експертів у сфері веб-розробки, які мають досвід використання React.js і Node.js.
3. Проведення практичних експериментів та розробка розподілених додатків з використанням React.js і Node.js для подальшої оцінки їх продуктивності, масштабованості та безпеки.
4. Порівняння використання React.js і Node.js з іншими технологіями для розробки розподілених додатків щодо їх переваг та недоліків.
5. Розгляд реальних випадків використання React.js і Node.js для створення розподілених додатків та аналіз їхнього успіху.
6. Збір та аналіз відгуків користувачів розподілених додатків на React.js і Node.js.
7. Аналіз всіх зібраних даних та результатів дослідження з метою сформулювати висновки щодо ефективності та придатності використання React.js і Node.js для розробки розподілених додатків. Висновки повинні відображати переваги та недоліки цих технологій, а також надавати рекомендації для їхнього використання в практиці.

Наукова новизна одержаних результатів

Наукова новизна отриманих результатів полягає у виявленні і аналізі можливостей використання React.js і Node.js як єдиної платформи для розробки розподілених додатків, включаючи аналіз їхніх переваг та недоліків, впливу на продуктивність та безпеку, порівняльний аналіз з іншими технологіями і надання рекомендацій для практичного застосування. Ці результати можуть бути

корисними для інженерів і розробників програмного забезпечення, що цікавляться розподіленим програмуванням і шукають оптимальні інструменти для розробки розподілених додатків.

Практичне значення одержаних результатів

Практичне значення отриманих результатів полягає у тому, що вони можуть служити джерелом інформації та рекомендацій для розробників і інженерів програмного забезпечення, які цікавляться розподіленим програмуванням. Отримані висновки та аналіз можуть допомогти їм вибирати найбільш оптимальні інструменти та підходи при створенні розподілених додатків, що в свою чергу впливає на підвищення продуктивності, масштабованості та безпеки їхніх проектів. Ці результати мають практичне застосування в сфері розробки програмного забезпечення та можуть допомогти покращити процес розробки розподілених додатків, що важливо для бізнесу і конкурентоспроможності на ринку.

Апробація одержаних результатів

Результати роботи, викладені у кваліфікаційній роботі магістра, були опубліковані на XVI університетській науково-практичній конференції студентів, аспірантів, докторантів і молодих вчених — «Молода наука-2023» [12], а також на III Всеукраїнській науково-практичній конференції за участю молодих науковців — «Актуальні питання сталого науково-технічного та соціально-економічного розвитку регіонів України» [13].

Глосарій

Виявлення ключових точок людини (англ. Keypoint detection) — це проблема розпізнавання людини та виявлення ключових точок її тіла (ніг, плечей, рук тощо).

React.js — JavaScript-бібліотека для розробки інтерфейсу користувача, яка дозволяє створювати компоненти і відображати їх на веб-сторінках.

Node.js — Виконавче середовище для JavaScript, яке дозволяє розробникам створювати серверну частину додатків з використанням JavaScript.

Розподілені додатки (англ. Distributed Applications) — Додатки, які складаються з різних компонентів або модулів, розміщених на різних серверах або в середовищі з розподіленою архітектурою.

Єдина платформа (англ. Single Platform) — Використання React.js і Node.js як єдиної технологічної платформи для розробки всієї структури додатка, включаючи клієнтську та серверну сторони.

Компонентна архітектура (англ. Component-Based Architecture) — Підхід до розробки, де додаток розбивається на компоненти, які можуть бути незалежно розробленими та підтримуваними частинами коду.

API (Application Programming Interface) — Інтерфейс, який дозволяє взаємодіяти з іншими частинами програми або зовнішніми сервісами.

Масштабованість (англ. Scalability) — Здатність додатка працювати ефективно при збільшенні обсягів даних та навантаження.

Мікросервіси (англ. Microservices) — Архітектурний підхід, при якому додаток розбивається на невеликі, незалежні компоненти, які можуть бути розгорнуті окремо.

Додатки в реальному часі (англ. Real-Time Applications) — Додатки, які здатні обробляти та відображати дані миттєво, без затримок, що дозволяє взаємодіяти з користувачами у реальному часі.

Управління станом (англ. State Management) — Процес визначення та керування станом додатку, який включає в себе дані та їх зміни впродовж часу.

Рендерінг на стороні сервера (англ. Server-Side Rendering) — Підхід, при якому відображення сторінки генерується на сервері перед відправкою клієнту, що поліпшує SEO та продуктивність.

Рендерінг на стороні клієнта (англ. Client-Side Rendering) — Підхід, при якому відображення сторінки відбувається на браузері користувача, що зменшує навантаження на сервер.

Віртуальний DOM (англ. Virtual DOM) — Віртуальне представлення структури інтерфейсу користувача в React, що допомагає оптимізувати оновлення та відображення змін.

Проміжний програмний рівень (англ. Middleware) — Програмний код, який обробляє запити між клієнтом та сервером, дозволяючи розширювати функціональність додатку.

РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ СТВОРЕННЯ РОЗПОДІЛЕНИХ ДОДАТКІВ ЄДИНОЇ СИСТЕМИ

1.1 Огляд проблеми створення розподілених додатків для єдиної системи

Розробка розподілених додатків для єдиної системи є складним завданням, яке вимагає уваги до ряду ключових аспектів, щоб забезпечити ефективну та надійну роботу системи. Розподілені системи, що базуються на архітектурі з розділеними компонентами, стикаються з численними викликами.

Як зауважує у своїй книзі Ендрю Таненбаум [22], правильне проектування є дуже важливим аспектом при створенні розподілених застосунків, бо такий підхід допоможе уникнути найбільших проблем при розробці.

Створення розподілених додатків для єдиної системи включає в себе ряд важливих викликів і проблем, які пов'язані з розподіленою архітектурою та взаємодією компонентів. Ось кілька ключових аспектів, які можуть виникнути при розробці таких додатків:

1. Комунікація між компонентами:

- Мережева комунікація: Розподілені додатки взаємодіють через мережу, і це може призводити до затримок і втрат пакетів. Налаштування ефективної мережевої комунікації та обробка можливих помилок є важливими аспектами.
- Протоколи взаємодії: Визначення правильних протоколів для обміну даними між різними частинами системи допомагає у вирішенні проблем синхронізації та забезпеченні цілісності даних.

2. Масштабованість:

- Горизонтальне та вертикальне масштабування: Вирішення питань масштабування, як горизонтального (додавання нових екземплярів або вузлів) так і вертикального (збільшення потужності окремих вузлів), дозволяє забезпечити високу продуктивність та ефективність роботи системи.

3. Управління станом і консистентність даних:

- Транзакції та консистентність: Забезпечення консистентності даних у розподіленій системі в умовах розподіленості та можливості виникнення збоїв є складною задачею. Використання транзакцій та розподіленої блокування може допомогти у вирішенні цих питань.

4. Безпека та автентифікація:

- Захист даних: Забезпечення безпеки даних, передачі і зберігання, включаючи шифрування та контроль доступу, є надзвичайно важливим.
- Аутентифікація та авторизація: Контроль доступу до різних частин системи та визначення прав користувачів є критичним для запобігання несанкціонованому доступу.

5. Управління помилками та відновлення:

- Обробка помилок: Розподілені системи повинні бути відповідальними до помилок, які можуть виникнути під час взаємодії компонентів. Це включає в себе механізми виявлення помилок, їх обробку та відновлення.

6. Моніторинг та керування:

- Моніторинг роботи системи: Важливо мати ефективні засоби моніторингу для виявлення неполадок та оптимізації продуктивності системи.
- Керування розподіленою системою: Забезпечення можливості віддаленого керування та адміністрування системи.

Загальною метою при розробці розподілених додатків є створення ефективної, стійкої до помилок та високопродуктивної системи, яка здатна працювати в умовах розподіленості та можливих збоїв.

Ендрю Таненбаум описує різні варіанти вирішення проблеми, такі як клієнт-сервер, peer-to-peer та багатоагентні системи. Також розповідається про масштабованість та IoT (Internet of Things).

1.2 Архітектурні патерни для створення розподілених додатків

Архітектурні патерни є загальними рекомендаціями та established best practices у дизайні розподілених систем. Вони визначають основні принципи для

створення структури системи, яка може ефективно функціонувати в розподіленому середовищі.

1.2.1 Клієнт-Сервер

Клієнт-серверна архітектура — це модель, в якій програмне забезпечення розділене на дві основні частини: клієнтську та серверну. Клієнти та сервери взаємодіють між собою через мережу, дозволяючи виконання різних завдань та обробку даних в ефективний спосіб.

Клієнт-серверна архітектура складається з таких компонентів:

1. Клієнт: Це програмний компонент або пристрій, який ініціює запити та отримує відповіді від сервера. Клієнтська частина зазвичай відповідає за користувацький інтерфейс та взаємодію з користувачем.

2. Сервер: Сервер — це програмне забезпечення або пристрій, який слухає запити від клієнтів та надає їм необхідні ресурси або послуги. Серверна частина зазвичай містить бізнес-логіку та обробляє запити від клієнтів.

В Клієнт-Серверній Архітектурі клієнти та сервери взаємодіють через мережу. Клієнти генерують запити та відправляють їх на сервер для обробки, очікуючи відповіді. Сервер слухає запити, обробляє їх та повертає результати. Розділення обов'язків між клієнтською та серверною частинами дозволяє ефективно розділити логіку користувацького інтерфейсу та бізнес-логіку[19]. Цей паттерн сприяє масштабуванню, оскільки клієнтські та серверні частини можна масштабувати незалежно. Також цей підхід дозволяє створювати клієнти для різних платформ, таких як десктоп або мобільні пристрої, використовуючи спеціальні клієнтські додатки. Переваги включають простоту в розробці та обслуговуванні, але може виникати підвищений обсяг мережевого трафіку та залежність від доступності мережі.

1.2.2 Мікросервіси

Мікросервісна архітектура — це підхід до розробки програмного забезпечення, в якому додаток розбивається на невеликі автономні сервіси, що функціонують незалежно один від одного[1]. Кожен мікросервіс представляє собою

окремий, добре визначений функціональний блок, який може бути розроблений, розгорнутий, та масштабований незалежно від інших.

Взаємодія між мікросервісами відбувається через мережу, часто використовуючи API. Кожен мікросервіс може бути написаний різними мовами програмування та використовувати різні технології, що дозволяє використовувати найкращі інструменти для конкретного завдання.

Однією з основних переваг мікросервісної архітектури є її гнучкість та масштабованість. Розбиття додатка на невеликі сервіси спрощує розробку та утримання коду, дозволяє різним командам працювати над різними мікросервісами паралельно, та полегшує впровадження змін без впливу на інші частини системи.

Кожен мікросервіс може бути незалежно масштабований, що дозволяє оптимізувати використання ресурсів та забезпечувати ефективну роботу системи при змінних навантаженнях. Також, завдяки автономності мікросервісів, їх можна розгортати окремо, що спрощує процес впровадження та зменшує ризик виникнення проблем при оновленнях.

Проте, перехід до мікросервісної архітектури вносить свої виклики. Сервіси повинні бути добре документованими, інакше можуть виникнути проблеми з розумінням їх функціональності та взаємодії. Також, керування конфігурацією, моніторингом, та забезпеченням безпеки в умовах розподіленості може бути складнішим завданням порівняно з традиційними монолітними додатками.

У підсумку, мікросервісна архітектура є потужним інструментом для розробки сучасних та масштабованих систем, але вимагає дбайливого планування та управління для досягнення найкращих результатів.

1.2.3 Шина подій (Event Bus)

Шина подій (Event Bus) представляє собою архітектурний паттерн, який використовує механізм обміну подіями між різними компонентами системи. Цей механізм дозволяє різним частинам системи взаємодіяти між собою, навіть якщо вони знаходяться в різних місцях або працюють незалежно[23].

У системі, побудованій на основі шини подій, існує централізований об'єкт, який відомий як шина подій. Цей об'єкт служить посередником для розсилки та отримання подій між різними частинами системи. Кожна частина, яка хоче взаємодіяти з іншими частинами, може відправляти чи слухати події через цю шину.

Взаємодія здійснюється за допомогою паттернів "продюсер-консюмер". Компоненти, які викликають події, виступають як продюсери, що генерують події та надсилають їх на шину. З іншого боку, компоненти, які реагують на події, виступають як консюмери, які підписуються на отримання певних подій та реагують на них.

Шина подій використовується для вирішення проблеми залежностей між компонентами системи, забезпечуючи їхню взаємодію через локальний або глобальний рівень подій. Це полегшує розвиток та супровід системи, оскільки компоненти можуть бути додані або видалені без значного впливу на інші частини.

Однак важливо правильно розглядати використання шини подій, оскільки може виникати складність в розумінні потоків подій та їхніх наслідків. Деякі події можуть мати широкий обсяг впливу на систему, тому слід уникати надмірного використання цього паттерну та добре розуміти взаємодію компонентів.

У підсумку, шина подій є потужним інструментом для побудови розподілених та легко масштабованих систем, проте вона повинна використовуватися з обачливістю та урахуванням конкретних вимог системи.

1.2.4 Шари (Layered Architecture)

Архітектурний паттерн "Шари" (Layered Architecture) є стратегією організації програмного забезпечення, в якій система поділяється на логічні шари, кожен з яких виконує конкретний функціональний набір. Кожен шар представляє собою групу пов'язаних функцій та сервісів, і взаємодіє тільки зі шарами, розташованими нижче або вище за ним[24].

Основні компоненти шарової архітектури:

- Представлення (Presentation Layer): Цей шар відповідає за взаємодію з користувачем та представлення інформації. Він містить компоненти, такі як інтерфейс користувача, контролери, та все, що відображається на екрані користувача.
- Бізнес-логіка (Business Logic Layer): Цей шар визначає бізнес-правила та операції, які виконуються в системі. Він взаємодіє з даними та обробляє їх, щоб виконати конкретні функції додатку.
- Доступ до даних (Data Access Layer): Шар доступу до даних забезпечує зв'язок із сховищем даних, таким як база даних чи інші зовнішні джерела. Він відповідає за зберігання та отримання даних, необхідних для виконання бізнес-логіки.
- Інфраструктурний шар (Infrastructure Layer): Цей шар містить різноманітні служби та компоненти, необхідні для підтримки роботи всіх інших шарів. Сюди входять різноманітні сервіси, інструменти безпеки, журналювання, інструменти тестування, тощо.

Використання шарової архітектури дозволяє забезпечити високу модульність та розділення обов'язків між різними частинами системи. Кожен шар може бути реалізований незалежно, що полегшує тестування та зміни в коді без впливу на інші шари.

Проте, необхідно дбати про те, щоб рівні взаємодіяли між собою ефективно та ефективно. Занадто велика кількість шарів може призвести до зайвого підвищення складності та непотрібного розділення логічно пов'язаних компонентів.

Узагальнюючи, шарова архітектура відображає підхід, що дозволяє логічно та функціонально розділити складні системи на менші та керовані компоненти для полегшення розробки та обслуговування.

1.2.5 Шлюз API (API Gateway)

Шлюз API (API Gateway) представляє собою ключовий архітектурний паттерн у розробці розподілених додатків, де взаємодія між різними компонентами системи відбувається через централізовану точку входу. Цей паттерн дозволяє

ефективно керувати та координувати вхідний та вихідний трафік, забезпечуючи численні важливі функції для створення розподілених та масштабованих додатків.

Шлюз API виконує роль маршрутизатора, направляючи запити до відповідних мікросервісів, що дозволяє ефективно розподіляти та контролювати навантаження. Ця централізована точка входу відповідає за обробку автентифікації та авторизації, забезпечуючи безпеку системи та контролюючи доступ до ресурсів.

Однією з ключових можливостей є трансформація даних, де шлюз API забезпечує перетворення форматів запитів та відповідей, спрощуючи взаємодію між різними компонентами системи. Крім того, він може обмежувати швидкість запитів та використовувати кешування для покращення продуктивності та зменшення навантаження на мікросервіси[28].

Шлюз API відіграє також роль у веденні логів та моніторингу, що допомагає виявляти та вирішувати проблеми, а також вдосконалювати продуктивність системи. Його централізована функціональність робить його важливим елементом управління трафіком, надаючи зручну та ефективну точку входу та контроль за взаємодією всіх компонентів.

Узагальнюючи, Шлюз API як паттерн є важливою складовою для створення розподілених та масштабованих додатків. Його централізована точка входу та численні функціональності роблять його ефективним інструментом для управління взаємодією та забезпечення безпеки у складних системах.

1.3 Сфери застосування розподілених додатків

Розподілені додатки є ключовим елементом сучасного програмного середовища, де вимагається висока ефективність, масштабованість та надійність програмних систем. Ці додатки розподіляють обчислювальні завдання і дані через мережу, сприяючи оптимізації використання ресурсів та забезпечуючи доступ до функціональності в режимі реального часу.

1.3.1 Хмарні обчислення

Хмарні обчислення визначають новий етап у розвитку інформаційних технологій, де компанії та організації активно використовують цей підхід для оптимізації своїх бізнес-процесів та покращення доступу до обчислювальних ресурсів[29]. Ця сфера застосування розподілених додатків охоплює низку ключових аспектів:

1. Еластичність обчислювальних ресурсів: Хмарні обчислення дозволяють організаціям динамічно масштабувати свої обчислювальні потреби в залежності від обсягів роботи. Це забезпечує ефективне використання ресурсів та оптимізацію витрат.

2. Забезпечення доступу до обчислювальної потужності: Користувачі можуть без зайвих зусиль отримувати доступ до потужних обчислювальних ресурсів через мережу. Це особливо корисно для стартапів та малих підприємств, які не мають власної інфраструктури.

3. Забезпечення безпеки та надійності: Хмарні сервіси забезпечують високий рівень безпеки та автоматичної резервної копії, що робить їх привабливими для компаній, які прагнуть захистити свої дані та послуги від втрати чи атак.

4. Обробка великих обсягів даних: Розподілені системи в хмарних обчисленнях забезпечують потужність для обробки та аналізу великих обсягів даних, що є критично важливим для виявлення тенденцій, прогнозування та прийняття стратегічних рішень.

5. Підтримка розподіленої розробки: Хмарні сервіси надають засоби для спільної розробки та управління версіями, що дозволяє розробникам ефективно співпрацювати, навіть якщо вони фізично розташовані в різних частинах світу.

Загально кажучи, хмарні обчислення стають домінуючим фактором у сфері розподілених додатків, переосмислюючи підхід до обчислень та надаючи потужність інноваціям та розвитку в різних галузях.

1.3.2 Фінанси

В сучасному фінансовому світі, де обробка великих обсягів фінансових даних та забезпечення безпеки транзакцій є критично важливими завданнями, розподілені додатки виявляються необхідним інструментом для фінансових установ та організацій. Сфера фінансів використовує розподілені додатки для оптимізації бізнес-процесів, забезпечення високого рівня безпеки та надійності фінансових операцій.

1. Обробка фінансових транзакцій: Розподілені додатки використовуються для швидкої та ефективної обробки великих обсягів фінансових транзакцій. Вони дозволяють фінансовим установам в режимі реального часу виконувати операції, вести облік та моніторити ризики.

2. Управління ризиками та безпекою: Розподілені системи грають важливу роль у виявленні та управлінні ризиками у фінансовому секторі. Вони забезпечують аналіз та моніторинг фінансових операцій, виявлення підозрілих активностей та забезпечення безпеки фінансових даних.

3. Електронна комерція та онлайн-платежі: Розподілені додатки в електронній комерції дозволяють забезпечити безпеку та швидкість онлайн-платежів. Вони інтегрують різні платіжні системи та забезпечують зручний доступ до фінансових послуг[30].

4. Управління активами та портфелем: У фінансовому секторі розподілені додатки використовуються для управління активами та портфелями інвестицій. Вони дозволяють аналізувати ринкові тенденції, оптимізувати портфельні стратегії та приймати рішення на основі аналізу даних.

5. Автоматизована обробка документів: Розподілені системи в фінансах допомагають автоматизувати обробку документів, включаючи фінансові звіти, контракти та інші документи. Це полегшує рутинні операції та зменшує ймовірність помилок.

Розподілені додатки в галузі фінансів грають визначальну роль у забезпеченні ефективності, безпеки та надійності фінансових процесів. Вони

дозволяють фінансовим установам високоефективно виконувати свої завдання в умовах постійних змін та вимог ринку.

1.3.3 Мережеві технології

У світі постійних технологічних інновацій та зростаючої важливості забезпечення зв'язності та швидкості обміну інформацією, мережеві технології виявляються необхідним інструментом для підтримки різноманітних сфер. Розподілені додатки у сфері мережевих технологій сприяють побудові надійних та ефективних мереж, що об'єднують різноманітні пристрої та сервіси:

1. Інтернет речей (IoT): Розподілені додатки використовуються для об'єднання та управління великою кількістю різноманітних пристроїв, забезпечуючи високу ефективність та взаємодію в реальному часі.

2. Мережі забезпечені якістю обслуговування (QoS): Розподілені додатки допомагають впроваджувати та управляти системами, які гарантують певний рівень якості обслуговування для різних видів трафіку, що є ключовим для ефективного функціонування мереж[31].

3. Керування мережами: У сфері мережевих технологій розподілені додатки використовуються для автоматизації процесів конфігурації, моніторингу та управління мережевим обладнанням, що дозволяє забезпечити оптимальну ефективність мережі.

4. Віртуалізація мереж: Розподілені системи дозволяють створювати та управляти віртуальними мережами, що спрощує розгортання та масштабування інфраструктури в мережах.

5. Мережева безпека: Розподілені додатки використовуються для створення та управління системами мережевої безпеки, виявлення загроз та реагування на їхній виникнення.

6. Спільна робота та комунікації: В мережевих технологіях розподілені додатки дозволяють спільну роботу користувачів, обмін даними та забезпечення комунікацій в режимі реального часу.

7. Смарт-консультації та підтримка: Розподілені додатки використовуються для реалізації інтелектуальних систем підтримки та консультування через мережі, що сприяє швидкому доступу до інформації та послуг.

Мережеві технології, підтримані розподіленими додатками, є необхідним складовим елементом для побудови ефективних, безпечних та високопродуктивних інформаційних систем. Вони забезпечують інтеграцію та взаємодію різноманітних пристроїв та платформ, розвиваючи потенціал сучасних мереж.

1.3.4 Електронна комерція

Електронна комерція, або e-commerce, є однією з найшвидше розвиваючихся галузей бізнесу, де використання розподілених додатків грає ключову роль у забезпеченні ефективності, безпеки та зручності для як споживачів, так і підприємств. Розподілені додатки в електронній комерції використовуються для оптимізації процесів, забезпечення безпеки та надання персоналізованого досвіду користувачам:

1. Ефективний управління складом: Розподілені додатки в електронній комерції допомагають підприємствам ефективно управляти запасами та складськими запасами. Вони автоматизують процеси замовлення, відстеження товарів та управління запасами[32].

2. Персоналізовані рекомендації та маркетинг: Розподілені системи використовують дані користувачів для створення персоналізованих рекомендацій та маркетингових пропозицій, що сприяє збільшенню конверсії та задоволенню покупців.

3. Безпека та обробка платежів: У сфері електронної комерції безпека є пріоритетом. Розподілені додатки використовуються для шифрування даних, обробки платежів та захисту особистої інформації покупців.

4. Оптимізація процесів доставки та логістики: Розподілені системи управління логістикою та доставкою дозволяють оптимізувати маршрути, відстежувати вантажі та надавати користувачам точну інформацію про стан їхніх замовлень.

5. Мультиканальний продаж: Електронна комерція все більше розвивається в різних каналах продажу. Розподілені додатки дозволяють підприємствам ефективно управляти мультиканальними стратегіями та координувати продажі через різноманітні платформи.

6. Аналітика та звітність: Розподілені системи аналітики в електронній комерції надають підприємствам можливість аналізувати даний в реальному часі, сприяючи прийняттю інформованих рішень та оптимізації стратегій продажу.

7. Інтеграція з соціальними мережами: Розподілені додатки дозволяють ефективно інтегрувати функціональність електронної комерції з соціальними мережами, сприяючи розширенню аудиторії та підвищенню взаємодії з клієнтами.

Електронна комерція, підтримана розподіленими додатками, розвивається швидкими темпами, впроваджуючи нові технології для поліпшення досвіду покупців та оптимізації операцій бізнесу. Розподілені технології вносять суттєвий внесок у побудову ефективних та конкурентоспроможних електронних торговельних платформ.

1.3.5 Наука та дослідження

Наука та дослідження є ключовими компонентами сучасного прогресу та розвитку суспільства. Застосування розподілених додатків у науці та дослідженнях грає значущу роль у полегшенні спільної роботи вчених, оптимізації обробки великих обсягів даних та забезпеченні глобального доступу до наукової інформації:

1. Глобальна спільна робота: Розподілені додатки дозволяють вченим з усього світу співпрацювати в режимі реального часу над спільними проектами. Це сприяє обміну ідеями та ресурсами, що веде до ефективного наукового співробітництва[33].

2. Обчислювальні задачі та суперкомп'ютери: У наукових дослідженнях, де потрібні великі обчислювальні потужності, розподілені додатки

використовуються для розв'язання складних математичних задач та моделювання процесів в реальному часі.

3. Обробка та аналіз великих даних: В сфері наукових досліджень розподілені системи використовуються для обробки та аналізу великих обсягів даних, що дозволяє вченим виявляти тенденції, робити висновки та розробляти нові наукові гіпотези.

4. Віртуальні лабораторії та експерименти: Розподілені додатки дозволяють створювати віртуальні лабораторії та експерименти, що полегшує доступ до наукових засобів та сприяє інноваціям у різних галузях.

5. Інтелектуальна обробка даних: Розподілені системи використовують технології штучного інтелекту та машинного навчання для автоматизації обробки даних, виявлення патернів та покращення якості досліджень.

6. Електронне публікування та відкритий доступ: Розподілені додатки сприяють електронному публікуванню наукових робіт та забезпечують відкритий доступ до наукової інформації, що сприяє швидкому обміну знаннями.

7. Мережева безпека та конфіденційність даних: В наукових дослідженнях, де важлива конфіденційність, розподілені додатки використовуються для забезпечення високого рівня мережевої безпеки та захисту конфіденційної інформації[34].

Загалом, розподілені додатки у науці та дослідженнях реалізують інновації, полегшуючи спільну роботу вчених, розвиваючи технології та забезпечуючи доступ до знань у глобальному масштабі.

1.4 Аналіз програмного забезпечення для створення розподілених додатків

1.4.1 Рішення компанії Magento

Magento — це визнана світова платформа електронної комерції, яка дозволяє підприємствам створювати потужні та масштабовані інтернет-магазини.

Заснована в 2008 році, компанія вже більше десяти років лідирує у своєму сегменті ринку.

1. Мова програмування PHP та Фреймворк Zend:

- PHP (Hypertext Preprocessor): Magento використовує PHP, що є мовою програмування з відкритим вихідним кодом, оптимізованою для веб-розробки. PHP використовується для створення динамічних веб-сайтів та взаємодії з базами даних.
- Zend Framework: Це потужний фреймворк PHP, який надає структуру та гнучкість для розробки великих веб-додатків. Використання Zend Framework допомагає у створенні масштабованих та надійних рішень.

2. Реляційна база даних MySQL:

- MySQL: Як одна з найпоширеніших відкритих реляційних систем управління базами даних (RDBMS), MySQL використовується Magento для зберігання та організації великих обсягів структурованих даних[10].

3. Хмарні технології:

- Хмарна інфраструктура: Magento надає можливості хмарного розгортання, що дозволяє компаніям використовувати віртуальні ресурси у хмарі для забезпечення високої доступності та ефективної масштабованості під час зростання бізнесу.

4. HTML, CSS, та JavaScript:

- Використовується для створення структури веб-сторінок. Відповідає за стилізацію та оформлення веб-сайту, надає зовнішній вигляд. Використовується для додавання інтерактивності та динаміки на сторінці, забезпечуючи користувачеві більше функціональних можливостей без перезавантаження сторінки.

5. Мікросервісна архітектура (Magento Commerce Cloud):

- Мікросервіси: Magento Commerce Cloud грунтується на мікросервісній архітектурі, яка використовує набір невеликих та незалежних мікросервісів, що полегшує розробку, тестування та масштабування окремих компонентів

системи. Це сприяє гнучкості та зручності управління різними аспектами платформи.

Ці технології взаємодіють, створюючи потужну та масштабовану платформу електронної комерції, яка задовольняє потреби різних бізнесів.

Переваги використання технологій:

1. PHP та Zend Framework надають ефективність та гнучкість у розробці великих веб-додатків.
2. HTML, CSS та JavaScript дозволяють створювати естетичні та функціональні веб-сайти.
3. Мікросервісна архітектура полегшує розробку та масштабування компонентів системи.

Недоліки використання технологій:

1. PHP може мати обмежену швидкість виконання порівняно з іншими мовами програмування.
2. Різні браузери можуть різноманітно інтерпретувати код HTML, CSS та JavaScript, що може впливати на єдність вигляду та функціональності веб-сайту.

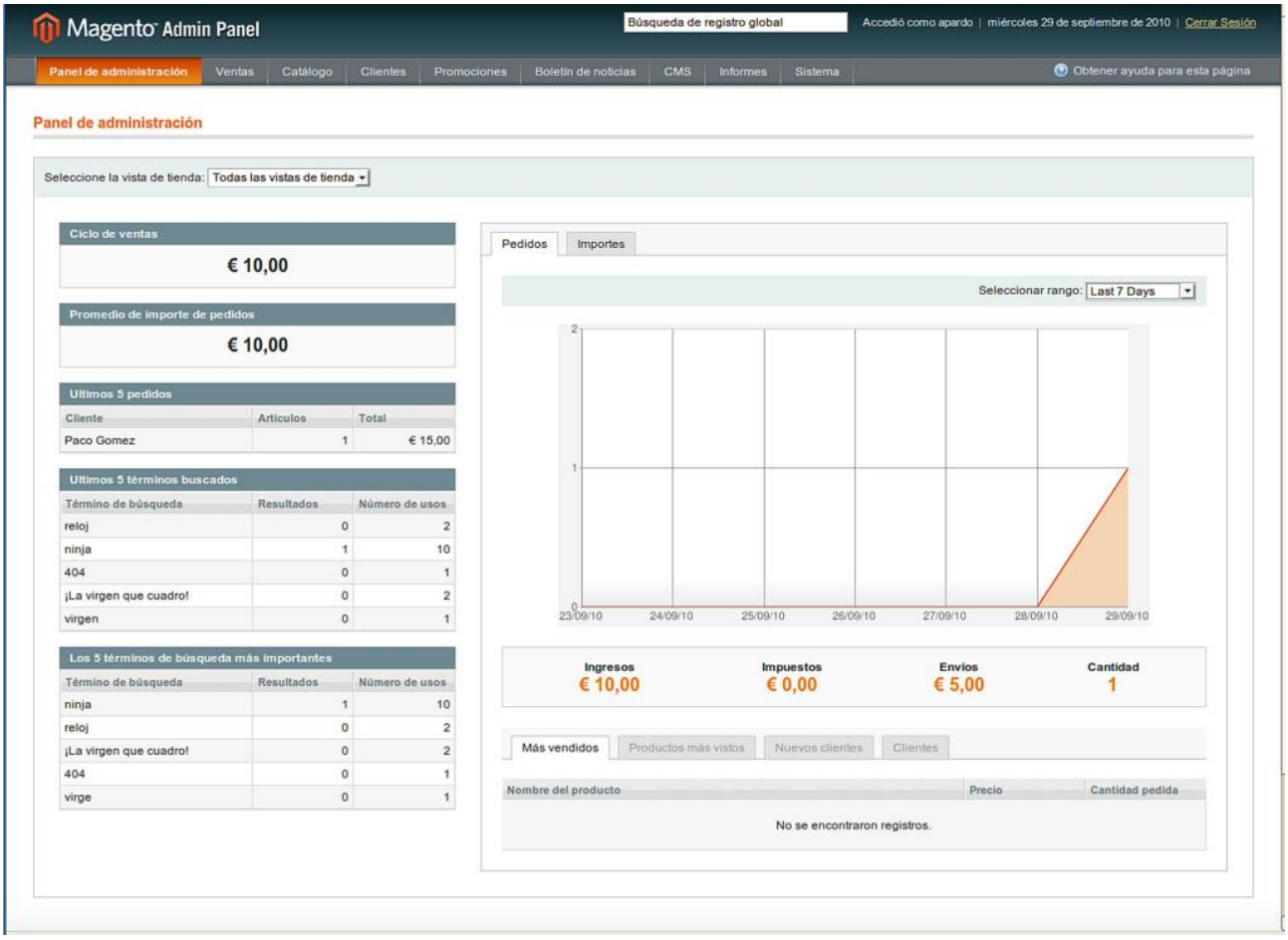

Рисунок 1 — Рішення продукту Magento

1.4.2 Рішення компанії eBay

eBay — це глобальна електронна комерційна платформа, яка виникла в 1995 році та перетворила спосіб, яким світ здійснює покупки та продажі. Заснована як онлайн-аукціон, eBay розширив свою діяльність та став місцем, де мільйони користувачів здійснюють торгівлю товарів і послуг, від унікальних антикваріатів до новітніх електронних пристроїв.

1. Java: eBay використовує Java як основну мову програмування для розробки своїх розподілених додатків. Java забезпечує високу продуктивність та масштабованість, що необхідно для обробки великого обсягу транзакцій та взаємодії з великою кількістю користувачів[9].

2. Node.js: Деякі частини системи eBay побудовані за допомогою Node.js. Це середовище виконання JavaScript, яке дозволяє ефективну обробку багатьох одночасних з'єднань. Node.js використовується, наприклад, для взаємодії з клієнтами на веб-сторінках, забезпечуючи асинхронні операції.

3. Розподілена архітектура: eBay використовує розподілену архітектуру, що дозволяє розділити функціональність на незалежні модулі. Це дозволяє легко масштабувати та оновлювати окремі компоненти системи, сприяючи ефективному функціонуванню платформи в умовах великої кількості користувачів та транзакцій.

eBay продовжує бути інноваційним лідером у світі електронної комерції, поєднуючи в собі високотехнологічні рішення та глобальний вплив для забезпечення зручного та надійного інтернет-торгівлі для мільйонів користувачів по всьому світу.

Переваги технологій:

1. Використання Java дозволяє забезпечити ефективну обробку великого обсягу транзакцій та високу масштабованість.

2. Node.js забезпечує ефективну обробку багатьох одночасних з'єднань, забезпечуючи швидке та асинхронне взаємодію на веб-сторінках.

3. Розподілена архітектура дозволяє легко масштабувати та оновлювати окремі компоненти системи, забезпечуючи ефективність та стійкість платформи.

Недоліки технологій:

1. Java може вимагати більше ресурсів порівняно з деякими іншими мовами програмування.

2. Хоча Node.js ефективно обробляє багато одночасних з'єднань, він може не бути оптимальним для всіх частин системи, особливо, якщо вимагається велика обчислювальна потужність.

3. Управління розподіленою архітектурою може вимагати додаткового часу та навичок.

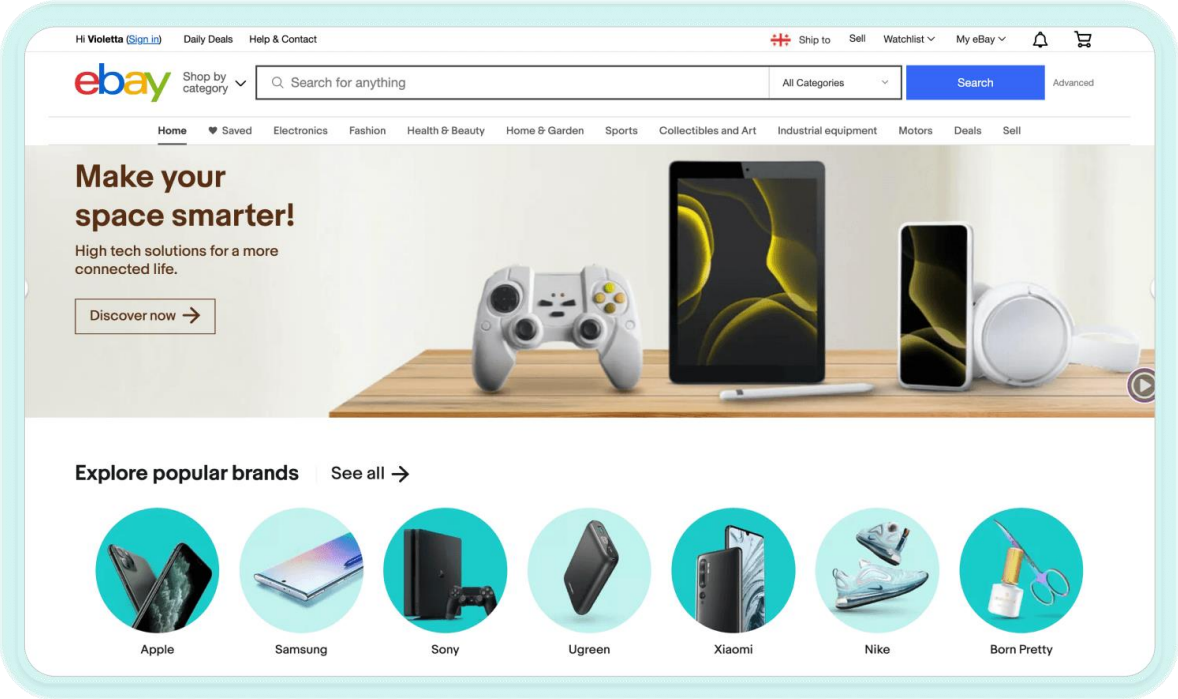

Рисунок. 2 — Рішення компанії eBay

1.4.3 Рішення компанії Slack

Slack є визнаним лідером у сегменті комунікаційних платформ, спеціалізованих на полегшенні співпраці в робочих колективах. Заснований у 2013 році, Slack швидко став невід'ємною частиною багатьох компаній та команд, надаючи ефективні інструменти для обміну інформацією та спілкування:

1. JavaScript та Electron:

- JavaScript: Як мова програмування для створення веб-інтерфейсу Slack на клієнтській стороні. Вона надає можливість динамічно змінювати та оновлювати елементи веб-сторінки без необхідності її перезавантаження.
- Electron: Це фреймворк для розробки крос-платформених десктоп-додатків з використанням веб-технологій[11]. Slack використовує Electron для створення десктоп-версії, що дозволяє користувачам взаємодіяти з Slack, як з будь-яким іншим десктоп-додатком, незалежно від операційної системи.

2. Real-time Messaging Protocol (RTM):

- Slack використовує RTM для забезпечення миттєвості обміну повідомленнями між користувачами. RTM дозволяє зберігати взаємодію в режимі

реального часу, що є ключовим аспектом для ефективної комунікації в офісному середовищі.

3. Microservices Architecture:

- Мікросервісна архітектура розбиває систему на невеликі і автономні модулі, які можуть функціонувати незалежно один від одного. Це полегшує розгортання, розвиток та масштабування окремих компонентів. Кожен сервіс відповідає за конкретну функцію або послугу.

Slack продовжує революціонізувати співпрацю та комунікації в робочих колективах, надаючи потужні інструменти для організації та спілкування, які враховують потреби сучасних робочих оточень, особливо в умовах роботи на відстані.

Переваги технологій Slack:

1. JavaScript дозволяє динамічно оновлювати веб-сторінку, забезпечуючи користувачам зручність.

2. Electron дозволяє створювати крос-платформені десктоп-додатки, що забезпечує єдність функціоналу незалежно від операційної системи.

3. Real-time Messaging Protocol (RTM) забезпечує миттєвість обміну повідомленнями, важливу для ефективної комунікації в реальному часі.

4. Microservices Architecture полегшує розгортання та масштабування окремих компонентів системи.

Недоліки технологій Slack:

1. JavaScript може бути обмеженою для складних обчислювальних операцій.

2. Electron може споживати більше ресурсів системи порівняно з нативними додатками.

3. RTM має високі вимоги до мережевого з'єднання, що може впливати на якість обміну повідомленнями.

4. Управління багатьма мікросервісами може вимагати додаткових зусиль та ресурсів.

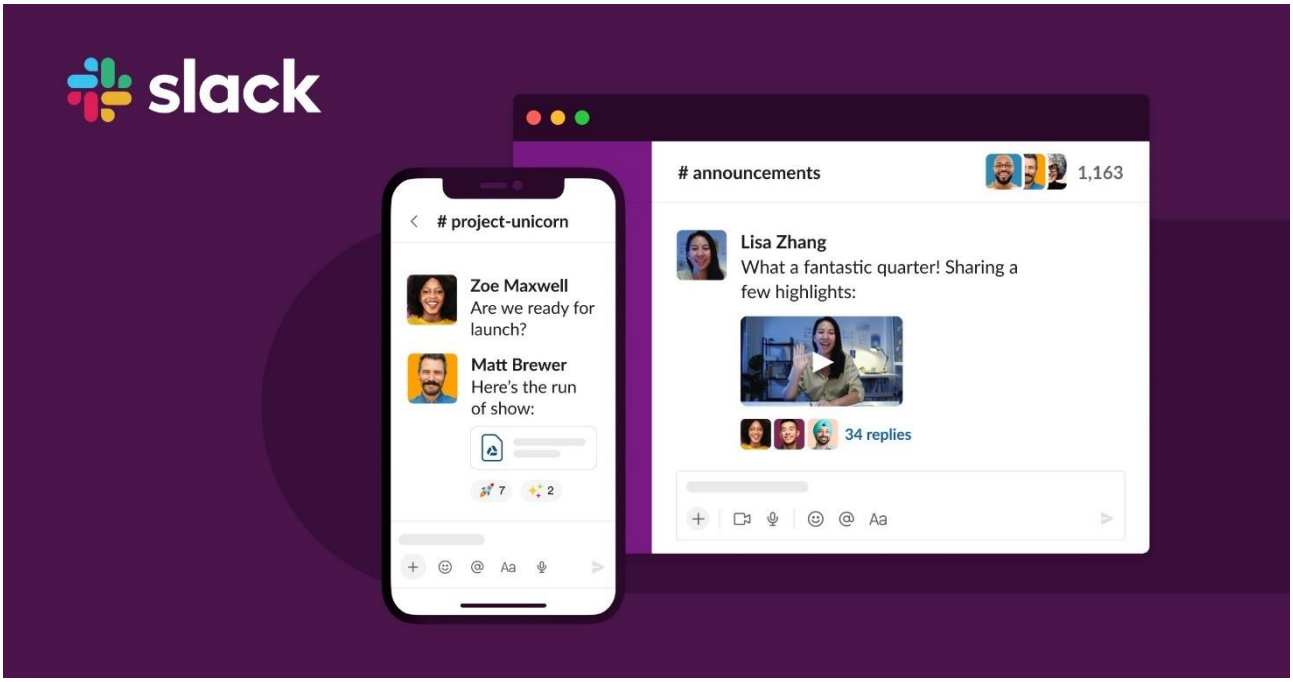

Рисунок 3 — Рішення продукту Slack

1.4.3 Рішення компанії Uber

Uber — це інноваційна технологічна компанія, яка змінила ландшафт пасажирських перевезень. Заснована в 2009 році, Uber надає платформу для замовлення транспорту та керування подорожами через мобільний додаток.

1. Мова Програмування:

- Java та Kotlin: Мобільні додатки Uber для платформ Android розроблені головним чином на Java та Kotlin. Ці мови дозволяють створювати швидкі та ефективні додатки для Android-платформи.
- Swift та Objective-C: Додатки для iOS розроблені з використанням Swift та Objective-C, мов програмування, які є стандартом для розробки програм на платформі iOS.

2. Backend та Серверні Технології:

- Node.js: Uber використовує Node.js для розробки свого серверного застосунку. Ця платформа дозволяє створювати ефективні та масштабовані серверні додатки, що важливо для обробки великого обсягу запитів[20].
- Python: Python використовується для розробки деяких серверних компонентів Uber. Він використовується для аналізу даних, машинного навчання та інших завдань.

3. Бази Даних та Сховища Даних:

- MySQL та PostgreSQL: Для зберігання та управління даними Uber використовує реляційні бази даних, такі як MySQL та PostgreSQL.
- Cassandra: Нереляційна база даних Cassandra використовується для обробки великого обсягу даних та забезпечення масштабованості системи.

Uber активно використовує розподілені системи та новітні технології для забезпечення ефективності, надійності та комфорту своїм користувачам у сфері пасажирських перевезень.

Переваги технологій в розробці Uber:

1. Java та Kotlin для Android, Swift та Objective-C для iOS, надають можливість створювати швидкі та ефективні додатки для обох платформ.
2. Використання Node.js для серверного застосунку забезпечує ефективність та масштабованість для обробки великого обсягу запитів.
3. Python використовується для розробки деяких серверних компонентів, зокрема для аналізу даних та машинного навчання.
4. Використання реляційних баз даних (MySQL та PostgreSQL) забезпечує надійність та управління даними.
5. Використання нереляційної бази даних Cassandra сприяє обробці великого обсягу даних та масштабованості системи.

Недоліки технологій в розробці Uber:

1. Залежність від двох різних мов програмування (Java/Kotlin для Android та Swift/Objective-C для iOS) може вимагати додаткових ресурсів для розробки та підтримки.
2. Використання різних технологій (Node.js та Python) може призводити до складнощів у взаємодії та обслуговуванні.
3. Використання різних типів баз даних (реляційних та нереляційних) може потребувати додаткового зусилля для їх управління та синхронізації.

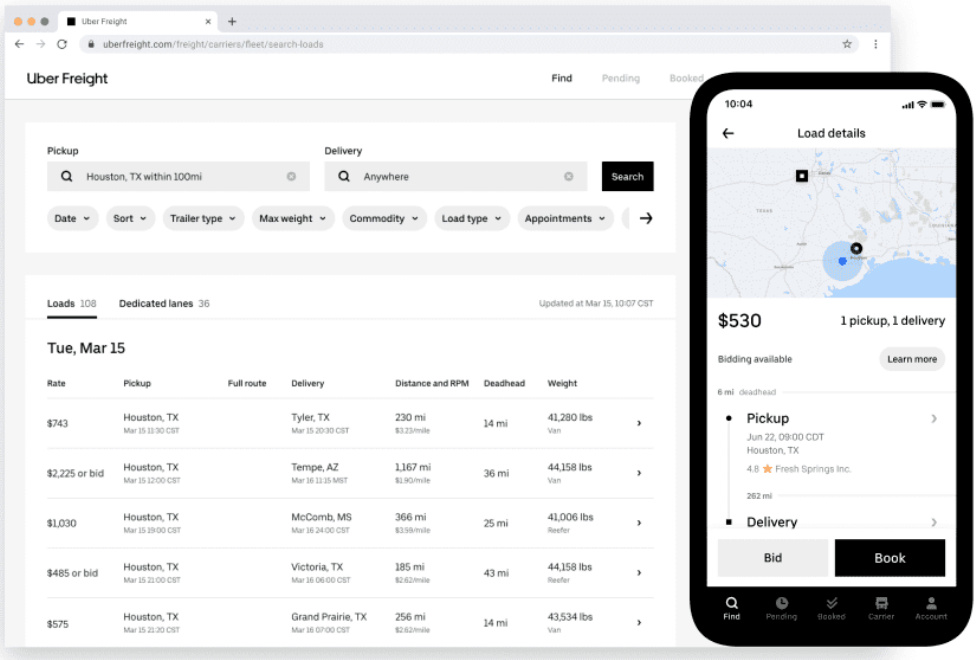

Рисунок 4 — Рішення компанії Uber

1.4.5 Рішення компанії Airbnb

Airbnb є платформою для знаходження та забронювання помешкань по всьому світу. Заснована в 2008 році, Airbnb перетворила спосіб подорожей, дозволяючи людям здається або орендовувати житло в більш неперетворених локаціях.

1. Мова Програмування:

- JavaScript та React: Airbnb використовує JavaScript для розробки веб-інтерфейсу свого сайту та React — для створення динамічних та ефективних користувацьких інтерфейсів[21].
- Swift та Kotlin: Додатки Airbnb для iOS та Android розроблені з використанням Swift та Kotlin, що дозволяє забезпечити ефективність та високу якість користувацького досвіду на мобільних пристроях.

2. Backend та Серверні Технології:

- Ruby on Rails: Airbnb використовує Ruby on Rails для свого серверного застосунку. Цей фреймворк прискорює розробку та надає розробникам зручні інструменти.

- Java та Scala: Деякі частини бекенду Airbnb написані на Java та Scala, що дозволяє компанії працювати з великою кількістю даних та забезпечує масштабованість.

3. Бази Даних та Сховища Даних:

- MySQL та PostgreSQL: Airbnb використовує реляційні бази даних, такі як MySQL та PostgreSQL, для зберігання даних користувачів, бронювань та інших елементів системи.

Airbnb вдало поєднує технологічну інноваційність з високоякісним користувацьким досвідом, створюючи ефективні розподілені додатки для своїх мільйонів користувачів.

Переваги технологій в розробці Airbnb:

1. Використання JavaScript та React для веб-інтерфейсу дозволяє створювати динамічні та ефективні користувацькі інтерфейси.

2. Використання Swift та Kotlin для мобільних додатків гарантує ефективність та високу якість користувацького досвіду.

3. Використання Ruby on Rails для серверного застосунку спрощує розробку та надає зручні інструменти для розробників.

4. Використання Java та Scala в бекенді дозволяє компанії ефективно працювати з великою кількістю даних та забезпечує масштабованість.

Недоліки технологій в розробці Airbnb:

1. Використання різних технологій (JavaScript/React для фронтенду та Swift/Kotlin для мобільних додатків) може вимагати додаткового зусилля для управління та підтримки.

2. Існування різних мов програмування в бекенді (Ruby on Rails, Java, Scala) може вплинути на єдність коду та ускладнити розробку.

3. Використання реляційних баз даних (MySQL та PostgreSQL) може обмежити швидкодію та масштабованість в разі значного збільшення обсягу даних.

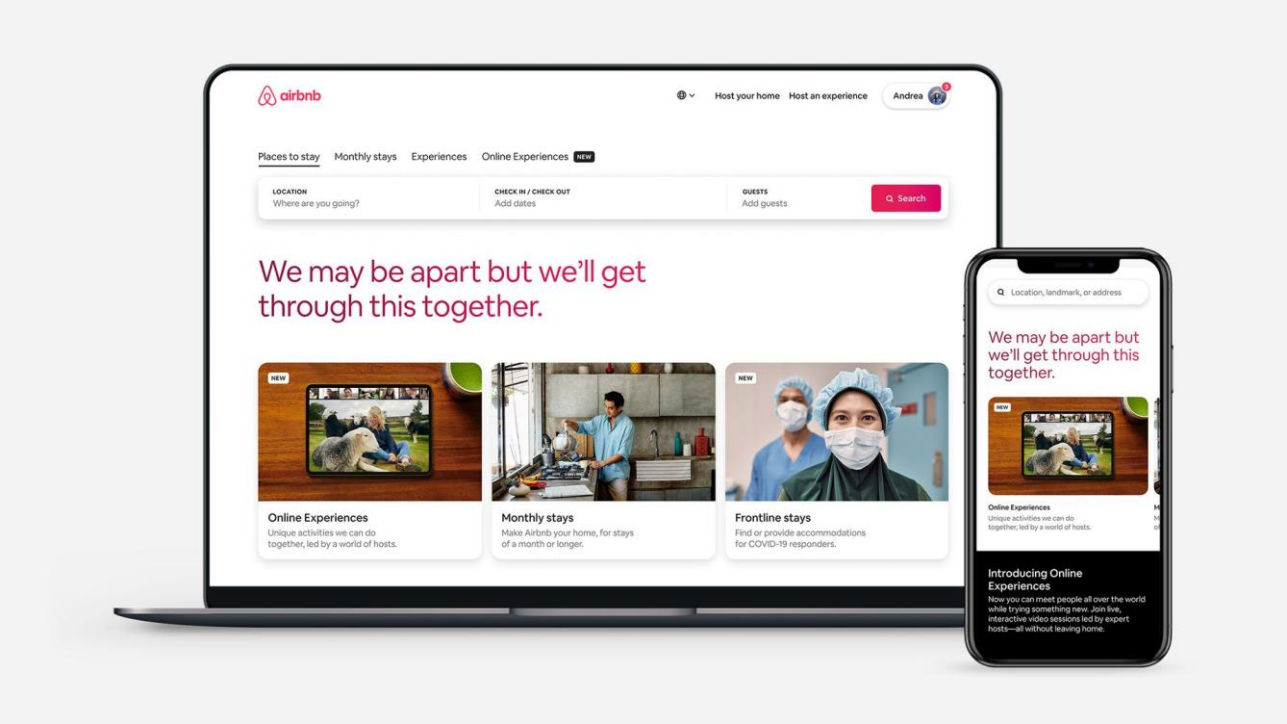

Рисунок 5 — Рішення компанії Airbnb

1.5 Висновки з розділу 1

1. У даному розділі була розглянута актуальна проблема створення розподілених додатків для єдиної системи, що визначає важливість подальших досліджень у цій області.

2. Дослідження дозволило виокремити основні виклики, пов'язані з розробкою розподілених додатків, та надати усвідомлення про їх вплив на сучасну IT-сферу.

3. Аналіз сучасного стану розподілених додатків підкреслив наявні тенденції та технології, що використовуються, що може послужити важливим вихідним пунктом для майбутнього розвитку.

4. Результати дослідження свідчать про важливість подальшого вдосконалення існуючих підходів до розробки розподілених додатків.

5. Проаналізовані підходи та технології створення розподілених додатків мають значущий потенціал для впровадження у практиці реальних проектів.

6. Висвітлені перспективи розвитку дозволяють спрямовувати увагу на ключові аспекти, які можуть покращити розробку та функціонування розподілених додатків.

РОЗДІЛ 2 ДОСЛІДЖЕННЯ ЗАСОБІВ ДЛЯ РОЗРОБКИ РОЗПОДІЛЕНИХ ДОДАТКІВ

2.1 Ключова роль React.js у розробці масштабованих та розподілених систем

React.js, або просто React, представляє собою потужний фреймворк для розробки інтерфейсів, який визначається своєю виразною та зручною для розробників синтаксичною структурою. Цей фреймворк виник з метою полегшення створення високоефективних та інтерактивних користувацьких інтерфейсів для веб-додатків.

Ключовою привабливістю React.js став його підхід до роботи із змінами в інтерфейсі, використовуючи віртуальний DOM. Цей механізм дозволяє оптимізувати процеси оновлення та відображення даних, забезпечуючи високу продуктивність веб-додатків[4]. Крім того, React впроваджує компонентну архітектуру, яка робить розробку більш модульною та легкою у відлагодженні[25].

Основні принципи React.js відображають його філософію простоти та гнучкості, що робить його ідеальним інструментом для тих, хто прагне розробляти ефективні та масштабовані веб-додатки. Відділення компонентів, зручне керування станом, та використання JSX (розширеного синтаксису JavaScript) роблять React зрозумілим та досить привабливим для широкого спектру розробників.

React.js виявляється надзвичайно корисним у забезпеченні масштабованості фронтенду в розподілених системах, де ефективна обробка великого обсягу даних та взаємодія з серверами визначають успіх проекту. Основною складовою його масштабованості є використання віртуального DOM.

Віртуальний DOM в React.js дозволяє зменшити кількість операцій оновлення інтерфейсу, забезпечуючи швидше та ефективніше відображення даних. Коли відбувається зміна, React генерує віртуальну копію DOM та порівнює її з реальним DOM, визначаючи лише ті частини, які потребують оновлення. Це

значно зменшує навантаження на браузер та дозволяє підтримувати ефективну роботу інтерфейсу в умовах великої кількості користувачів.

Компонентна модель React також грає ключову роль у створенні легко масштабованих і адаптивних інтерфейсів. Вона дозволяє розбити інтерфейс на невеликі та незалежні компоненти, які можна використовувати окремо. Це полегшує розвиток та масштабування окремих елементів системи, роблячи їх легко адаптованими для різних вимог та потреб. Використання компонентної моделі React дозволяє також використовувати мікросервісну архітектуру, де окремі компоненти можуть функціонувати незалежно один від одного. Це сприяє гнучкості та ефективності управління різними частинами інтерфейсу, роблячи їх легко масштабованими при рості обсягів даних та користувацького трафіку.

2.1.1 Мікросервісна Архітектура та Роль React.js

Мікросервісна архітектура — це підхід до розробки програмного забезпечення, в якому додаток складається з невеликих, незалежних служб (мікросервісів), які працюють разом. React.js, з своєю компонентною архітектурою, ідеально вписується у цей підхід, роблячи розробку та підтримку мікросервісів більш зручною та ефективною:

1. Незалежні Компоненти: React.js дозволяє розбити інтерфейс на невеликі, самодостатні компоненти. Ці компоненти можуть функціонувати незалежно один від одного, що ідеально підходить для мікросервісної архітектури. Кожен компонент може бути розроблений, розгорнутий та масштабований окремо, спрощуючи процес розробки та підтримки.

2. Легке Інтегрування: Завдяки компонентній природі React.js, нові функції або мікросервіси можна додавати, не впливаючи на інші частини системи. Це дозволяє легко інтегрувати новий функціонал без необхідності переробки всього додатку.

3. Гнучкість та Швидкість Розгортання: Розробка окремих мікросервісів на React.js дозволяє розвивати кожен сервіс незалежно, що робить систему

гнучкою та легко адаптованою до змін. Швидкість розгортання нових функцій або оновлень підвищується, оскільки це потребує менше часу та ресурсів.

4. Спільне Використання Компонентів: React.js дозволяє використовувати компоненти повторно. У мікросервісній архітектурі це може призвести до створення бібліотеки готових до використання компонентів, яку можна використовувати в різних частинах системи.

5. Легкість Тестування: Тестування окремих компонентів на React.js стає простішим завдяки їхній ізольованій природі. Це полегшує виявлення та виправлення помилок, а також додає впевненості у стабільності мікросервісів.

6. Масштабованість: Через можливість розробки та масштабування окремих компонентів, React.js сприяє масштабованості системи в цілому. Велика кількість мікросервісів може працювати паралельно, ефективно оброблюючи великий потік запитів.

React.js відкриває нові можливості для створення розподілених систем, працюючи як потужний інструмент для розробки масштабованих та гнучких мікросервісів[2].

2.1.2 Крос-платформність та розподілені додатки з використанням React Native

React Native, будуючи на основі основ React.js, є потужним інструментом для розробки крос-платформних мобільних додатків та відкриває нові можливості для створення розподілених систем[3].

1. Розробка для Обох Платформ: Однією з ключових переваг React Native є можливість розробляти мобільні додатки для обох основних платформ - Android та iOS. Загальний код може бути використаний для побудови додатків, що полегшує управління та розвиток для розподілених систем, які охоплюють обидві платформи.

2. Загальний Код та Швидкість Розробки: Використання React Native дозволяє командам розробників використовувати багато спільного коду для

розгортання функціоналу на різних платформах. Це призводить до економії часу і ресурсів, що є важливим аспектом у розподілених системах, де швидкість реагування на зміни важлива.

3. Ефективна Підтримка та Оновлення: Оскільки React Native дозволяє використовувати спільний код, підтримка та оновлення додатків на обох платформах стає більш ефективною. Виправлення помилок та впровадження нового функціоналу можуть бути внесені для всіх користувачів одночасно.

4. Масштабованість для Розподілених Систем: React Native дозволяє створювати масштабовані та ефективні мобільні додатки, що стає важливим у розподілених системах з великою кількістю користувачів та різними пристроями.

5. Нативна Ефективність та Взаємодія: Використання React Native дозволяє взаємодіяти з нативними компонентами платформ, забезпечуючи високу ефективність та якість використання додатків.

6. Адаптивність до Різних Умов: Крос-платформеність React Native дозволяє додаткам працювати на різних пристроях та різних роздільностях екрану, що є важливим в умовах розподілених систем з різноманітністю пристроїв[35].

Використання React Native у розподілених системах може виявитися стратегічно важливим для створення ефективних, швидких та масштабованих мобільних додатків.

2.1.3 Redux у розробці масштабованих та розподілених систем

Redux виявляється ключовим актором у світі розробки масштабованих та розподілених систем, надаючи ефективний механізм для управління станом додатків Основні аспекти цієї бібліотеки та її роль у специфіці цих проектів:

1. Централізоване Управління Станом:

- Redux зберігає стан додатка в єдиному об'єкті, дозволяючи ефективно керувати всією системою. Це особливо цінно в розподіленому середовищі, де стан може бути розділений між декількома серверами чи клієнтами.

2. Співпраця з React.js:

- Інтеграція з React.js робить Redux привабливим вибором для фронтенду розподілених додатків. З цим tandem'ом можна легко та ефективно керувати станом компонентів.

3. Дієве Імутабельне Оновлення:

- Redux сприяє дієвому використанню імутабельності, що є ключовим елементом у масштабованих системах. Це дозволяє уникнути неочікуваних станів і зробити систему більш дієвою.

4. Масштабованість в Розподілених Додатках:

- В розподіленому середовищі, де додатки можуть працювати на різних серверах, Redux допомагає уніфікувати та легко керувати станом в усіх частинах системи.

5. Легкість Відлагодження:

- Через централізований стан і чітко визначені дії, Redux спрощує відлагодження[15]. Розробники можуть ефективно відстежувати та аналізувати зміни в системі.

6. Плагінні Можливості:

- Redux надає можливості для використання плагінів, що полегшує налаштування під конкретні потреби розподіленої системи.

7. Синхронізація Стану у Реальному Часі:

- Використання middleware в Redux дозволяє ефективно синхронізувати стан у реальному часі, що важливо для розподілених систем, де інформація повинна бути актуальною.

8. Виклики та Рекомендації:

- Вивчення Redux та його відповідність конкретним потребам є важливим. Обізнаність із засадами бібліотеки дозволяє розробникам максимально використовувати її переваги[27].

- Обговорення вибору Redux повинно базуватися на вимогах конкретного проекту, розумінні його концепцій та переваг.

Використання Redux у розробці масштабованих та розподілених систем стає стратегічним рішенням, яке сприяє покращенню легкості управління станом та ефективності систем.

2.2 Роль та застосування Node.js у розробці розподілених додатків

2.2.1 Node.js та його переваги

Node.js — це потужний серверний фреймворк, побудований на основі мови програмування JavaScript. Його унікальність полягає в можливості використання JavaScript для створення серверних додатків, що відкриває нові можливості для розробників у побудові повноцінних веб-додатків та мікросервісів.

Node.js став важливим інструментом для розробників завдяки своїй спрощеній асинхронній моделі, яка дозволяє обробляти багато запитів одночасно. Це особливо важливо в умовах сучасного вебу, де миттєвість і ефективність грають ключову роль у задоволенні потреб користувачів. Node.js використовує подійно-орієнтовану архітектуру, що сприяє швидкій обробці подій і дозволяє побудову ефективних та масштабованих додатків[5].

Однією з ключових особливостей Node.js є його асинхронна природа. Він використовує неблокуючий ввід/вивід, що дозволяє виконувати інші завдання під час очікування на завершення операцій введення/виведення. Це забезпечує ефективне використання ресурсів та підвищує продуктивність додатків.

Крім того, Node.js спрощує створення масштабованих додатків за рахунок можливості обробки тисяч асинхронних з'єднань одночасно[6]. Це особливо корисно в контексті розподілених систем, де швидкість та ефективність грають критичну роль у взаємодії різних компонентів системи.

Node.js сприяє модульності та розширюваності, що є важливими аспектами при розробці розподілених додатків. Здатність використовувати різні модулі та пакети, а також розробляти власні, дозволяє розробникам ефективно створювати та підтримувати складні системи. Модульна архітектура Node.js полегшує

розширення функціоналу та взаємодію з іншими компонентами розподіленої системи.

2.2.2 Можливості застосування Node.js у розподілених додатках

Node.js визначається своєю високою ефективністю в розробці серверів з великим навантаженням[26]. Його однопотокова архітектура та асинхронна модель роботи дозволяють ефективно взаємодіяти з багатьма одночасними з'єднаннями, що робить його ідеальним вибором для сучасних розподілених систем. Node.js забезпечує ненав'язливий та швидкий спосіб обробки HTTP-запитів, що особливо важливо в умовах великого обсягу трафіку.

Node.js став ключовим інструментом для розробки розподілених додатків завдяки своїм властивостям та можливостям, що забезпечують ефективність та швидкість в таких системах. Асинхронна модель Node.js дозволяє ефективно обробляти велику кількість запитів одночасно[16]. У розподілених додатках, де важлива невідкладність та продуктивність, це забезпечує плавну взаємодію з різними компонентами системи. Однією з головних переваг Node.js є здатність ефективно масштабуватися. У розподілених системах, де кількість користувачів та обсяги даних постійно зростають, Node.js надає рішення для легкого розгортання та масштабування. Однопотокова архітектура Node.js та його близькість до JSON дозволяють ефективно використовувати ресурси системи. В розподілених додатках це стає важливим фактором оптимізації та забезпечення високої ефективності.

Node.js видається оптимальним вибором для великих та розподілених додатків, надаючи необхідний інструментарій для високопродуктивної, швидкої та масштабованої розробки.

2.3 Вебсокети у розподілених додатках

2.3.1 Роль вебсокетів у розподіленій архітектурі

Роль вебсокетів у розподіленій архітектурі полягає в забезпеченні зв'язку між різними компонентами системи в режимі реального часу. Основні аспекти цієї ролі включають:

1. Двосторонній зв'язок: Вебсокети надають можливість обмінюватися даними між клієнтом і сервером у двох напрямках. Це створює постійне та інтерактивне з'єднання, дозволяючи серверу надсилати сповіщення клієнтам без їхнього явного запитування, і навпаки[17].

2. Реальний час: Завдяки вебсокетам можливо забезпечити обмін даними у реальному часі. Це робить їх ідеальним інструментом для сценаріїв, де важлива миттєва взаємодія, таких як чати, співпраця над документами або онлайн-гри.

3. Зменшення затримок: Вебсокети дозволяють уникнути затримок, які можуть виникнути при повторних запитаннях до сервера. Замість цього, вони утримують постійне підключення, готове до взаємодії в будь-який момент.

4. Масштабованість: Завдяки своїй асинхронній природі та низькій навантаженості на мережу, вебсокети є ефективним інструментом для масштабування. Вони можуть обслуговувати багато одночасних з'єднань без втрати продуктивності.

5. Постійний зв'язок: Вебсокети забезпечують постійне з'єднання, що дозволяє уникнути необхідності постійно підключатися та відключатися, як це відбувається при кожному HTTP-запиті. Це зменшує накладні витрати та сприяє ефективності[18].

Таким чином, вебсокети відіграють ключову роль у розподіленій архітектурі, забезпечуючи ефективний та масштабований зв'язок між різними частинами системи.

2.3.2 Масштабованість та забезпечення стабільності вебсокетів

Масштабованість та забезпечення стабільності вебсокетів є ключовими аспектами для ефективного використання цієї технології в розподіленій архітектурі. Важливі аспекти цих питань включають:

1. Масштабованість:

- Архітектурний дизайн: При проектуванні системи з вебсокетами слід ураховувати можливість масштабування. Використання архітектурних паттернів, таких як шини подій або мікросервісна архітектура, дозволяє легко розширювати та додавати нові компоненти системи.
- Навантаження та ресурси: Ефективне керування навантаженням важливе для забезпечення масштабованості. Використання навантажувачів та балансерів навантаження може допомогти розподілити завдання між різними серверами[14].

2. Стабільність:

- Обробка помилок: Ретельна обробка помилок та винятків дозволяє уникати витоків та зберігати стабільність системи в цілому. Резервне зберігання даних та механізми відновлення можуть забезпечити надійність у випадках виникнення проблем.
- Моніторинг та журналювання: Системи моніторингу та журналювання дозволяють оперативно виявляти проблеми та надавати швидкі рішення для їх виправлення. За допомогою таких інструментів можна вчасно виявити навантаження та вдосконалити роботу системи.

3. Балансування:

- Балансування навантаження: Ефективне розподілення трафіку між різними серверами допомагає уникнути перевантаження та забезпечити рівномірне розподілений навантаження. Балансери навантаження можуть автоматично реагувати на зміни навантаження та перенаправляти трафік.

Масштабованість та стабільність вебсокетів грають важливу роль у розробці розподілених систем, де швидкість та надійність є ключовими. Правильне

проектування, моніторинг та обслуговування можуть значно полегшити управління цими аспектами технології в контексті великих та розподілених проектів.

2.3.3 Переваги використання вебсокетів

1. Вебсокети дозволяють миттєвий обмін даними між клієнтом і сервером у режимі реального часу. Це особливо корисно для розподілених систем, де потрібно негайно оновлювати інформацію на всіх з'єднаних пристроях.

2. Завдяки постійному відкритому з'єднанню вебсокетів, затримка передачі даних значно знижується порівняно з традиційними методами, такими як HTTP. Це дозволяє отримати швидку та ефективну взаємодію між клієнтом і сервером.

3. Вебсокети відмінно справляються з обробкою багатьох одночасних з'єднань, що робить їх ідеальним рішенням для систем з великою кількістю користувачів або пристроїв.

4. У випадку втрати з'єднання, вебсокети автоматично намагаються відновити з'єднання, що забезпечує стабільну і надійну роботу додатка при нестабільному Інтернет-з'єднанні[36].

5. Для реалізації функціоналу реального часу вебсокети спрощують вивчення та впровадження порівняно з іншими методами, такими як довгоживучі запити або опитування.

6. Вебсокети підтримуються більшістю сучасних браузерів та можуть бути використані в різних типах додатків, включаючи веб-сайти, мобільні додатки та десктопні застосунки.

2.4 Висновки до розділу 2

1. Розділ 2 надав детальний огляд ключових інструментів для розробки розподілених додатків, таких як React.js, Node.js та WebSocket.

2. Дослідження переваг та недоліків кожного інструменту дозволило отримати глибше розуміння їхнього внеску у створення масштабованих систем.

3. React.js відрізняється компонентною архітектурою та ефективністю віртуального DOM, що робить його потужним інструментом для фронтенд розробки.

4. Node.js використовується для серверної розробки, забезпечуючи асинхронну обробку та масштабованість у розподілених системах.

5. Використання WebSocket дозволяє миттєво обмінюватися даними, проте необхідно дбати про стабільність з'єднання для забезпечення надійності.

РОЗДІЛ 3 РОЗРОБКА ЗАСТОСУНКУ ДЛЯ РОЗПОДІЛЕНИХ СИСТЕМ

3.1 Розробка інтерфейсу застосунку

Інтерфейс застосунку було розроблено відповідно до задачі за допомогою якої буде доцільно передати користь від використання ReactJs & NodeJs у розподілених додатках.

Задача для вирішення полягає у створенні автоматизованої системи для проведення змагань. Система має замінити усі основні моменти у змаганнях, що раніше виконувалися людьми: створення заходу, реєстрація учасників, відлік з секундоміром на старті та фініші, підрахунок результатів вручну. Система складається з сервера та 3 видів клієнтів: мобільний застосунок, десктопний застосунок та веб-сайт.

Мобільний застосунок використовується на старті – за допомогою нього оператор вмикає ворота старта, будується черга і кожен атлет починає свій маршрут вчасно, по номеру і в своїй категорії. На старті є можливість поставити на паузу, відмінити участь атлета (якщо він не прийшов або його дискваліфіковано), змінити порядок атлетів або категорій. Цей функціонал виконується у мобільному застосунку.

Десктопний застосунок використовується на фініші – він напряму зв’язаний з воротами фініша. Вмикаючи десктопний застосунок ми одразу ж вмикаємо й систему воріт фініша. Тут ми можемо перевірити якість підключення системи, перевірити процес проходження фінішу на коректність, також звичайно створити івент і почати змагання. Процес створення івенту швидкий і легкий: ми вводимо назву, обираємо час, додаємо опису і створюємо стейджі – так звані маршрути на яких будуть змагання. Також ми можемо зупинити івент, закрити його і видалити.

Веб-сайт використовується , в першу чергу, як інформаційна складова. Учасники реєструються через веб-сайту, вболівальники можуть переглядати данні і звичайно всі по результату можуть побачити результати змагань.

Тож аналізуючи даний функціонал було розроблено інтерфейс системи.

3.1.1 Інтерфейс мобільного застосунку

На головному екрані зображено список змагань, які є відкритими і доступними до перегляду.(див. рис. 7) При натисканні на табличку відкривається список стейджів з вказаним часом та оператором, який за нього відповідає. Якщо оператора вже обрано, то інший оператор не може переобрати стейдж на себе. Це зроблено для того, щоб уникнути конфліктів.

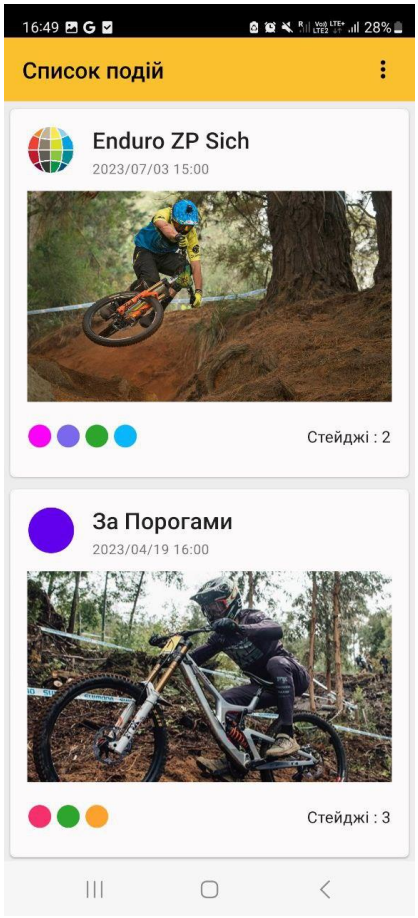

Рисунок 7 — Головна сторінка мобільного застосунку

При виборі стейджу відривається сторінку певного стейджу.(див. рис. 8) Там зображено список атлетів – черга, в якій вони будуть виступати. В кожного атлета є ім’я, номерок та час виступу. Також кожен атлет відноситься до певної категорії. Також ми бачимо велике коло зверху сторінки – це індикатор підключення до воріт старту. В залежності від кольору ми розуміємо чи ворота підключені, чи відкриті, чи в режимі очікування. Також біля кожного атлету є контекстне меню: воно використовується у випадку якщо ми хочемо відмінити атлета або змінити порядок черги.

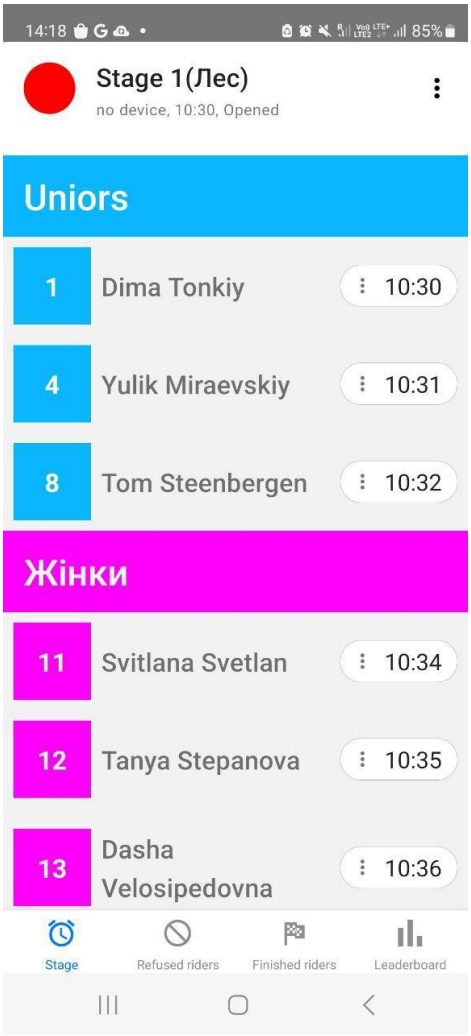

Рисунок 8 — Сторінка Стейджу та черги мобільного застосунку

3.1.2 Інтерфейс веб-сайту

На головній сторінці веб-сайту зображено список відкритих змагань. (Див. рис. 9) Тут будь-який користувач може подивитися інформацію про змагання та зареєструватися на нього.

Рисунок 9 — Головна сторінка сайту

Після закінчення змагань усі бажаючі можуть переглянути результати змагань: дізнатися час та місце.(Див. рис. 10)

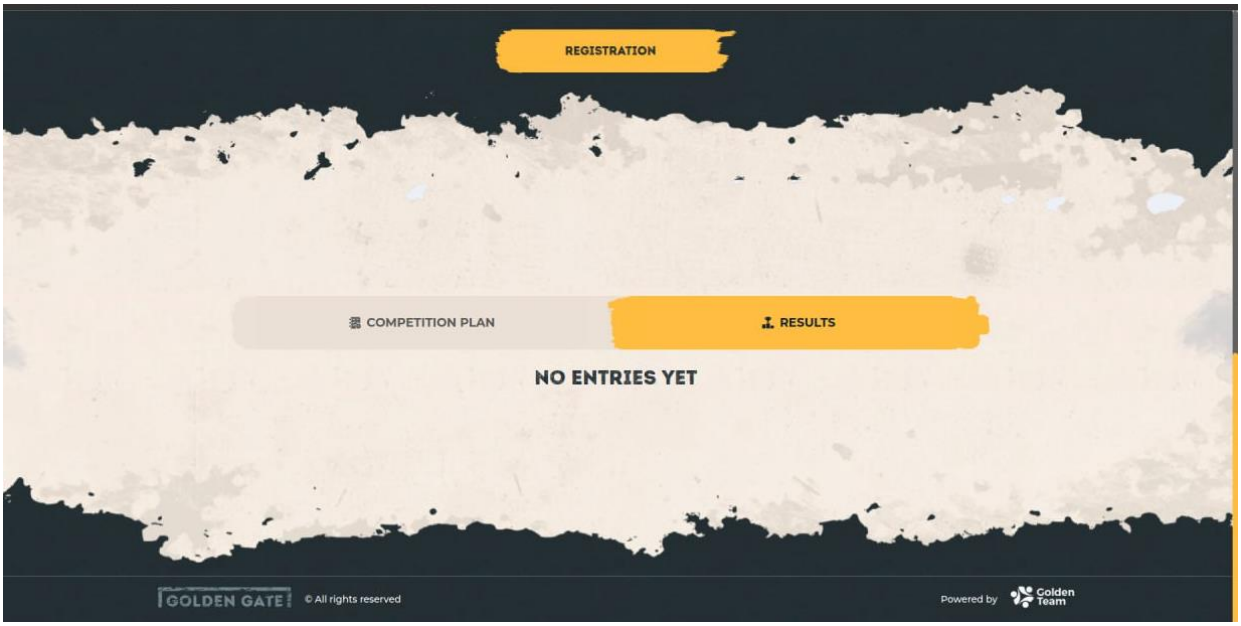

Рисунок 10 — Сторінка результатів

На сторінці атлетів ми маємо список усіх атлетів.(див. рис. 11) Кожен рядок розвертається і можна побачити усю важливу інформацію до кожного стейджа по певному атлету: місце, час початку і закінчення, різниця між кращим атлетом і вибраним. Також на цій сторінці є можливість редагування і видалення атлета.

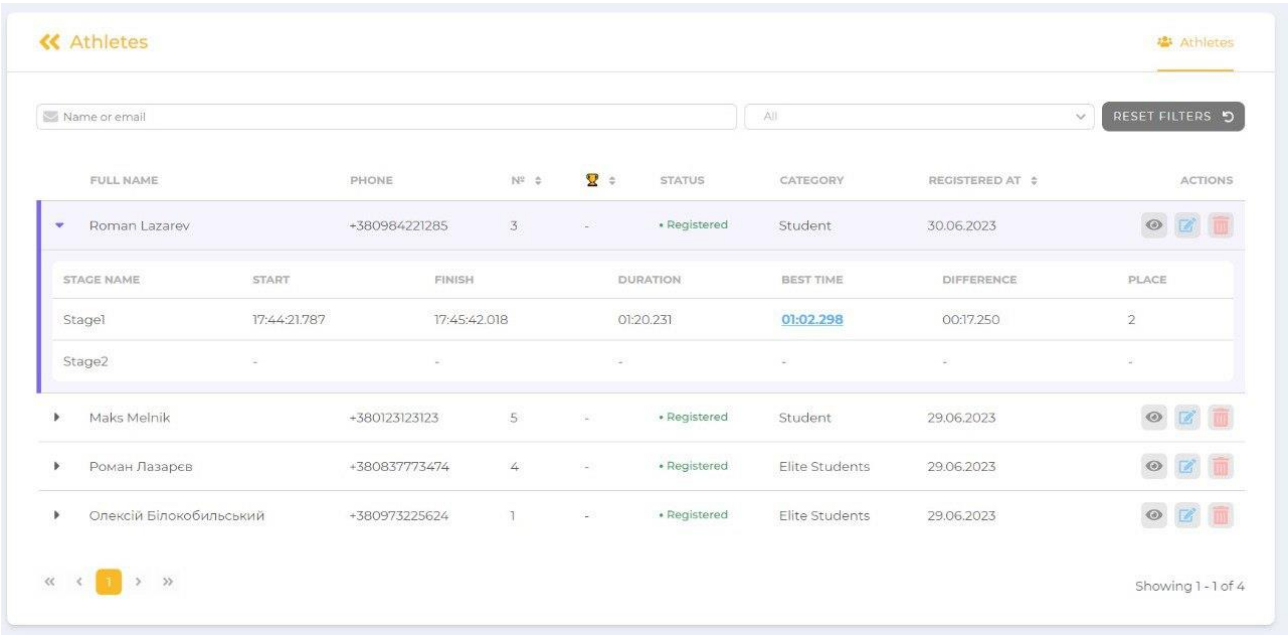

Рисунок 11 — Таблиця атлетів

3.1.3 Інтерфейс десктопного застосунку

Так як десктопний застосунок напряму пов’язаний з воротами фінішу, то інтерфейс застосунку будується на основні цього зв’язку. Тож існує функціонал для додавання номерку атлету. (Див. рис. 12) Потрібно ввести номерок і зчитати або ввести вручну RFID тег. Таким чином відбувається процес прошиття мітки.

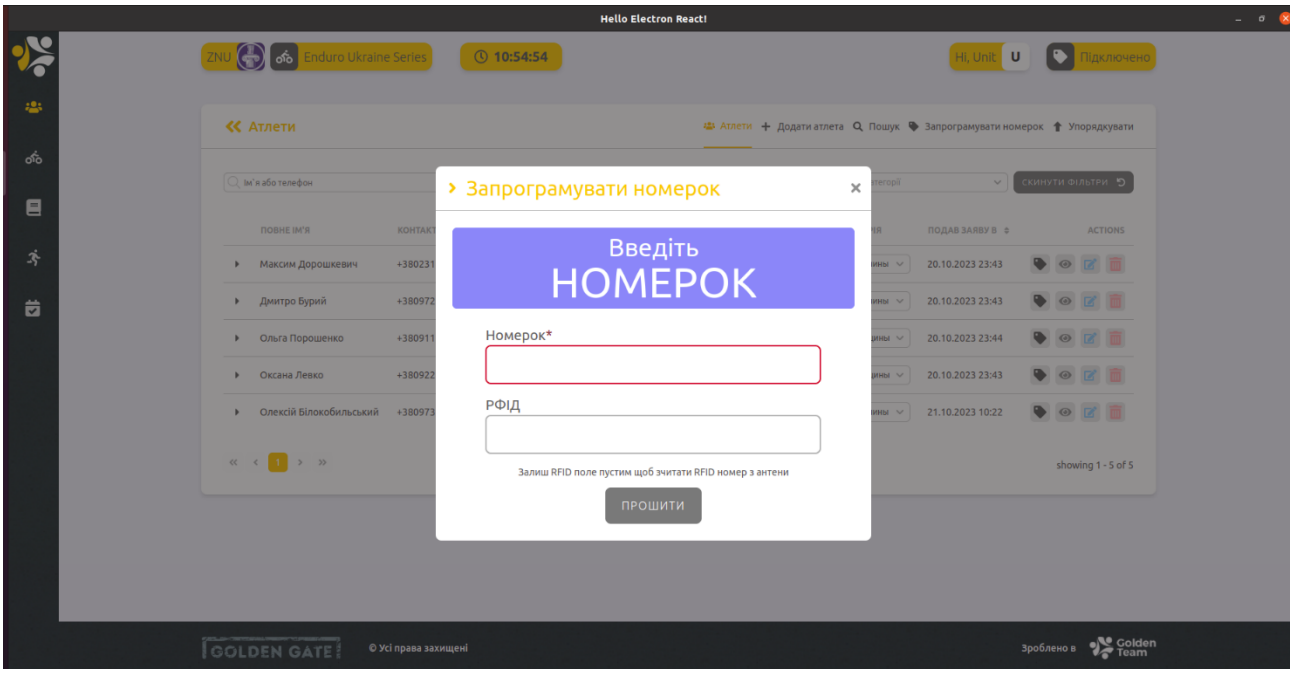

Рисунок 12 — Модальне вікно для присвоєння номерка атлету

Також існує функціонал перевірки готовності системи. (див. рис. 13). На модальному вікні зображено 3 таби: Обладнання, Антенна і Команди. Команди використовуються для перевірки певних команд, які відправляються на ворота фінішу. Обравши таб «Антенна» можна перевірити коректність зчитування номерка з мітки цією антенною. Перевірка обладнання заключається в тому, що існує 5 індикаторів, які по замовчуванню відображені червоним кольором. Натиснувши на кнопку «Тестувати систему» застосунок відправляє запит спочатку на веб сервер і отримуючи відповідь аналізує чи є зв’язок і чи він коректний. Після цього відправляється запит на RFID зчитувач (Alien або Impinj відповідно

від змагань), також аналізується відповідь і змінюється індикатор. Після цього перевіряється зв’язок з антенами зчитувача. Далі перевіряється зв’язок з воротами старта, їхня кількість залежить від кількості стейджів у даному змаганні.

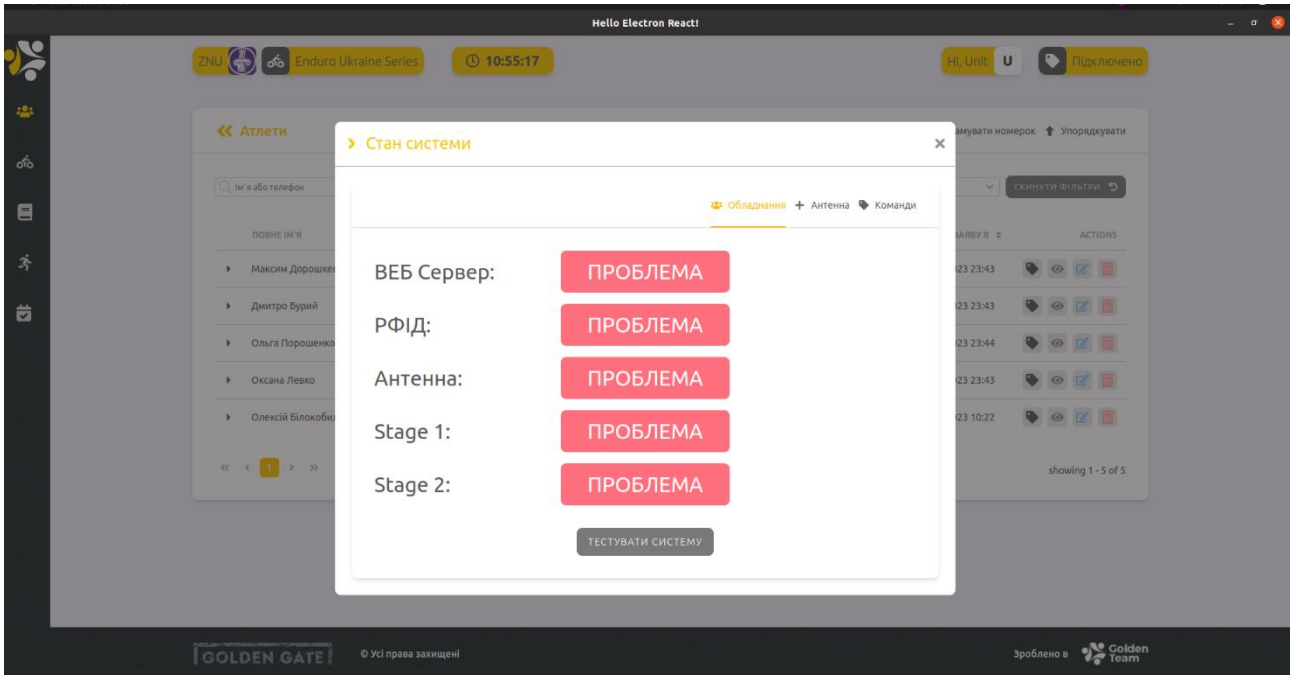

Рисунок 13 — Модальне вікно перевірки системи

3.2 Розробка програмного забезпечення

3.2.1 Архітектура системи

Архітектура даної системи має розподілений характер і складається з кількох ключових компонентів: веб-серверу, веб-сайту, мобільного додатку, що спілкується з Arduino та десктопного додатку, що спілкується з Impinj.(див. рис. 14) Сервер обмінюється даними з клієнтами за допомогою HTTP запитів, а для моментального оновлення оперативної інформації використовуються вебсокети[7]. Для зв’язку між мобільним додатком і Arduino (воротами старту) використовується стандарт Bluetooth, що забезпечує стабільне з’єднання та швидкісний обмін даними. Для зчитування даних з міток за допомогою зчитувача Impinj

використовується протокол інтернету речей MQTT – він гарантує швидкісну та точну відправку даних на десктопний застосунок.

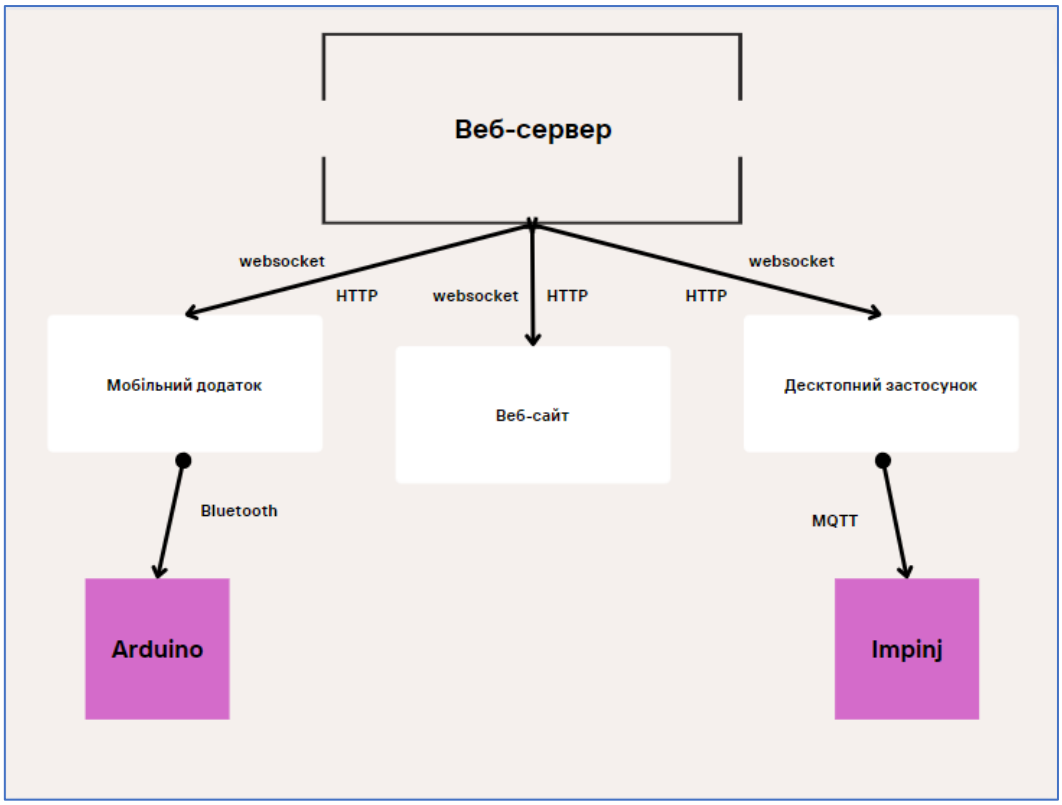

Рисунок 14 — Архітектура системи

3.2.2 Розробка серверної частини

Основним моментом у створенні системи було зробити її зручною для масштабування та швидкою для змін. Тому звичайно було використано усі основні принципи ООП [8]. У лістингу 1 описується створення базового класу сервера, від якого будуть наслідувати усі інші класи даного функціоналу. Клас NetServer, який використовує модуль net для створення сервера TCP. Клас слугує для взаємодії із клієнтським додатком, який використовує Electron. При створенні екземпляра класу вказується головне вікно BrowserWindow, і відбувається ініціалізація параметрів сервера, таких як IP-адреса і порт, які отримуються з конфігурації або локального IP-адреси пристрою.Сервер слухає події, такі як з'єднання (connection), помилка (error), закриття (close). При отриманні з'єднання викликається метод handleConnection, який встановлює кодування з'єднання, обробляє

отримані дані від читача RFID та висилає їх головному вікну. Додатково, клас має методи для обробки помилок (handleError), закриття (handleClose), запуску (handleStart), та зупинки (handleStop) сервера через IPC-повідомлення від головного процесу Electron. Клас також надає методи start, stop та isListening для ручного управління станом сервера.

Лістинг 1 — Реалізація базового класу Сервера

```
export default class NetServer {
    private mainWindow : BrowserWindow | null = null;
    private socket: net.Socket | null = null;
    private server: net.Server;
    private listenAddress: string;
    private listenPort: number;
    constructor(mainWindow: BrowserWindow) {
        this.mainWindow = mainWindow;
        this.listenAddress = getMyIPAddress() ||
config.listener.host;
        this.listenPort = config.listener.port;
        this.server = net.createServer();


        this.handleConnection =
this.handleConnection.bind(this);
        this.server.on('connection',
this.handleConnection);
        this.handleError = this.handleError.bind(this);
        this.server.on('error', this.handleError);
        this.handleClose = this.handleClose.bind(this);
        this.server.on('close', this.handleClose);
        this.handleStart = this.handleStart.bind(this);
        ipcMain.on('startRfidListener', this.handleStart)
        this.handleStop = this.handleStop.bind(this);
        ipcMain.on('stopRfidListener', this.handleStop)
```

```
    }
    private handleError(err: any) {
        console.log('Rfid status',
this.server?.listening, this.server, err);
    }
    private handleClose(err: any) {
        console.log('RFID Listener Server stopped.',
err);
    }
    private handleConnection(conn: net.Socket) {
        conn.setEncoding('utf8');
        conn.on('data', (rawReaderData: string) => {
            if (this.socket) {
                this.socket.destroy();
            }
            this.socket = conn;
            const readerDataArray = rawReaderData
                .split(/[\0\r\n]/)
                .filter((str) => str !== '');
            readerDataArray.forEach((row: string) => {
                const subArray = row.split(/,/);
                if (subArray.length >= 4) {
                    switch (subArray[0]) {
                        case 'TAG':
                        if
(subArray[1].match(/^([\w\d]{4}\s?){1,6}$/)) {

this.mainWindow?.webContents.send('rfidTag', subArray);
                        }
                        break;
                        default:
                        break;
                    }
```

```
                }
            });
        });
    }

    private handleStart(event: IpcMainEvent) {
        const port = this.listenPort;
        const address = this.listenAddress;
        this.stop().then(() => {
            this.server.listen(port, address, () => {
                console.info(`RFID Listener Server
started on: ${address}:${port}`);
                event.reply('startRfidListener:end',
true);
            });
        }).catch(e => {
            console.log('Can not start NetServer()', e);
        });
    }

    private handleStop(event: IpcMainEvent) {
        if (this.socket) {
            this.socket.destroy();
            this.socket = null;
        }
        this.stop().then(() => {
            console.log('NetServer stopped!');
            event.reply('stopRfidListener:end', true);
        }).catch(e => {
            console.log('Can not stop NetServer()', e);
        });
    }
```

```
    public stop() {
        return new Promise<void>((resolve, reject) => {
            if (this.server && this.server.listening) {
                this.server.close((error => {
                    console.log('netServer: Try to stop',
error)
                    if (!error) {
                        resolve();
                    } else {
                        reject(error)
                    }
                }));
                resolve();
            } else {
                resolve();
            }
        });
    }
    public start() {
        const port = this.listenPort;
        const address = this.listenAddress;
        this.server.listen(port, address, () => {
            console.info(`RFID Listener Server started
on: ${address}:${port}`);
        });
    }
    public isListening() {
        return this.server && this.server.listening
    }
}
```

У лістингу 2 демонструється створення класу HTTPServer, що реалізовує функціонал спілкування RFID зчитувача з застосунком. Цей код розширює функціональність класу `NetServer`. Він використовує модуль `net` для створення сервера TCP, призначеного для обміну даними з клієнтським додатком на Electron. Клас успадковує методи для обробки з'єднань, запуску та зупинки сервера. Також він містить додатковий метод `checkIfConnected`, який обіцяє повернути булеве значення та може використовуватися для перевірки стану з'єднання. Крім того, він налаштовує параметри сервера, такі як IP-адреса та порт, отримані з конфігурації або локального IP-адреси пристрою. Клас визначає методи для обробки подій запуску та зупинки через взаємодію з головним процесом Electron за допомогою IPC-повідомлень.

Лістинг 2 — Реалізація класу-спадкоємця для спілкування за допомогою HTTP протоколу

```
export default class HTTPServer extends NetServer{
    private socket: net.Socket | null = null;
    private server: net.Server;
    private listenAddress: string;
    private listenPort: number;
    constructor(mainWindow: BrowserWindow) {
        super(mainWindow);
        this.mainWindow = mainWindow;
        this.listenAddress = getMyIPAddress() ||
config.listener.host;
        this.listenPort = config.listener.port;
        this.server = net.createServer();
        this.server.on('connection',
this.handleConnection);
        this.server.on('error', this.handleError);
        this.server.on('close', this.handleClose);
```

```
    }
    protected handleConnection(conn: net.Socket) {
        conn.setEncoding('utf8');
        conn.on('data', (rawReaderData: string) => {
            if (this.socket) {
                this.socket.destroy();
            }
            this.socket = conn;
            const readerDataArray = rawReaderData
                .split(/[\0\r\n]/)
                .filter((str) => str !== '');

            readerDataArray.forEach((row: string) => {
                const subArray = row.split(/,/);
                console.log('DATA', subArray);
                if (subArray.length >= 4) {
                    switch (subArray[0]) {
                    case 'TAG':
                        if
(subArray[1].match(/^([\w\d]{4}\s?){1,6}$/))
{this.mainWindow?.webContents.send('rfidTag', subArray);
                        }
                        break;
                    default:
                        break;
                    }
                }
            });
        });
    }

    protected handleStart(event: IpcMainEvent) {
        const port = this.listenPort;
```

```
        const address = this.listenAddress;

        this.stop().then(() => {
            this.server.listen(port, address, () => {
                console.info(`RFID Listener Server
started on: ${address}:${port}`);
                event.reply('startRfidListener:end',
true);
            });
        }).catch(e => {
            console.log('Can not start NetServer()', e);
        });
    }
    protected handleStop(event: IpcMainEvent) {
        if (this.socket) {
            this.socket.destroy();
            this.socket = null;
            console.log('NetServer.socket stopped!');
        }
        this.stop().then(() => {
            console.log('NetServer stopped!');
            event.reply('stopRfidListener:end', true);
        }).catch(e => {
            console.log('Can not stop NetServer()', e);
        });
    }

    public stop() {
        return new Promise<void>((resolve, reject) => {
            if (this.server && this.server.listening) {
                this.server.close((error => {
                    console.log('netServer: Try to stop',
error ? error: 'success' );
```

```
                    if (!error) {
                        resolve();
                    } else {
                        reject(error);
                    }
                }));
                resolve();
            } else {
                resolve();
            }
        });
    }
    protected checkIfConnected(): Promise<boolean> {
        return Promise.resolve(true);
    }
}
```

У лістингу 3 демонструється створення класу MQTTServer, що реалізовує функціонал спілкування Impinj зчитувача з застосунком. Цей код визначає клас MQTTServer, який розширює функціональність класу NetServer. Клас використовує бібліотеку `mqtt` для встановлення з'єднання з MQTT-брокером та обробки повідомлень. Він ініціалізує параметри брокера, клієнтський ідентифікатор та тему для підписки з конфігурації. При підключенні до брокера викликається метод handleConnection, а при отриманні повідомлення - метод handleMessage, який декодує дані RFID та відправляє їх головному вікну. Клас також має методи для обробки запуску та зупинки сервера через IPC-повідомлення від головного процесу Electron. Метод stop відписується від теми та при необхідності відключається від брокера. Метод checkIfConnected перевіряє, чи зберігається з'єднання з брокером, відправляючи тестове повідомлення. Загалом, клас MQTTServer реалізує інтеграцію з MQTT-брокером для отримання та обробки даних RFID в контексті Electron-додатку.

Лістинг 3 — Реалізація спілкування Impinj з застосунком за допомогою MQTT протоколу

```
export default class MQTTServer extends NetServer {
    private client : mqtt.MqttClient;
    private brokerUrl = config.impinj.brokerUrl;
    private clientId = 'mqttjs_' +
Math.random().toString(16).substr(2, 8);
    private topic = config.impinj.topic;
    constructor(mainWindow: BrowserWindow) {
        super(mainWindow);
        this.mainWindow = mainWindow;
        this.client = mqtt.connect(this.brokerUrl, {
clientId:this.clientId });
        this.client.on('connect', this.handleConnection);
        this.client.on('error', this.handleError);
        this.client.on('close', this.handleClose);
        this.handleMessage =
this.handleMessage.bind(this);
      }
    protected handleConnection() {
        console.log('!Connected to MQTT broker');
    }
    protected handleMessage(topic:any, message:any) {
        try {
            const json = JSON.parse(message?.toString());
            if (json['tagInventoryEvent']['epc']){
                const decodedData =
Buffer.from(base64.toByteArray(json['tagInventoryEvent'][
'epc'])).toString('hex');
                this.sendRfidTag(['check', decodedData,
moment(json['timestamp']).valueOf()]);
            }
        } catch (error) {
```

```
            console.log('Can not decoded data from rfid',
message);
        }
    }
    protected handleStart(event: IpcMainEvent) {
        this.client.subscribe(this.topic,  (err:any) => {
            if (err) {
                console.error(`Failed to subscribe to
${this.topic}: ${err}`);
            } else {
                console.log(`MQTT started and subscribed
to ${this.topic}`);
                this.client.on('message',
this.handleMessage);
                event.reply('startRfidListener:end',
true);
            }
        });
    }

    protected handleStop(event: IpcMainEvent) {
        this.stop().then(() => {
            event.reply('stopRfidListener:end', true);
        }).catch(e => {
            console.log('Can not stop MQTTServer()', e);
        });
    }
    public stop() {
        return new Promise<void>((resolve, reject) => {
            if (this.client && this.client.connected) {
                this.client.unsubscribe(this.topic, {},
(error?: Error, packet?: mqtt.Packet) => {
                    if (!error) {
                        resolve();
```

```
                    } else {
                        reject(error);
                    }
                });

            } else {
                resolve();
            }
        });
    }
    protected checkIfConnected() {
        return new Promise<boolean>((resolve, reject) =>
{
            const topic: string = config.impinj.topic;
            const url: string = config.impinj.brokerUrl;
            if (this.client.options.hostname &&
url.includes(this.client.options.hostname)){
                this.client.publish(topic, 'test', (err)
=> {
                    if (!err) {
                        return
resolve(this.client.connected);
                    } else {
                        return reject(err);
                    }
                });
            } else {
                return reject('MQTT broker is not
available');
            }
        });
    }
}
```

3.2.3 Вебсокети

Всі клієнти спілкуються з сервером за допомогою вебсокетів. Лістинг 4 деконструє підключення вебсокетів на стороні сервера. Цей код реалізує клас SocketServer, що діє як WebSocket-сервер для забезпечення двостороннього зв'язку між клієнтами та сервером. У конструкторі виконується ініціалізація серверних властивостей, а метод start() запускає HTTP- та WebSocket-сервери, налаштовані на певний порт з CORS для прийому з'єднань з будь-яких джерел. Метод onConnection() обробляє нові з'єднання, зберігає їх та взаємодіє з ними. При закритті з'єднання викликається onCloseConnection(), який встановлює таймер для затримки видалення з'єднання. Крім того, є метод onAccept(), що реагує на отримання сигналу 'accept' від клієнта. Метод send() відправляє повідомлення всім підключеним клієнтам. Усі дії спрямовані на підтримку безперервної та ефективної комунікації між сервером та клієнтами за допомогою WebSocket-з'єднань.

Лістинг 4 — Реалізація підключення вебсокетів на стороні сервера

```
export default class SocketServer extends BaseContext{
    private httpServer: http.Server;
    private connects: {};
    private disconnection_delay: null;
    constructor(opts: IContextContainer) {
        super(opts);
        this.onConnection = this.onConnection.bind(this);
        this.onCloseConnection =
this.onCloseConnection.bind(this);
        this.onAccept = this.onAccept.bind(this);
        this.sendToKey = this.sendToKey.bind(this);
    }
    public start() {
        const { config } = this.di;
```

```
        this.httpServer = http.createServer();
        this.disconnection_delay =
config.socket.disconnection_delay;
        const io = new Server(this.httpServer,  { cors: {
            origin: '*',
            methods: ["GET", "POST"],
        }});
        io.on("connection",  this.onConnection);
   this.httpServer.listen(config.socket.port).on('listening',
() => {
            console.info(`Socket server listening on
localhost:${config.socket.port}`);
        }).on('error', err => {
            console.error(`Error on the socket server:`,
err);
        });
    }
    public stop() {
        if (this.httpServer) {
            this.httpServer.close();
        }
    }
    private onConnection(socket) {
        this.logBuffers();
        const key = socket.handshake.query.sessionKey;
        if (!(key in this.connects)) {
            this.connects[key] = { socket: null, buffer: {},
timeOut: null };
        }
        else {
            clearTimeout(this.connects[key]['timeOut']);
            this.connects[key]['timeOut'] = null;
        }
        this.connects[key]['socket'] = socket;
```

```
        socket.conn.on("close", (reason) => {
            if(reason.includes('transport close')) {
                this.onCloseConnection(key)
            }
        });
        socket.on('accept', this.onAccept);
        const buffer = this.connects[key]['buffer'];
        Object.keys(buffer).map(hash => {
            this.sendToKey(buffer[hash]['roomName'],
buffer[hash]['msg'], key);
        });
    }
    private onCloseConnection(sessionKey) {
        if (this.connects.hasOwnProperty(sessionKey)) {

if(!this.connects[sessionKey].hasOwnProperty('timeOut') ||
this.connects[sessionKey]['timeOut'] === null) {
                this.connects[sessionKey]['timeOut'] =
setTimeout(() => {

if(this.connects.hasOwnProperty(sessionKey)) {
                        delete this.connects[sessionKey];
                    }
                }, this.disconnection_delay)
            }
        }
    }
    public send(entityName: string, msg: any) {
        if (!isEmpty(this.connects)) {
            Object.keys(this.connects).map((key) =>
this.sendToKey(entityName, msg, key) );
        }
    }
}
```

3.3 Висновки до розділу 3

1. Було налаштоване середовище для розробки програмного забезпечення за допомогою ReactJs, NodeJs, NextJs, React Native та середи розробки Visual Studio Code.
2. Було розроблено інтерфейс та створено відповідні компоненти для реалізації однотипного UI на всіх клієнтах.
3. Виконано задачу створення коректної архітектури для розподілених додатків. Окремо створено архітектуру веб-сайту, десктопного застосунку і мобільного застосунку.
4. Інтегровано необхідні бібліотеки для спілкування застосунків з Arduino, Impinj, Alien.
5. Розроблено програму для кожного з застосунків.
6. Інтегровано функціонал спілкування між сервером та клієнтами за допомогою вебсокетів за допомогою бібліотеки Socket.io.
7. На етапі оптимізації було створено базовий клас netServer для масштабування системи та підключення до різних RFID зчитувачів. Окрім цього, було створено класи-спадкоємці, що наслідують від базового класу та реалізовують функціонал спілкування з певним зчитувачем за допомогою певного протоколу.
8. На етапі оптимізації також було оптимізована реалізація паузи для мобільного застосунку.

РОЗДІЛ 4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ РОБОТИ КОМП'ЮТЕРНОЇ СИСТЕМИ РОЗПОДІЛЕНИХ ДОДАТКІВ

4.1 Результати тестування роботи розподілених додатків єдиної системи

У процесі розробки системи було створено архітектуру, що являє собою розподілені додатки, які працюють по єдиній схемі. Мається на увазі, що при використанні ReactJs + Nodejs у розробці такого типу систем вирішується дуже багато проблем, що пришвидшують та покращують розробку. Аналізувати усі покращення будемо при проведенні реальних змагань з велосипедного спорту, що проходили на гірсько-лижному курорті «Плай» у серпні 2023 року.

Систему було протестовано на реальних умовах і не лабораторному середовищі для того, щоб точно визначити чи правильно працюють зчитування міток, чи моментально оновлюються усі дані за допомогою вебсокетів і ,звичайно, знайти можливі помилки або проблеми, що неможливо визначити у лабораторних умовах.

Тож було створено новий захід, зареєстровано учасників та присвоєно кожному номерок з міткою, яку буде зчитувати антенна (див. рис. 15). У цьому випадку відмінно відпрацювала система: десктопний застосунок присвоював номерки, і в ту ж секунду учасники могли переглянути свій номерок у телеграм каналі. В цьому випадку видно наскільки правильним рішенням було використовувати вебсокети для реалізації спілкування між сервером та клієнтами.

Рисунок 15 — Номерок учасника з RFID міткою

Далі відбувся початок змагань. При цьому оператор, що сидить на старті та керує їм та чергою за допомогою мобільного застосунку знаходиться на вершині гори. Ворота фінішу встановлено внизу гори (див. рис. 16). Веб-сайт відкритий на ноутбуці адміністратора, що знаходиться у палатці адміністрації. Змагання починаються, на старті перший атлет. Він проходить крізь ворота старту і це моментально видно на оперативній таблиці на воротах фінішу і на веб-сайті. Таким чином ще раз підтверджується коректність використання вебсокетів при виконанні цієї задачі, так як гарантується надійне та швидке доставлення даних до всіх клієнтів.

Рисунок 16 — Ворота фінішу

Після цього тестується ситуація коли атлет, що повинен стартувати наступним не прийшов, бо запізнюється чи щось сталося на старті і потрібно поставити старт на паузу, щоб атлет не відмінився. При цьому атлет, що стартував на змаганнях перший вже готується до старту на другому стейджі. Тож оператор ставить свій перши стейдж на паузу, повідомляє другого оператора і той ставить свій стейдж на паузу. При цьому постійно оновлюється івент для коректного відображення таймеру і пересування черги на відповідну кількість часу. При перших ітераціях такі дії призводили до руйнування системи, бо відбувався конфлікт. Тоді на ітерації оптимізації було змінено процес паузи і на змагання тест пройшов успішно. Стейджі було відновлено, час пересунувся коректно і змагання було продовжено.

Далі переходимо до процесу фінішу. Було протестовано швидкість реакції системи на перетинання фінішу на великій швидкості (див. рис. 17). При перетинанні атлетом воріт фінішу RFID зчитувач забирає інформацію з мітки та відправляє на сервер, що в свою чергу оброблює та по вебсокетам відсилає клієнтам. В даному випадку відправляється номерок та час проїзду.

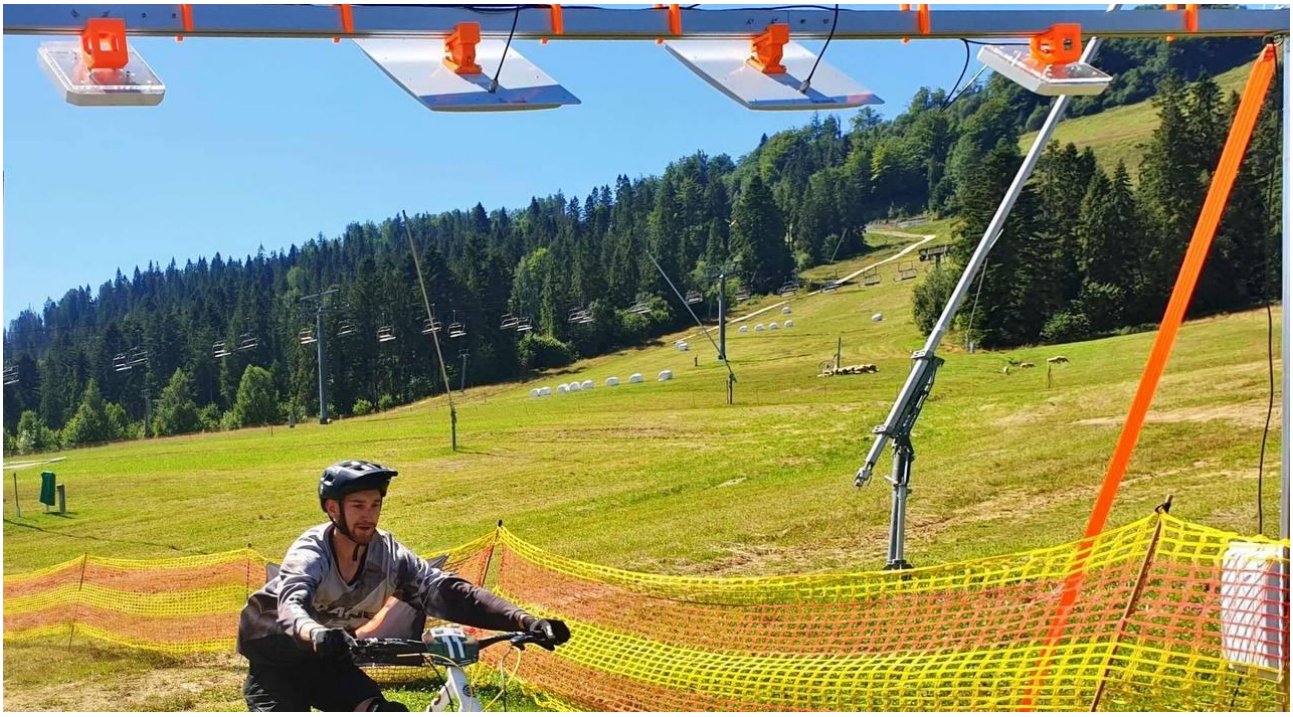

Рисунок 17 — Момент перетинання воріт фінішу

Було перевірено роботу 2 зчитувачів: Alien та Impinj. Перед стартом змагань було проведено тестовий заїзд, де першими їхали ті, хто має низьку швидкість. Тоді система Alien відпрацювала відмінно, дані передалися на сервер і відправилися клієнтам. Можна було робити висновок, що даний зчитувач підходить для системи. Він дешевий, швидко програмується виконує свої функції. Але потім відбувався заїзд професіоналів. І при цьому Alien не зчитав мітку на великій швидкості. Це є великим недоліком, бо зникає весь сенс цієї системи. Тож було проведено швидкісний заїзд із зчитувачем Impinj. У даному випадку алгоритм і система відпрацювали ідеально. Тобто на великій швидкості Impinj зчитав дані і моментально оновилися оперативні таблиці на клієнтах. При цьому тесті детально видно наскільки правильним було рішення створити базовий клас для додавання різних зчитувачів, бо тепер Impinj можна використовувати для високошвидкісних видів спорту, а Alien для менш швидких (наприклад, біг). Також дуже важливим є перспектива масштабування, тож в даному випадку ми можемо додавати різні зчитувачі, які будуть працювати по своєму, але у означеному інтерфейсі. Це пришвидшить розробку та робить систему надійною. Також з отриманих результатів зрозуміло, що використання вебсокетів працює коректно і

відповідно. Тож система виконує відповідає усім вимогам для даного запиту. І що не менш важливо система має єдину мову програмування, що спрощує зміни, пришвидшує розробку та виправлення помилок. Тобто існує три абсолютно різних застосунки, що за допомогою ReactJs пов’язані у єдине ціле, працюють по однаковим правилам та мають однаковий вигляд. Такий архітектурний підхід є найкращим рішенням для розподілених додатків.

4.2 Висновки до розділу 4

1. Використання Impinj для зчитування міток є більш оптимальним варіантом, бо він гарантує коректне зчитування даних на великих швидкості. В той самий час було виявлено, що використання Alien буде ідеальним варіантам для проведення інших змагань, таких як біг.
2. Для вирішення проблеми моментального оновлення оперативних таблиць на клієнтах системи було правильно обрано саме вебсокети, бо оновлення веб-сайту, мобільного застосунку і десктопу відбувається моментально після зміни статусу атлета.
3. Робота системи залежить не тільки від системи безпосередньо, а ще й від людського фактору. У даному випадку йдеться про операторів, які мають вчасно ставити змагання на паузу при виникненні форс-мажорної ситуації і гарно володіти системою.
4. Змагання пройшли успішно, тож система є робочою і готовою до використання. Також система легко масштабується та редагується у випадку зміни дизайну або додавання нового виду змагань.

ВИСНОВКИ

1. Досліджена проблема створення розподілених додатків єдиної системи: розглянуті переваги створення таких систем, їхні різновиди, варіанти реалізації, їхні переваги та недоліки.

2. Розроблено систему, що складається з сервера та 3 клієнтів: веб-сайт, мобільний застосунок та десктопний застосунок. Також система працює з 2 типами RFID зчитувачів Impinj та Alien, створено функціонал роботи з Arduino. Система є повністю готовою для використання на будь-яких операційних системах.

3. Досліджені результати роботи системи на реальних змаганнях з велосипедного спорту. Проаналізовано роботу Impinj та Alien: перший є дорожчим і складнішим у програмуванні, але зчитує дані з міток на великій швидкості; другий є дешевший і простіший, але не працює правильно при великій швидкості.

4. Визначено, що використання вебсокетів у розподілених додатках є ідеальним рішення для моментального оновлення інформації. Завдяки швидкості роботи ReactJs інформація на сторінках оновлюється дуже швидко.

5. Поставлена проблема використання ReactJs та NodeJs як єдиної платформи для створення розподілених додатків повністю вирішена.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fowler M. Microservices: A Definition of This Architectural Style. URL: https://martinfowler.com/articles/microservices.html (дата звернення: 03.09.2023)

2. Richardson C. Microservices Patterns: With Examples in Java. O'Reilly Media, 2018. С. 91–98.

3. Lehecka J., Kornel L. React Native in Action. Manning Publications, 2018. С. 702–707

4. Mehta M., Bovell D., Banks B., Porcello N. Learning React: Functional Web Development with React and Redux. O'Reilly Media, 2017. С. 22-23.

5. Wilson E., Strobl C., Opsian M. Node.js Design Patterns. Packt Publishing, 2014. С. 56-59

6. Hunt C. Node.js in Action. Manning Publications, 2013. С. 102-104.

7. Shaver M. Pro JavaScript Design Patterns. Apress, 2008. С. 102-104.

8. Smith K., Collins B. JavaScript Patterns. O'Reilly Media, 2010. С. 222-230.

9. Brown A. Enterprise Node.js: Designing Scalable Architecture with JavaScript. O'Reilly Media, 2013. С. 158–161.

10. Raj K. Building Scalable Apps with Redis and Node.js. O'Reilly Media, 2014. С. 158–161.

11. Triglia R. WebSocket: Lightweight Client-Server Communications. O'Reilly Media, 2015. С. 705–709.

12. Сахарова А. В., магістрант, Заєць В. І., доцент — науковий керівник. Дослідження ефективності алгоритмів обробки та покращення зображень для застосування на знімках літальних апаратів. *Молода наука-2023* : зб. наук. праць студентів, аспірантів і молодих вчених. Запоріжжя : ЗНУ, 2023. Т. 5. С. 73–75.

13. Сахарова А. В., Заєць В. І., доцент — науковий керівник. Аналіз використання ReactJs і NodeJs для створення розподілених додатків. «Актуальні питання сталого науково-технічного та соціально-економічного розвитку регіонів

України» : матеріали III Всеукр. наук. практ. конф. за участі молодих вчених. Запоріжжя : Запорізький національний університет, 2023. С. 29.

14. Franks M., Dely M., Meyarivan T. Professional WebSocket Programming: Developing Real-Time Applications with HTML5 and WebSockets. Wiley, 2013.

15. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

16. Hohpe G., Woolf B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2003.

17. WebSockets - A Conceptual Deep Dive. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (дата звернення: 10.06.2023).

18. Newman S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2015. С. 120–125.

19. Freeman E., Robson D. Head First Design Patterns. O'Reilly Media, 2004. С.110-112.

20. Flanagan D. JavaScript: The Definitive Guide. O'Reilly Media, 2020. P. 52–63.

21. Hughes B. Pro Node.js for Developers. Apress, 2013. С. 120–125.

22. Resig J., Bibeault B., Maras B. Secrets of the JavaScript Ninja. Manning Publications, 2016. P. 52–63.

23. Sommerville I. Software Engineering. Addison-Wesley, 2011. С. 120–125.

24. Hunter, Douglas, and Lima, Juliana. "Fullstack Open: The Definitive Guide for Modern JavaScript Web Development." URL: https://fullstackopen.com/ (дата звернення: 12.03.2023).

25. Документація ReactJS URL: https://react.dev/ (дата звернення : 10.10.2022).

26. Документація NodeJS URL: https://nodejs.org/en (дата звернення : 09.11.2022).

27. Документація Redux URL: https://redux.js.org/ (дата звернення : 22.12.2022).

28. Adolph, Scott, et al. "Building Evolutionary Architectures: Support Constant Change." O'Reilly Media, 2017.

29. Brown, Kyle. "Node.js Design Patterns." Packt Publishing, 2014.

30. Goncalves, Arun. "WebSocket: Lightweight Client-Server Communications." O'Reilly Media, 2015.

31. "Architectural Styles and Patterns - Tutorialspoint." URL: https://www.tutorialspoint.com/software_architecture_design/architectural_styles_and_patterns.htm (Дата звернення: 25.11.2022)

32. "RESTful API Designing guidelines — The best practices." URL: https://hackernoon.com/restful-api-designing-guidelines-the-best-practices-60e1d954e7c9 (Дата звернення: 03.02.2023)

33. "WebSocket API - Web APIs | MDN." URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket (Дата звернення: 03.02.2023)

34. "Microservices Architecture: What Is, Benefits & Examples." URL: https://www.microfocus.com/documentation/silk-performer/205/en/silkperformer-205-webhelp-en/GUID-C09A7DCD-666D-4E7D-AE9D-C147FD9F8A72.html (Дата звернення: 13.04.2023).

35. "React Native Documentation." URL: https://reactnative.dev/docs/getting-started (Дата звернення: 22.12.2022).

36. "Scalable React Apps with the Component Folder Pattern." URL: https://hackernoon.com/scalable-react-apps-with-the-component-folder-pattern-4d427a4cfaeb (Дата звернення: 13.10.2022).

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Сахарова Аліна Валеріївна, студентка 2 курсу, денної форми навчання,

Інженерного навчально-наукового інституту ім. Ю.М. Потебні ЗНУ, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz118bd-15@stu.zsea.edu.ua,

- підтверджую, що написана мною кваліфікаційна робота на тему: «**Використання ReactJs і NodeJs як єдиної платформи для розробки розподілених додатків»** відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст. 42 Закону України «Про освіту», зі змістом яких ознайомлений;

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

- згоден на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою Інтернет-системи, в також архівування моєї роботи у базі даних цієї системи.

Дата _____ Підпис _____ Сахарова Аліна Валеріївна
(студент)

Дата _____ Підпис _____Заяц Валерій Іванович
(науковий керівник)