МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ

Факультет мехатроніки та комп’ютерних технологій

Кафедра комп’ютерних наук

**КВАЛІФІКАЦІЙНА РОБОТА**

на тему:

**Порівняльний аналіз розробки користувацьких інтерфейсів зкомпонентною архітектурою і нативними мовами програмування**.

Рівень вищої освіти \_\_\_\_\_\_\_\_\_\_\_\_другий (магістерський)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Спеціальність\_\_\_\_\_\_\_\_\_122«комп’ютерні науки»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Освітня програма\_\_\_\_\_комп'ютерні науки\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Виконав студент групи МГІТ1-23\_\_\_

\_\_ Шаренко Антон Ігорович\_\_\_\_\_\_\_

Науковий керівник\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

к.т.н., доц. Колиско О.З.\_\_\_\_\_\_\_\_\_\_\_

Рецензент\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Київ 2024

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ

Факультет \_мехатроніки та комп’ютер них технологій\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра\_\_\_комп’ютерних наук\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Освітня програма \_комп’ютерні науки \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ЗАТВЕРДЖУЮ

Завідувач кафедри \_КН\_\_\_\_\_\_\_

\_\_\_\_\_\_ \_Наталія ЧУПРИНКА\_\_

«\_\_\_\_\_» \_листопада\_ 2024 р.

**ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Шаренку Антону Ігоровичу\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Тема кваліфікаційної роботи \_ *Порівняльний аналіз розробки користувацьких інтерфейсів зкомпонентною архітектурою і нативними мовами програмування*\_\_

науковий керівник роботи\_Колиско О.З. к.т.н., доц.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

затверджені наказом КНУТД від «03»\_\_\_вересня\_\_2024 року №188-уч

1. Вихідні дані до кваліфікаційної роботи *Розробка кафедри комп’ютерних наук. Матеріали з тематики дипломної роботи* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Зміст кваліфікаційної роботи (перелік питань, які потрібно опрацювати) *Розділ 1.Критичний огляд літератури, джерел, аналогів ПЗ за тематикою.*

*Розділ 2. Алгоритмічне та(або) математичне забезпечення. Розділ 3. Програмна* реалізація.

1. Дата видачі завдання *08.2024р* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# КАЛЕНДАРНИЙ ПЛАН

|  |  |  |  |
| --- | --- | --- | --- |
| №  з/п | Назва етапу кваліфікаційної роботи | Орієнтовний термін виконання | Примітка про виконання |
| 1 | Вступ | 09.10.2024 |  |
| 2 | Розділ 1. ДОСЛІДЖЕННЯ ПРОБЛЕМИ СТВОРЕННЯ РОЗПОДІЛЕНИХ ДОДАТКІВ | 12.10.2024 |  |
| 3 | Розділ 2. ДОСЛІДЖЕННЯ ЗАСОБІВ ДЛЯ РОЗРОБКИ РОЗПОДІЛЕНИХ ДОДАТКІВ | 20.10.2024 |  |
| 4 | Розділ 3. РОЗРОБКА ЗАСТОСУНКУ ДЛЯ РОЗПОДІЛЕНИХ СИСТЕМ  ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ РОБОТИ | 25.10.2024 |  |
| 5 | Висновки | 25.10.2024 |  |
| 6 | Оформлення (чистовий варіант) | 30.10.2024 |  |
| 7 | Подача кваліфікаційної роботи науковому керівнику для відгуку | 30.10.2024 |  |
| 8 | Подача кваліфікаційної роботи для рецензування | 30.10.2024 |  |
| 9 | Перевірка кваліфікаційної роботи  на наявність ознак плагіату (за 10 днів до захисту) |  |  |
| 10 | Подання кваліфікаційної роботи  завідувачу кафедри (за 7 днів до захисту) |  |  |

З завданням ознайомлений:

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_Антон ШАРЕНКО\_\_\_\_\_\_

Науковий керівник \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_Оксана КОЛИСКО\_\_\_\_\_\_

# АНОТАЦІЯ

**Шаренко А.І. Порівняльний аналіз розробки користувацьких інтерфейсів зкомпонентною архітектурою і нативними мовами програмуванн. Дипломна магістерська робота за спеціальністю 122 – «Комп’ютерні науки» – Київський національний університет технологій та дизайну, Київ, 2024 рік**

Сторінок: 85

Рисунків: 17

Таблиці: 3

Діаграма: 1

Джерел: 36

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 техно- логій для забезпечення ефективного взаємодії між клієнтською та серверною частинами, а також для підтримки масштабованості та гнучкості додатків.

Дослі дження 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

# Sharenko A.I. Comparative analysis of user interface development with component architecture and native programming languages. Master's thesis in specialty 122 - "Computer Science" - Kyiv National University of Technologies and Design, Kyiv, 2024

Pages: 85

Drawings: 17

Tables: 3

Diagram: 1

Source: 36

The aim and objectives of the research are to explore the possibilities and ad- vantages of using ReactJS and NodeJS frameworks as a unified platform for develop- ing distributed applications. The main goal is to analyze the integration of these tech- nologies 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 ana- lyzed, where ReactJS and NodeJS serve as a central foundation for efficient and mod- ular development.

Keywords: ReactJS, NodeJS, distributed applications, web development, tech- nology integration.

ЗМІСТ

Вступ…………………………………………………………………………………..7

[РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ СТВОРЕННЯ РОЗПОДІЛЕНИХ ДОДАТКІВ ЄДИНОЇ СИСТЕМИ 13](#_Toc182732142)

[1.1 Огляд проблеми створення розподілених додатків для єдиної сис- 13](#_Toc182732143)

[1.2 Архітектурні патерни для створення розподілених додатків 14](#_Toc182732144)

[1.2.1 Клієнт-Сервер 16](#_Toc182732146)

[1.2.2 Мікросервіси 17](#_Toc182732147)

[1.2.3 Шина подій (Event Bus) 22](#_Toc182732149)

[1.2.4 Шари (Layered Architecture) 2](#_Toc182732150)2

[1.2.5 Шлюз API (API Gateway) 24](#_Toc182732151)

[1.3 Сфери застосування розподілених додатків 24](#_Toc182732152)

[1.3.1 Хмарні обчислення 25](#_Toc182732153)

[1.3.2 Фінанси 26](#_Toc182732154)

[1.3.3 Мережеві технології 27](#_Toc182732155)

[1.3.4 Електронна комерція 30](#_Toc182732157)

[1.3.5 Наука та дослідження 31](#_Toc182732158)

[1.4 Аналіз програмного забезпечення для створення розподілених додатків 32](#_Toc182732159)

[1.4.1 Рішення компанії Magento 32](#_Toc182732160)

[1.4.2 Рішення компанії eBay 34](#_Toc182732163)

[1.4.3 Рішення компанії Slack 37](#_Toc182732167)

[1.4.3 Рішення компаніі Uber 39](#_Toc182732170)

[1.4.5 Рішення компанії Airbnb 41](#_Toc182732173)

[1.5 Висновки з розділу 1 43](#_Toc182732176)

[РОЗДІЛ 2 ДОСЛІДЖЕННЯ ЗАСОБІВ ДЛЯ РОЗРОБКИ РОЗПОДІЛЕНИХ ДОДАТКІВ 44](#_Toc182732177)

[2.1 Ключова роль React.js у розробці масштабованих та розподілених систем 44](#_Toc182732178)

[2.1.1 Мікросервісна Архітектура та Роль React.js 45](#_Toc182732179)

[2.1.2 Крос-платформність та розподілені додатки з використанням React Native 46](#_Toc182732180)

[2.1.3 Redux у розробці масштабованих та розподілених систем 47](#_Toc182732181)

[2.2 Роль та застосування Node.js у розробці розподілених додатків 49](#_Toc182732182)

[2.2.1 Node.js та його переваги 49](#_Toc182732183)

[2.2.2 Можливості застосування Node.js у розподілених додатках 50](#_Toc182732184)

[2.3 Вебсокети у розподілених додатках 51](#_Toc182732185)

[2.3.1 Роль вебсокетів у розподіленій архітектурі 51](#_Toc182732186)

[2.3.2 Масштабованість та забезпечення стабільності вебсокетів 52](#_Toc182732187)

[2.3.3 Переваги використання вебсокетів 53](#_Toc182732188)

[2.4 Висновки до розділу 2 53](#_Toc182732189)

[РОЗДІЛ 3 РОЗРОБКА ЗАСТОСУНКУ ДЛЯ РОЗПОДІЛЕНИХ СИСТЕМ 55](#_Toc182732190)

[3.1 Розробка інтерфейсу застосунку 55](#_Toc182732191)

[3.1.1 Інтерфейс мобільного застосунку 56](#_Toc182732192)

[3.1.2 Інтерфейс веб-сайту 58](#_Toc182732193)

[3.1.3 Інтерфейс десктопного застосунку 59](#_Toc182732194)

[3.2 Розробка програмного забезпечення 61](#_Toc182732195)

[3.2.1 Архітектура системи 61](#_Toc182732196)

[3.2.2 Розробка серверної частини 62](#_Toc182732197)

[3.2.3 Вебсокети 74](#_Toc182732198)

[3.3 Висновки до розділу 3 77](#_Toc182732199)

[РОЗДІЛ 4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ РОБОТИ КОМП’ЮТЕРНОЇ СИСТЕМИ РОЗПОДІЛЕНИХ ДОДАТКІВ 78](#_Toc182732200)

[4.1 Результати тестування роботи розподілених додатків єдиної системи 78](#_Toc182732201)

[4.2 Висновки до розділу 4 82](#_Toc182732202)

[ВИСНОВКИ 83](#_Toc182732203)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 84](#_Toc182732204)

ДОДАТОК А 87  
ДОДАТОК Б 101

# ВСТУП

## Актуальність теми

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

Використання ReactJS та Node.js дозволяє розробляти клієнтську та серве- рну частини додатків, використовуючи однакові мови програмування (JavaScript/TypeScript), спрощуючи комунікацію та підтримку проекту. Одна- кова мова програмування спрощує впровадження змін та розширення функціо- нальності додатку, оскільки розробники можуть легко розуміти і модифікувати як клієнтську, так і серверну частини, що сприяє швидкому виправленню поми- лок та розвитку проекту. Крім того, використання ReactJS та Node.js дозволяє легко обмінювати компоненти та бібліотеки між клієнтською та серверною час- тинами, 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 відкри- ває нові можливості для створення мікросервісної архітектури. ReactJS надає зручні інструменти для створення інтерфейсів користувача та клієнтської логіки, дозволяючи розробникам створювати інтерактивні та ефективні інтерфейси. Од- ночасно, Node.js забезпечує швидкість та надійність серверної частини, дозволя- ючи обробляти запити та здійснювати обмін даними з іншими сервісами. Ця

комбінація робить ReactJS та Node.js ідеальними для створення розподілених до- датків, де кожен сервіс може бути розгляданий як незалежний модуль. Ця архі- тектурна модель дозволяє командам розробників працювати над окремими сер- вісами паралельно, поліпшуючи ефективність розробки та масштабованість.

Використання ReactJS та Node.js як єдиної платформи для розробки розпо- ділених додатків відкриває безліч можливостей для розробників. Уніфікована те- хнологічна стекова архітектура дозволяє створювати компоненти як на клієнтсь- кій, так і на серверній стороні з використанням однієї мови програмування, спро- щуючи розробку та підтримку проекту. Реактивний підхід ReactJS та швидкість Node.js допомагають розробникам створювати продуктивні та масштабовані до- датки. Інтеграція з іншими технологіями та підтримка великої спільноти розро- бників роблять ReactJS та Node.js цікавими інструментами для розробки розпо- ділених 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 ство- ренні розподілених додатків, а також встановлення практичних рекомендацій і кращих практик для їх використання. Завдання включають аналіз переваг та об- межень, 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 плат- форми для розробки розподілених додатків.

## Методи дослідження

Для вирішення поставленої задачі використовуються наступні методи до- слідження:

1. Дослідження і аналіз наукових статей, книг, документації і онлайн-ре- сурсів, що стосуються використання React.js і Node.js в розподіленому програ- муванні.
2. Опитування або інтерв'ювання експертів у сфері веб-розробки, які ма- ють досвід використання React.js і Node.js.
3. Проведення практичних експериментів та розробка розподілених дода- тків 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 ство- рення 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 сфор- мулювати висновки щодо ефективності та придатності використання React.js і Node.js для розробки розподілених додатків. Висновки повинні відображати пе- реваги 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 мож- ливостей використання React.js і Node.js як єдиної платформи для розробки роз- поділених додатків, включаючи аналіз їхніх переваг та недоліків, впливу на про- дуктивність 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 цікав- ляться розподіленим програмуванням і шукають оптимальні інструменти для ро- зробки розподілених додатків.

## Практичне значення одержаних результатів

Практичне значення отриманих результатів полягає у тому, що вони мо- жуть служити джерелом інформації та рекомендацій для розробників і інженерів програмного забезпечення, які цікавляться розподіленим програмуванням. Отри- мані висновки та аналіз можуть допомогти їм вибирати найбільш оптимальні ін- струменти та підходи при створенні розподілених додатків, що в свою чергу впливає на підвищення продуктивності, масштабованості та безпеки їхніх прое- ктів. 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 опу- бліковані на XVI університетській науково-практичній конференції студентів, аспірантів, докторантів і молодих вчених — «Молода наука-2023» [12], а також на ІІІ Всеукраїнській науково-практичній конференції за участю молодих науко- вців — «Актуальні питання сталого науково-технічного та соціально-економіч- ного 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 скла- даються 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 взає- модіяти з іншими частинами програми або зовнішніми сервісами.

*Масштабованість (англ. Scalability)* — Здатність додатка працювати ефе- ктивно при збільшенні обсягів даних та навантаження.

*Мікросервіси (англ. Microservices)* — Архітектурний підхід, при якому до- даток розбивається на невеликі, незалежні компоненти, які можуть бути розгор- нуті окремо.

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

*Управління станом (англ. State Management)* — Процес визначення та ке- рування 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 стру- ктури 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 фун- кціональність додатку.

# РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ СТВОРЕННЯ РОЗПОДІЛЕНИХ ДОДАТКІВ ЄДИНОЇ СИСТЕМИ

## Огляд проблеми створення розподілених додатків для єдиної сис-

**теми**

Розробка розподілених додатків для єдиної системи є складним завданням, яке вимагає уваги до ряду ключових аспектів, щоб забезпечити ефективну та на- дійну роботу системи. Розподілені системи, що базуються на архітектурі з розді- леними 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 під- хід 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 вза- ємодією 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 ме- режевої комунікації та обробка можливих помилок є важливими аспектами.
   * Протоколи взаємодії: Визначення правильних протоколів для обміну даними між різними частинами системи допомагає у вирішенні проблем синхро- нізації та забезпеченні цілісності даних.
2. Масштабованість:
   * Горизонтальне та вертикальне масштабування: Вирішення питань мас- штабування, 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, вклю- чаючи шифрування та контроль доступу, є надзвичайно важливим.
   * Аутентифікація та авторизація: Контроль доступу до різних частин си- стеми та визначення прав користувачів є критичним для запобігання несанкціо- нованому 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 моніто- рингу для виявлення неполадок та оптимізації продуктивності системи.
   * Керування розподіленою системою: Забезпечення можливості віддале- ного керування та адміністрування системи.

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

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

## Архітектурні патерни для створення розподілених додатків

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

створення структури системи, яка може ефективно функціонувати в розподіле- ному середовищі.

Наведемо приклади 2-х даних патернів (табл. 1.1.).

# Таблиця 1.1. Приклади структурних патернів класів/об'єктів

|  |  |
| --- | --- |
| Компонувальник **(Composite) – GoF** | |
| **Проблема** | Як обробляти групу або композицію структур об'єктів одночасно? |
| **Рішення** | Визначити класи для композитних і атомарних об'єктів таким чином, щоб вони реалізовували той самий інтерфейс. |
| **Фасад (Facade) – GoF** | |
| **Проблема** | Як забезпечити уніфікований інтерфейс із набором розрізнених реалізацій або інтерфейсів, наприклад, з підсистемою, якщо небажано високе зв'язування із цією підсистемою або реалізація  підсистеми може змінитися? |
| **Рішення** | Визначити одну точку взаємодії з підсистемою – фасадний об'єкт, що забезпечує загальний інтерфейс із підсистемою й покласти на нього обов'язок по взаємодії з її компонентами. Фасад – це зовнішній об'єкт, що забезпечує єдину точку входу для служб підсистеми. Реалізація інших компонентів підсистеми закрита, її не видно зовнішнім компонентам. Фасадний об'єкт забезпечує реалізацію патерна "Стійкий до змін" з погляду захисту від змін у реалізації  підсистеми. |

|  |  |
| --- | --- |
|  |  |

*Патерни проектування поведінки (Behavioral)*

До поведінкових патернів належать:

* Інтерпретатор (Interpreter) - GoF;
* Ітератор (Iterator) або Курсор (Cursor) - GoF;
* Команда (Command), Дія (Action) або Транзакція (Транзакція) - GoF;
* Спостерігач (Observer), Опублікувати - підписатися (Publish - Subscribe) або Delegation Event Model - GoF;
* Не розмовляйте з невідомими (Don't talk to strangers) - GRASP;
* Відвідувач (Visitor) - GoF;
* Посередник (Mediator) - GoF;
* Стан (State) - GoF;
* Стратегія (Strategy) - GoF;
* Зберігач (Memento) - GoF;
* Ланцюжок обов'язків (Chain of Responsibility) - GoF;
* Шаблонний метод (Template Method) - GoF;
* Високе зачеплення (High Cohesion) - GRASP;
* Контролер (Controller) - GRASP;
* Поліморфізм (Polymorphism) - GRASP;
* Штучний (Pure Fabrication) - GRASP;

## Клієнт-Сервер

Клієнт-серверна архітектура — це модель, в якій програмне забезпечення розділене на дві основні частини: клієнтську та серверну. Клієнти та сервери вза- ємодіють між собою через мережу, дозволяючи виконання різних завдань та об- робку 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 корис- тувацький інтерфейс та взаємодію з користувачем.
2. Сервер: Сервер — це програмне забезпечення або пристрій, який слу- хає запити від клієнтів та надає їм необхідні ресурси або послуги. Серверна час- тина зазвичай містить бізнес-логіку та обробляє запити від клієнтів.

В Клієнт-Серверній Архітектурі клієнти та сервери взаємодіють через ме- режу. Клієнти генерують запити та відправляють їх на сервер для обробки, очі- куючи відповіді. Сервер слухає запити, обробляє їх та повертає результати. Роз- ділення 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 масшта- бувати незалежно. Також цей підхід дозволяє створювати клієнти для різних пла- тформ, таких як десктоп або мобільні пристрої, використовуючи спеціальні клі- єнтські додатки. Переваги включають простоту в розробці та обслуговуванні, але може виникати підвищений обсяг мережевого трафіку та залежність від доступ- ності мережі.

## Мікросервіси

Мікросервісна архітектура — це підхід до розробки програмного забезпе- чення, в якому додаток розбивається на невеликі автономні сервіси, що функці- онують 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 викорис- товуючи API. Кожен мікросервіс може бути написаний різними мовами програ- мування та використовувати різні технології, що дозволяє використовувати най- кращі інструменти для конкретного завдання.

Однією з основних переваг мікросервісної архітектури є її гнучкість та ма- сштабованість. Розбиття додатка на невеликі сервіси спрощує розробку та утри- мання 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 мо- жна розгортати окремо, що спрощує процес впровадження та зменшує ризик ви- никнення проблем при оновленнях.

Проте, перехід до мікросервісної архітектури вносить свої виклики. Сер- віси 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 роз- робки сучасних та масштабованих систем, але вимагає дбайливого планування та управління для досягнення найкращих результатів.

# Таблиця 1.2. Приклади поведінкових патернів класів/об'єктів

|  |  |  |
| --- | --- | --- |
| Ітератор (Iterator) або Курсор (Cursor) – GoF | | |
| **Проблема** | | Складений об'єкт, наприклад, список, повинен надавати доступ до своїх елементів (об'єктів), не розкриваючи їхню внутрішню структуру, причому перебирати список потрібно по-різному залежно  від завдання. |
| **Рішення** | | Створюється клас "Ітератор", який визначає інтерфейс для доступу і перебору елементів, "КонкретнийІтератор" реалізує інтерфейс класу "Ітератор" і стежить за поточною позицією при обході "Агрегат". "Агрегат" визначає інтерфейс для створення об'єкту - ітератора.  "КонкретнийАгрегат" реалізує інтерфейс створення ітератора і |
|  | повертає екземпляр класу "КонкретнийІтератор", "КонкретнийІтератор" відстежує поточний об'єкт в агрегаті і може обчислити наступний об'єкт при переборі.  Даний патерн підтримує різні способи перебору агрегату. |
| **Відвідувач (Visitor) – Go** | |
| **Проблема** | Над кожним об'єктом деякої структури виконується операція.  Визначити нову операцію, не змінюючи класи об'єктів. |
| **ішення** | Клієнт, що використовує даний патерн, повинен створити об'єкт класу "КонкретнийВідвідувач", а потім відвідати кожен елемент структури. "Відвідувач", оголошує операцію "Відвідати" для кожного класу "КонкретнийЕлемент" (ім'я та сигнатура даної операції ідентифікують клас, елемент якого відвідує "Відвідувач" - тобто, відвідувач може звертатися до елемента прямо). "КонкретнийВідвідувач" реалізує всі операції, оголошені в класі "Відвідувач". Кожна операція реалізує фрагмент алгоритму, визначеного для класу відповідного об'єкта в структурі.  Клас "КонкретнийВідвідувач" надає контекст для цього алгоритму і зберігає його локальний стан. "Елемент" визначає операцію "Прийняти", яка приймає "Відвідувача" як аргумент, "КонкретнийЕлемент" реалізує операцію "Прийняти", яка приймає "Відвідувача" як аргумент. "СтруктураОб’єкта" може перерахувати свої аргументи і надати відвідувачеві високорівневий інтерфейс для відвідування своїх елементів.  Даний патерн логічно використовувати, якщо в структурі присутні об'єкти багатьох класів з різними інтерфейсами, і необхідно виконати над ними операції, що залежать від конкретних класів, або якщо класи, встановлюють структуру об'єктів, змінюються рідко, але нові операції над цією структурою додаються часто.  Даний патерн спрощує додавання нових операцій, об'єднує  споріднені операції в класі "Відвідувач". |

|  |  |
| --- | --- |
|  | У даному патерні утруднено додавання нових класів "КонкретнийЕлемент", оскільки потрібне оголошення нової абстрактної операції в класі "Відвідувач". |
| **Стан (State) – Go** | |
| **Проблема** | Варіювати поводження об'єкта залежно від його внутрішнього стану |
| **Рішення** | Клас "Контекст" делегує залежать від стану запити поточному об'єкту "КонкретнийСтан" (зберігає екземпляр підкласу "КонкретнийСтан", яким визначається поточний стан), і визначає інтерфейс, що представляє інтерес для клієнтів. " КонкретнийСтан" реалізує поведінку, асоційовану з якимось станом об'єкта "Контекст". "Стан" визначає інтерфейс для інкапсуляції поведінки, асоційованого з конкретним екземпляром "Контексту".  Даний патерн локалізує залежне від стану поведінку і ділить його на частини, що відповідають станам, переходи між станами стають явними. |

## 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 по- дії та надсилають їх на шину. З іншого боку, компоненти, які реагують на події, виступають як консюмери, які підписуються на отримання певних подій та реа- гують на них.

Шина подій використовується для вирішення проблеми залежностей між компонентами системи, забезпечуючи їхню взаємодію через локальний або гло- бальний рівень подій. Це полегшує розвиток та супровід системи, оскільки ком- поненти 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 по- дії 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 розподі- лених та легко масштабованих систем, проте вона повинна використовуватися з обачливістю та урахуванням конкретних вимог системи.

## Шари (Layered Architecture)

Архітектурний паттерн "Шари" (Layered Architecture) є стратегією органі- зації програмного забезпечення, в якій система поділяється на логічні шари, ко- жен з яких виконує конкретний функціональний набір. Кожен шар представляє собою групу пов'язаних функцій та сервісів, і взаємодіє тільки зі шарами, розта- шованими 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 ін- терфейс користувача, контролери, та все, що відображається на екрані користу- вача.
      * 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 різнома- нітні служби та компоненти, необхідні для підтримки роботи всіх інших шарів. Сюди входять різноманітні сервіси, інструменти безпеки, журналювання, інстру- менти тестування, тощо.

Використання шарової архітектури дозволяє забезпечити високу модуль- ність 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 ефекти- вно та ефективно. Занадто велика кількість шарів може призвести до зайвого під- вищення складності та непотрібного розділення логічно пов'язаних компонентів. Узагальнюючи, шарова архітектура відображає підхід, що дозволяє логі-

чно та функціонально розділити складні системи на менші та керовані компоне- нти для полегшення розробки та обслуговування.

## Шлюз API (API Gateway)

Шлюз API (API Gateway) представляє собою ключовий архітектурний пат- терн 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 додат- ків.

Шлюз API виконує роль маршрутизатора, направляючи запити до відпові- дних мікросервісів, що дозволяє ефективно розподіляти та контролювати наван- таження. 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 за- безпечує 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 змен-

шення навантаження на мікросервіси[28].

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

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

## Сфери застосування розподілених додатків

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

## Хмарні обчислення

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

1. Еластичність обчислювальних ресурсів: Хмарні обчислення дозволя- ють організаціям динамічно масштабувати свої обчислювальні потреби в залеж- ності від обсягів роботи. Це забезпечує ефективне використання ресурсів та оп- тимізацію витрат.
2. Забезпечення доступу до обчислювальної потужності: Користувачі мо- жуть без зайвих зусиль отримувати доступ до потужних обчислювальних ресур- сів через мережу. Це особливо корисно для стартапів та малих підприємств, які не мають власної інфраструктури.
3. Забезпечення безпеки та надійності: Хмарні сервіси забезпечують ви- сокий рівень безпеки та автоматичної резервної копії, що робить їх привабли- вими для компаній, які прагнуть захистити свої дані та послуги від втрати чи атак.
4. Обробка великих обсягів даних: Розподілені системи в хмарних обчис- леннях 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 поту- жність інноваціям та розвитку в різних галузях.

## Фінанси

В сучасному фінансовому світі, де обробка великих обсягів фінансових да- них та забезпечення безпеки транзакцій є критично важливими завданнями, роз- поділені 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 використову- ються для швидкої та ефективної обробки великих обсягів фінансових транзак- цій. Вони дозволяють фінансовим установам в режимі реального часу викону- вати операції, вести облік та моніторити ризики.
2. Управління ризиками та безпекою: Розподілені системи грають важ- ливу роль у виявленні та управлінні ризиками у фінансовому секторі. Вони за- безпечують аналіз та моніторинг фінансових операцій, виявлення підозрілих ак- тивностей та забезпечення безпеки фінансових даних.
3. Електронна комерція та онлайн-платежі: Розподілені додатки в елект- ронній комерції дозволяють забезпечити безпеку та швидкість онлайн-платежів. Вони інтегрують різні платіжні системи та забезпечують зручний доступ до фі- нансових 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 стра- тегії 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 ймовір- ність помилок.

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

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

## Мережеві технології

У світі постійних технологічних інновацій та зростаючої важливості забез- печення зв'язності та швидкості обміну інформацією, мережеві технології вияв- ляються необхідним інструментом для підтримки різноманітних сфер. Розподі- лені додатки у сфері мережевих технологій сприяють побудові надійних та ефе- ктивних мереж, що об'єднують різноманітні пристрої та сервіси:

1. Інтернет речей (IoT): Розподілені додатки використовуються для об'єд- нання 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 рі- вень якості обслуговування для різних видів трафіку, що є ключовим для ефек- тивного функціонування мереж[31].
3. Керування мережами: У сфері мережевих технологій розподілені дода- тки використовуються для автоматизації процесів конфігурації, моніторингу та управління мережевим обладнанням, що дозволяє забезпечити оптимальну ефе- ктивність 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 ство- рення та управління системами мережевої безпеки, виявлення загроз та реагу- вання 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 використову- ються 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 високопродукти- вних інформаційних систем. Вони забезпечують інтеграцію та взаємодію різно- манітних пристроїв та платформ, розвиваючи потенціал сучасних мереж.

# Таблиця 1.3. Приклади що породжують патерни класів/об'єктів

|  |  |
| --- | --- |
| **Одинак (Singleton) – Go** | |
| **Проблема** | Який спеціальний клас повинен створювати "Абстрактну фабрику" і як одержати до неї доступ? Необхідний лише один екземпляр спеціального класу, різні об'єкти повинні звертатися до цього  екземпляра через єдину точку доступу. |
| **Рішення** | Створити клас і визначити статичний метод класу, що повертає цей єдиний об'єкт. Розумніше створювати саме статичний екземпляр спеціального класу, а не оголосити необхідні методи статичними, оскільки при використанні методів екземпляра можна застосувати механізм спадкування й створювати підкласи. Статичні методи в мовах програмування не поліморфні й не допускають перекриття в похідних класах. Рішення на основі створення екземпляра є більше гнучким, оскільки згодом може знадобитися вже не єдиний екземпляр об'єкта. |
| **Фабричний метод (Factory Method) або Віртуальний конструктор (Virtual**  **Constructor) – Go** | |
| **Проблема** | Визначити інтерфейс для створення об'єкту, але залишити підкласам рішення про те, який клас інстанціювати, тобто, делегувати  інстанціювання підкласами. |
| **Рішення** | Абстрактний клас "Творець" оголошує Фабричний Метод, який повертає об'єкт типу "Продукт" (абстрактний клас, що визначає інтерфейс об'єктів, що створюються фабричним методом). "Творець" також може визначити реалізацію за замовчуванням Фабричного Методу, який повертає "КонкретнийПродукт". "КонкретнийТворець" заміщає Фабричний Метод, який повертає об'єкт "КонкретнийПродукт". "Творець" "покладається" на свої підкласи у визначенні Фабричного Методу, що повертає об'єкт "КонкретнийПродукт". Даний патерн позбавляє проектувальника від необхідності вбудовувати в код залежать класів від програми. Однак при застосуванні даного патерну виникає додатковий рівень  підкласів. |

|  |  |
| --- | --- |
|  |  |

Архітектурні системні патерни

*Структурні патерни*

До структурних патернів належать:

* Репозиторій;
* Клієнт / сервер;
* об'єктно - орієнтований, Модель предметної області (Domain Model), модуль таблиці (Data Mapper);
* Багаторівнева система (Layers) чи абстрактна машина;
* Потоки даних (конвеєр або фільтр).

## Електронна комерція

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

1. Ефективний управління складом: Розподілені додатки в електронній комерції допомагають підприємствам ефективно управляти запасами та складсь- кими запасами. Вони автоматизують процеси замовлення, відстеження товарів та управління запасами[32].
2. Персоналізовані рекомендації та маркетинг: Розподілені системи вико- ристовують дані користувачів для створення персоналізованих рекомендацій та маркетингових пропозицій, що сприяє збільшенню конверсії та задоволенню по- купців.
3. Безпека та обробка платежів: У сфері електронної комерції безпека є пріоритетом. Розподілені додатки використовуються для шифрування даних, об- робки платежів та захисту особистої інформації покупців.
4. Оптимізація процесів доставки та логістики: Розподілені системи уп- равління логістикою та доставкою дозволяють оптимізувати маршрути, відсте- жувати вантажі та надавати користувачам точну інформацію про стан їхніх за- мовлень.
5. Мультиканальний продаж: Електронна комерція все більше розвива- ється 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 про- дажу.
7. Інтеграція з соціальними мережами: Розподілені додатки дозволяють ефективно інтегрувати функціональність електронної комерції з соціальними ме- режами, сприяючи розширенню аудиторії та підвищенню взаємодії з клієнтами.

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

## Наука та дослідження

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

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 спів- робітництва[33].
2. Обчислювальні задачі та суперкомп'ютери: У наукових досліджен- нях, де потрібні великі обчислювальні потужності, розподілені додатки

використовуються для розв'язання складних математичних задач та моделю- вання процесів в реальному часі.

1. Обробка та аналіз великих даних: В сфері наукових досліджень роз- поділені системи використовуються для обробки та аналізу великих обсягів да- них, що дозволяє вченим виявляти тенденції, робити висновки та розробляти нові наукові гіпотези.
2. Віртуальні лабораторії та експерименти: Розподілені додатки дозво- ляють створювати віртуальні лабораторії та експерименти, що полегшує доступ до наукових засобів та сприяє інноваціям у різних галузях.
3. Інтелектуальна обробка даних: Розподілені системи використовують технології штучного інтелекту та машинного навчання для автоматизації обро- бки даних, виявлення патернів та покращення якості досліджень.
4. Електронне публікування та відкритий доступ: Розподілені додатки сприяють електронному публікуванню наукових робіт та забезпечують відкри- тий доступ до наукової інформації, що сприяє швидкому обміну знаннями.
5. Мережева безпека та конфіденційність даних: В наукових дослі- дженнях, 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 іннова- ції, полегшуючи спільну роботу вчених, розвиваючи технології та забезпечуючи доступ до знань у глобальному масштабі.

## Аналіз програмного забезпечення для створення розподілених додатків

## Рішення компанії Magento

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

Заснована в 2008 році, компанія вже більше десяти років лідирує у своєму сег- менті 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 ба- зами 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 уп- равління базами даних (RDBMS), MySQL використовується Magento для збері- гання та організації великих обсягів структурованих даних[10].
3. Хмарні технології:
   * Хмарна інфраструктура: Magento надає можливості хмарного розгор- тання, що дозволяє компаніям використовувати віртуальні ресурси у хмарі для забезпечення високої доступності та ефективної масштабованості під час зрос- тання бізнесу.
4. HTML, CSS, та JavaScript:
   * Використовується для створення структури веб-сторінок. Відповідає за стилізацію та оформлення веб-сайту, надає зовнішній вигляд. Використову- ється для додавання інтерактивності та динаміки на сторінці, забезпечуючи ко- ристувачеві більше функціональних можливостей без перезавантаження сторі- нки.
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 плат- форми.

Ці технології взаємодіють, створюючи потужну та масштабовану платфо- рму 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 функці- ональні веб-сайти.
3. Мікросервісна архітектура полегшує розробку та масштабування ком- понентів 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.

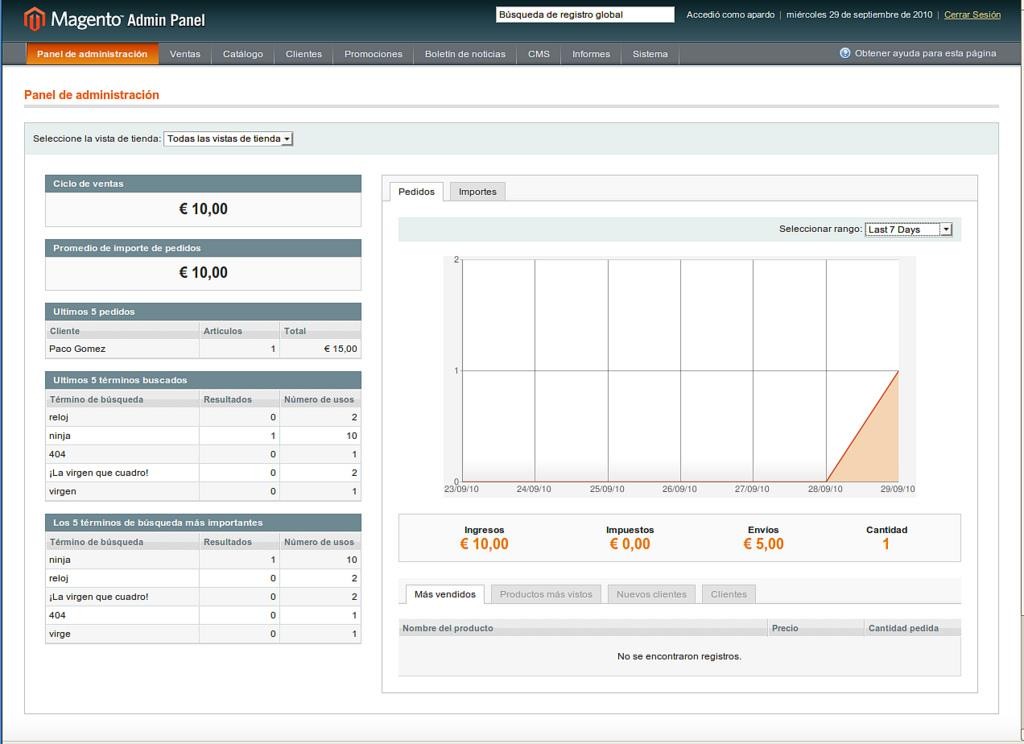


Рисунок 1 — Рішення продукту Magento

## Рішення компанії eBay

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

1. Java: eBay використовує Java як основну мову програмування для роз- робки своїх розподілених додатків. Java забезпечує високу продуктивність та ма- сштабованість, що необхідно для обробки великого обсягу транзакцій та взаємо- дії 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 кліє- нтами 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 ефектив- ному функціонуванню платформи в умовах великої кількості користувачів та транзакцій.

eBay продовжує бути інноваційним лідером у світі електронної комерції, по- єднуючи в собі високотехнологічні рішення та глобальний вплив для забезпе- чення 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 мо- вами програмування.
2. Хоча Node.js ефективно обробляє багато одночасних з'єднань, він може не бути оптимальним для всіх частин системи, особливо, якщо вимагається ве- лика обчислювальна потужність.
3. Управління розподіленою архітектурою може вимагати додаткового часу та навичок.

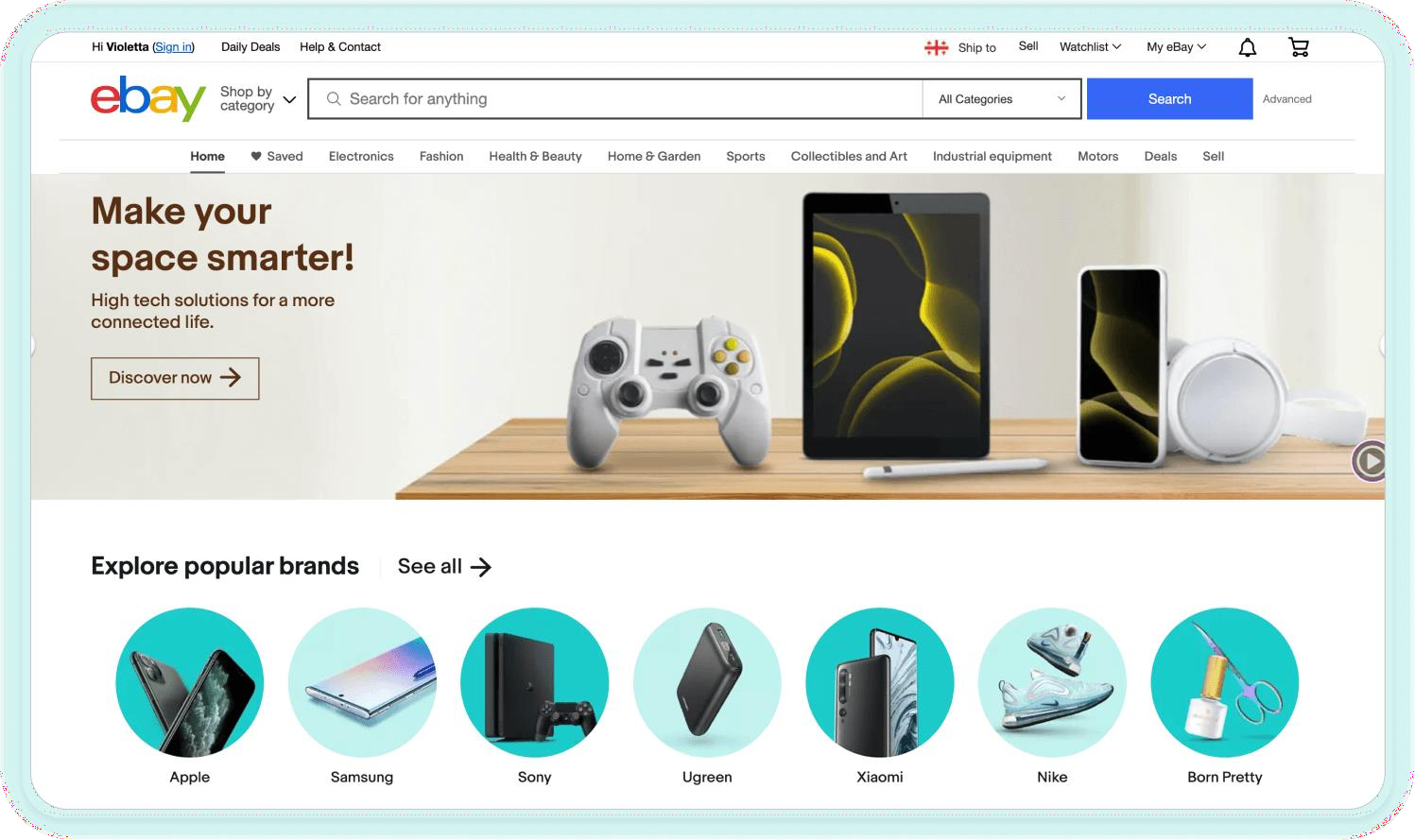
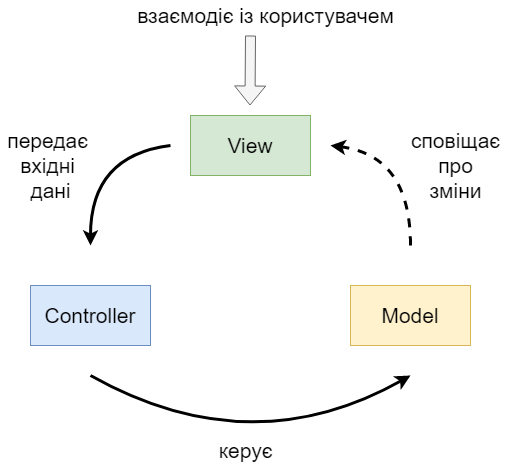
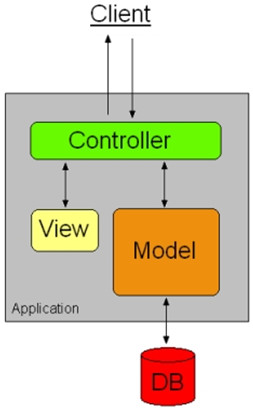


Рисунок. 2 — Рішення компанії eBay





1.4.1. Діаграма послідовностей

## Рішення компанії Slack

Slack є визнаним лідером у сегменті комунікаційних платформ, спеціалізо- ваних 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 оновлю- вати елементи веб-сторінки без необхідності її перезавантаження.
   * Electron: Це фреймворк для розробки крос-платформених десктоп-до- датків з використанням веб-технологій[11]. Slack використовує Electron для ство- рення 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 повідом- леннями між користувачами. RTM дозволяє зберігати взаємодію в режимі

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

1. Microservices Architecture:
   * Мікросервісна архітектура розбиває систему на невеликі і автономні модулі, які можуть функціонувати незалежно один від одного. Це полегшує роз- гортання, розвиток та масштабування окремих компонентів. Кожен сервіс відпо- відає 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 ві- дстані.

## 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 пові- домленнями, важливу для ефективної комунікації в реальному часі.
4. Microservices Architecture полегшує розгортання та масштабування ок- ремих компонентів системи.

## Недоліки технологій Slack:

1. JavaScript може бути обмеженою для складних обчислювальних опера-

цій.

1. Electron може споживати більше ресурсів системи порівняно з натив-

ними додатками.

1. RTM має високі вимоги до мережевого з'єднання, що може впливати на якість обміну повідомленнями.
2. Управління багатьма мікросервісами може вимагати додаткових зусиль та ресурсів.

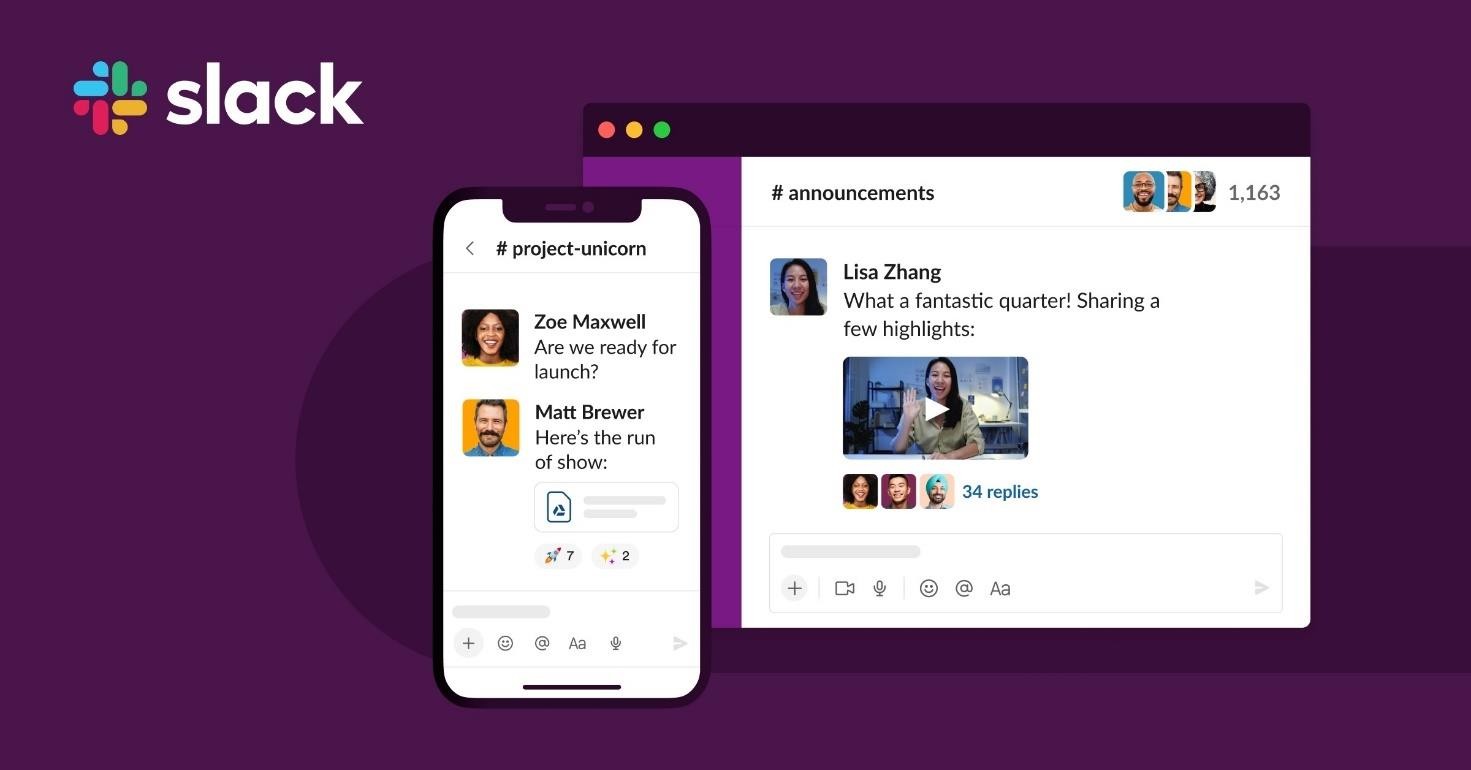


Рисунок 3 — Рішення продукту Slack

## Рішення компаніі Uber

Uber — це інноваційна технологічна компанія, яка змінила ландшафт паса- жирських перевезень. Заснована в 2009 році, Uber надає платформу для замов- лення 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 за- стосунку. Ця платформа дозволяє створювати ефективні та масштабовані серве- рні додатки, що важливо для обробки великого обсягу запитів[20].
   * Python: Python використовується для розробки деяких серверних ком- понентів 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 ви- користовує реляційні бази даних, такі як MySQL та PostgreSQL.
   * Cassandra: Нереляційна база даних Cassandra використовується для об- робки 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 мож- ливість створювати швидкі та ефективні додатки для обох платформ.
2. Використання Node.js для серверного застосунку забезпечує ефектив- ність 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) забезпе- чує надійність та управління даними.
5. Використання нереляційної бази даних Cassandra сприяє обробці вели- кого 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.

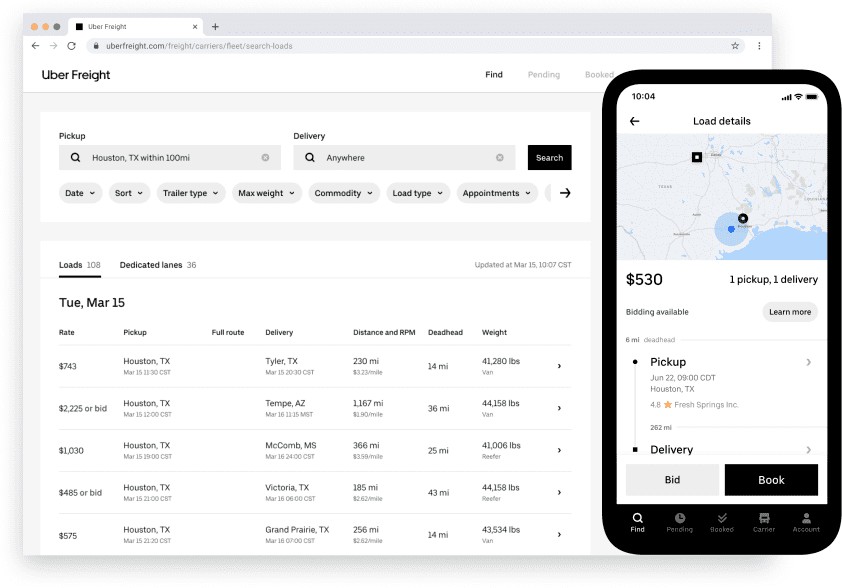


Рисунок 4 — Рішення компанії Uber

## 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, до- зволяючи людям здається або орендовувати житло в більш неперетворених ло- каціях.

1. Мова Програмування:
   * JavaScript та React: Airbnb використовує JavaScript для розробки веб- інтерфейсу свого сайту та React — для створення динамічних та ефективних ко- ристувацьких інтерфейсів[21].
   * Swift та Kotlin: Додатки Airbnb для iOS та Android розроблені з викори- станням 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 ін- струменти.
   * Java та Scala: Деякі частини бекенду Airbnb написані на Java та Scala, що дозволяє компанії працювати з великою кількістю даних та забезпечує мас- штабованість.
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 ін- ших елементів системи.

Airbnb вдало поєднує технологічну інноваційність з високоякісним корис- тувацьким досвідом, створюючи ефективні розподілені додатки для своїх міль- йонів користувачів.

## Переваги технологій в розробці Airbnb:

1. Використання JavaScript та React для веб-інтерфейсу дозволяє створю- вати динамічні та ефективні користувацькі інтерфейси.
2. Використання Swift та Kotlin для мобільних додатків гарантує ефекти- вність та високу якість користувацького досвіду.
3. Використання Ruby on Rails для серверного застосунку спрощує розро- бку та надає зручні інструменти для розробників.
4. Використання Java та Scala в бекенді дозволяє компанії ефективно пра- цювати 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 уп- равління 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 об- межити швидкодію та масштабованість в разі значного збільшення обсягу даних.

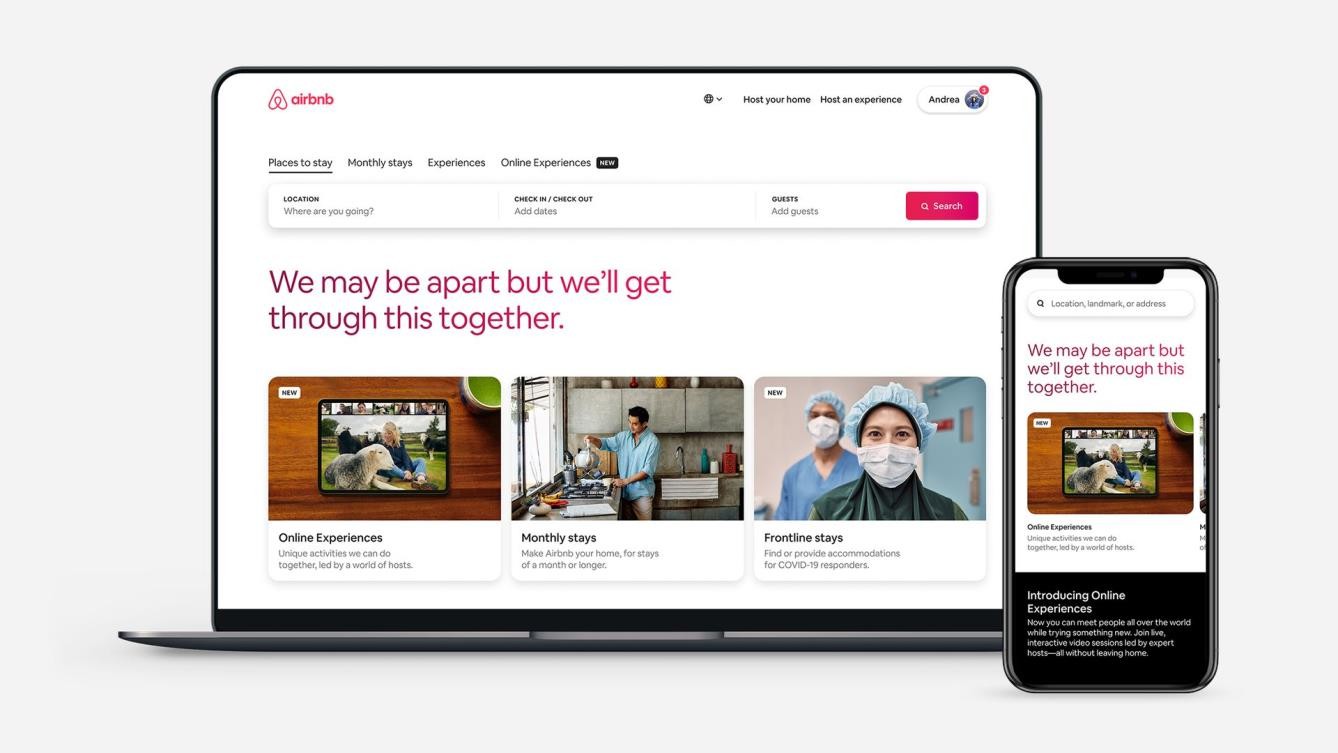


Рисунок 5 — Рішення компанії Airbnb

## 1.5 Висновки з розділу 1

1. У даному розділі була розглянута актуальна проблема створення роз- поділених додатків для єдиної системи, що визначає важливість подальших дос- ліджень у цій області.
2. Дослідження дозволило виокремити основні виклики, пов'язані з роз- робкою розподілених додатків, та надати усвідомлення про їх вплив на сучасну ІТ-сферу.
3. Аналіз сучасного стану розподілених додатків підкреслив наявні тен- денції та технології, що використовуються, що може послужити важливим вихі- дним пунктом для майбутнього розвитку.
4. Результати дослідження свідчать про важливість подальшого вдоско- налення 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 розподіле- них додатків.

# РОЗДІЛ 2 ДОСЛІДЖЕННЯ ЗАСОБІВ ДЛЯ РОЗРОБКИ РОЗПОДІЛЕНИХ ДОДАТКІВ

## Ключова роль React.js у розробці масштабованих та розподілених систе

React.js, або просто React, представляє собою потужний фреймворк для ро- зробки інтерфейсів, який визначається своєю виразною та зручною для розроб- ників 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 оптимі- зувати процеси оновлення та відображення даних, забезпечуючи високу продук- тивність 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 гну- чкості, що робить його ідеальним інструментом для тих, хто прагне розробляти ефективні та масштабовані веб-додатки. Відділення компонентів, зручне керу- вання станом, та використання JSX (розширеного синтаксису JavaScript) роблять React зрозумілим та досить привабливим для широкого спектру розробників.

React.js виявляється надзвичайно корисним у забезпеченні масштабовано- сті 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 онов- лення 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 ма- сштабованих і адаптивних інтерфейсів. Вона дозволяє розбити інтерфейс на не- великі та незалежні компоненти, які можна використовувати окремо. Це полег- шує 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 ком- поненти можуть функціонувати незалежно один від одного. Це сприяє гнучкості та ефективності управління різними частинами інтерфейсу, роблячи їх легко ма- сштабованими при рості обсягів даних та користувацького трафіку.

## Мікросервісна Архітектура та Роль React.js

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

1. Незалежні Компоненти: React.js дозволяє розбити інтерфейс на неве- ликі, самодостатні компоненти. Ці компоненти можуть функціонувати незале- жно один від одного, що ідеально підходить для мікросервісної архітектури. Ко- жен компонент може бути розроблений, розгорнутий та масштабований окремо, спрощуючи процес розробки та підтримки.
2. Легке Інтегрування: Завдяки компонентній природі React.js, нові фун- кції 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.

1. Спільне Використання Компонентів: React.js дозволяє використову- вати компоненти повторно. У мікросервісній архітектурі це може призвести до створення бібліотеки готових до використання компонентів, яку можна викори- стовувати в різних частинах системи.
2. Легкість Тестування: Тестування окремих компонентів на React.js стає простішим завдяки їхній ізольованій природі. Це полегшує виявлення та випра- влення помилок, а також додає впевненості у стабільності мікросервісів.
3. Масштабованість: Через можливість розробки та масштабування окре- мих компонентів, React.js сприяє масштабованості системи в цілому. Велика кі- лькість мікросервісів може працювати паралельно, ефективно оброблюючи ве- ликий потік запитів.

React.js відкриває нові можливості для створення розподілених систем, працюючи як потужний інструмент для розробки масштабованих та гнучких мі- кросервісів[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 можливо- сті 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 до- зволяє 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 реа- гування на зміни важлива.

1. Ефективна Підтримка та Оновлення: Оскільки React Native дозволяє використовувати спільний код, підтримка та оновлення додатків на обох плат- формах стає більш ефективною. Виправлення помилок та впровадження нового функціоналу можуть бути внесені для всіх користувачів одночасно.
2. 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 пристро- ями.
3. Нативна Ефективність та Взаємодія: Використання React Native дозво- ляє взаємодіяти з нативними компонентами платформ, забезпечуючи високу ефективність та якість використання додатків.
4. Адаптивність до Різних Умов: Крос-платформеність React Native до- зволяє 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 стра- тегічно важливим для створення ефективних, швидких та масштабованих мобі- льних додатків.

## Redux у розробці масштабованих та розподілених систем

Redux виявляється ключовим актором у світі розробки масштабованих та розподілених систем, надаючи ефективний механізм для управління станом до- датків 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 фронте- нду 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 ста- нів 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 відла- годження[15]. Розробники можуть ефективно відстежувати та аналізувати зміни в системі.
6. Плагінні Можливості:
   * Redux надає можливості для використання плагінів, що полегшує нала- штування 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 по- винна бути актуальною.
8. Виклики та Рекомендації:
   * Вивчення Redux та його відповідність конкретним потребам є важли- вим. Обізнаність із засадами бібліотеки дозволяє розробникам максимально ви- користовувати її переваги[27].
   * Обговорення вибору Redux повинно базуватися на вимогах конкрет- ного 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.

## Роль та застосування Node.js у розробці розподілених додатків

## 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 спро- щеній 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 ко- рисно 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 си- стеми.

## 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 мо- дель роботи дозволяють ефективно взаємодіяти з багатьма одночасними з'єднан- нями, 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 об- робляти велику кількість запитів одночасно[16]. У розподілених додатках, де ва- жлива невідкладність та продуктивність, це забезпечує плавну взаємодію з різ- ними компонентами системи. Однією з головних переваг Node.js є здатність ефе- ктивно 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 дода- тках це стає важливим фактором оптимізації та забезпечення високої ефективно- сті.

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

## Вебсокети у розподілених додатках

## 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 інте- рактивне з'єднання, дозволяючи серверу надсилати сповіщення клієнтам без їх- нього 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 нава- нтаженості на мережу, вебсокети є ефективним інструментом для масштабу- вання. Вони можуть обслуговувати багато одночасних з'єднань без втрати про- дуктивності.
5. Постійний зв'язок: Вебсокети забезпечують постійне з'єднання, що до- зволяє уникнути необхідності постійно підключатися та відключатися, як це ві- дбувається при кожному HTTP-запиті. Це зменшує накладні витрати та сприяє ефективності[18].

Таким чином, вебсокети відіграють ключову роль у розподіленій архітек- турі, забезпечуючи ефективний та масштабований зв'язок між різними части- нами системи.

## Масштабованість та забезпечення стабільності вебсокетів

Масштабованість та забезпечення стабільності вебсокетів є ключовими ас- пектами для ефективного використання цієї технології в розподіленій архітек- турі. Важливі аспекти цих питань включають:

1. Масштабованість:

* Архітектурний дизайн: При проектуванні системи з вебсокетами слід ураховувати можливість масштабування. Використання архітектурних паттер- нів, таких як шини подій або мікросервісна архітектура, дозволяє легко розши- рювати та додавати нові компоненти системи.
* Навантаження та ресурси: Ефективне керування навантаженням важ- ливе для забезпечення масштабованості. Використання навантажувачів та бала- нсерів навантаження може допомогти розподілити завдання між різними серве- рами[14].

1. Стабільність:

* Обробка помилок: Ретельна обробка помилок та винятків дозволяє уни- кати витоків та зберігати стабільність системи в цілому. Резервне зберігання да- них та механізми відновлення можуть забезпечити надійність у випадках виник- нення 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 наванта- ження та вдосконалити роботу системи.

1. Балансування:

* Балансування навантаження: Ефективне розподілення трафіку між різ- ними серверами допомагає уникнути перевантаження та забезпечити рівномірне розподілений навантаження. Балансери навантаження можуть автоматично реа- гувати на зміни навантаження та перенаправляти трафік.

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

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

## Переваги використання вебсокетів

1. Вебсокети дозволяють миттєвий обмін даними між клієнтом і сервером у режимі реального часу. Це особливо корисно для розподілених систем, де пот- рібно негайно оновлювати інформацію на всіх з'єднаних пристроях.
2. Завдяки постійному відкритому з'єднанню вебсокетів, затримка пере- дачі даних значно знижується порівняно з традиційними методами, такими як HTTP. Це дозволяє отримати швидку та ефективну взаємодію між клієнтом і сер- вером.
3. Вебсокети відмінно справляються з обробкою багатьох одночасних з'єднань, що робить їх ідеальним рішенням для систем з великою кількістю ко- ристувачів або пристроїв.
4. У випадку втрати з'єднання, вебсокети автоматично намагаються від- новити з'єднання, що забезпечує стабільну і надійну роботу додатка при нестабі- льному Інтернет-з'єднанні[36].
5. Для реалізації функціоналу реального часу вебсокети спрощують ви- вчення 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 дода- тки та десктопні застосунки.

## 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 ві- ртуального DOM, що робить його потужним інструментом для фронтенд розро- бки.
4. Node.js використовується для серверної розробки, забезпечуючи асин- хронну 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

## Розробка інтерфейсу застосунку

Інтерфейс застосунку було розроблено відповідно до задачі за допомогою якої буде доцільно передати користь від використання ReactJs & NodeJs у розпо- ділених 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 скла- дається з сервера та 3 видів клієнтів: мобільний застосунок, десктопний застосу- нок та веб-сайт.

Мобільний застосунок використовується на старті – за допомогою нього оператор вмикає ворота старта, будується черга і кожен атлет починає свій мар- шрут 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 мобіль- ному застосунку.

Десктопний застосунок використовується на фініші – він напряму зв’яза- ний 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 вида- лити.

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.

## Інтерфейс мобільного застосунку

На головному екрані зображено список змагань, які є відкритими і доступ- ними до перегляду.(див. рис. 7) При натисканні на табличку відкривається спи- сок 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.

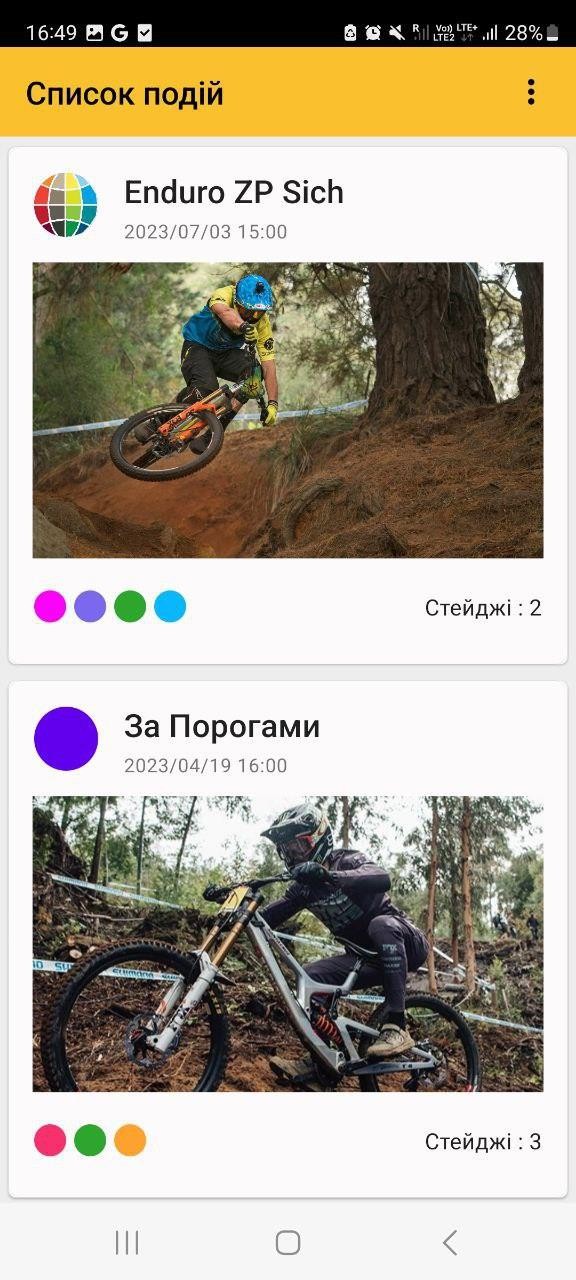


Рисунок 7 — Головна сторінка мобільного застосунку

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 підклю- чення до воріт старту. В залежності від кольору ми розуміємо чи ворота підклю- чені, чи відкриті, чи в режимі очікування. Також біля кожного атлету є контекс- тне меню: воно використовується у випадку якщо ми хочемо відмінити атлета або змінити порядок черги.

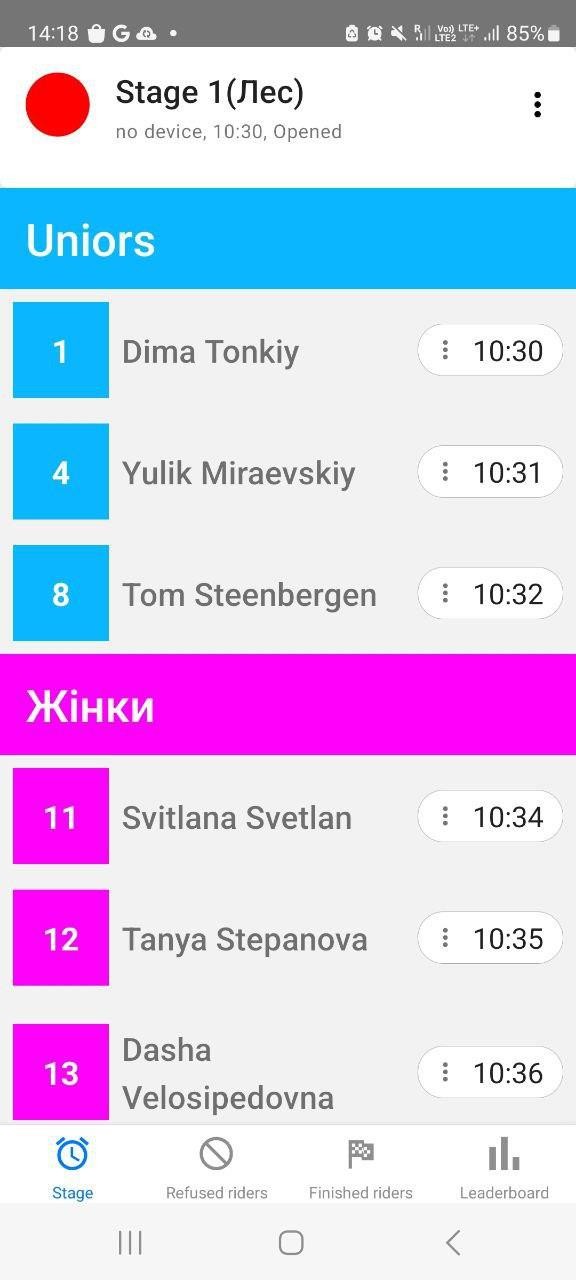


Рисунок 8 — Сторінка Стейджу та черги мобільного застосунку

## 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 зма- гань: дізнатися час та місце.(Див. рис. 10)

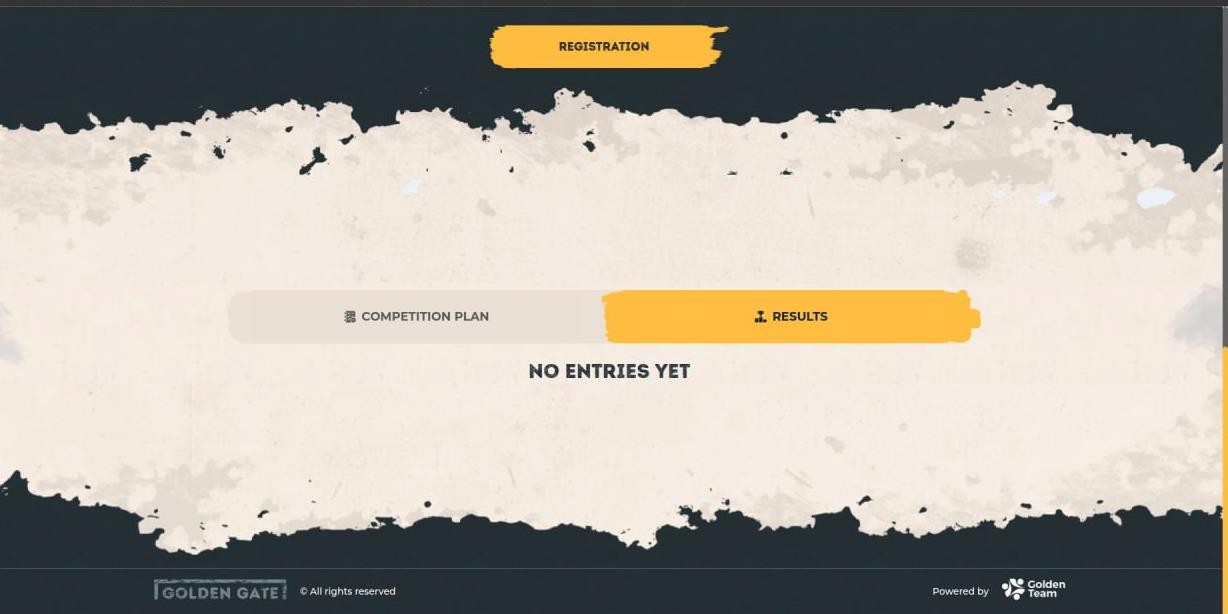


Рисунок 10 — Сторінка результатів

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

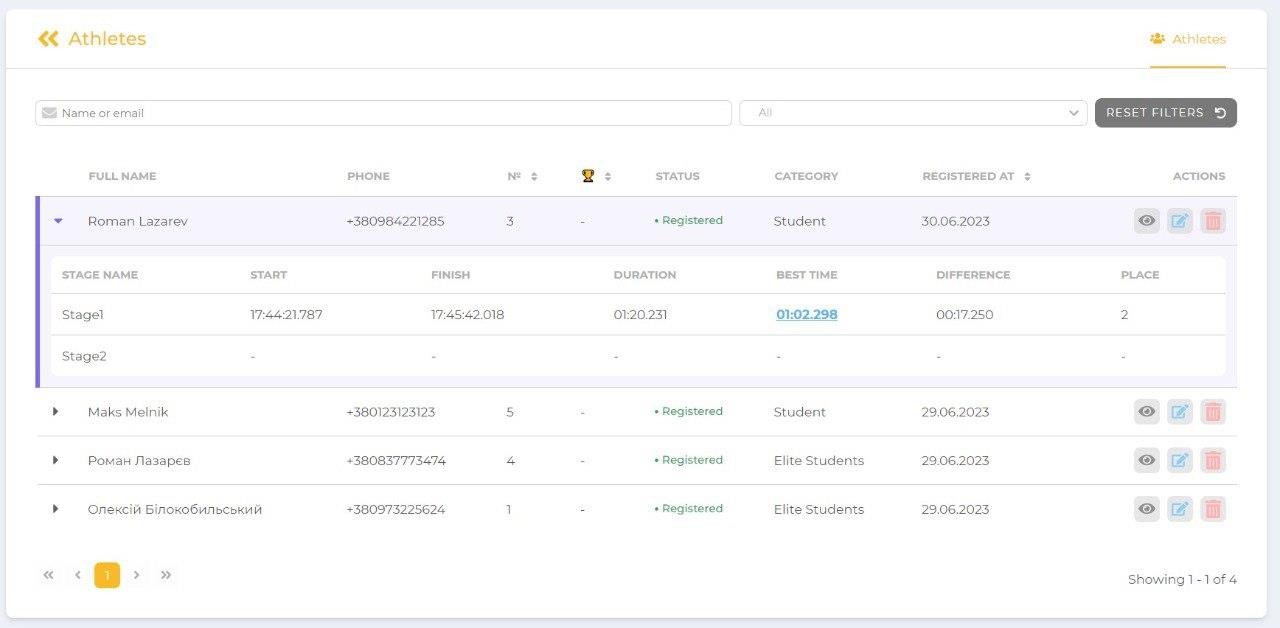


Рисунок 11 — Таблиця атлетів

## 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.

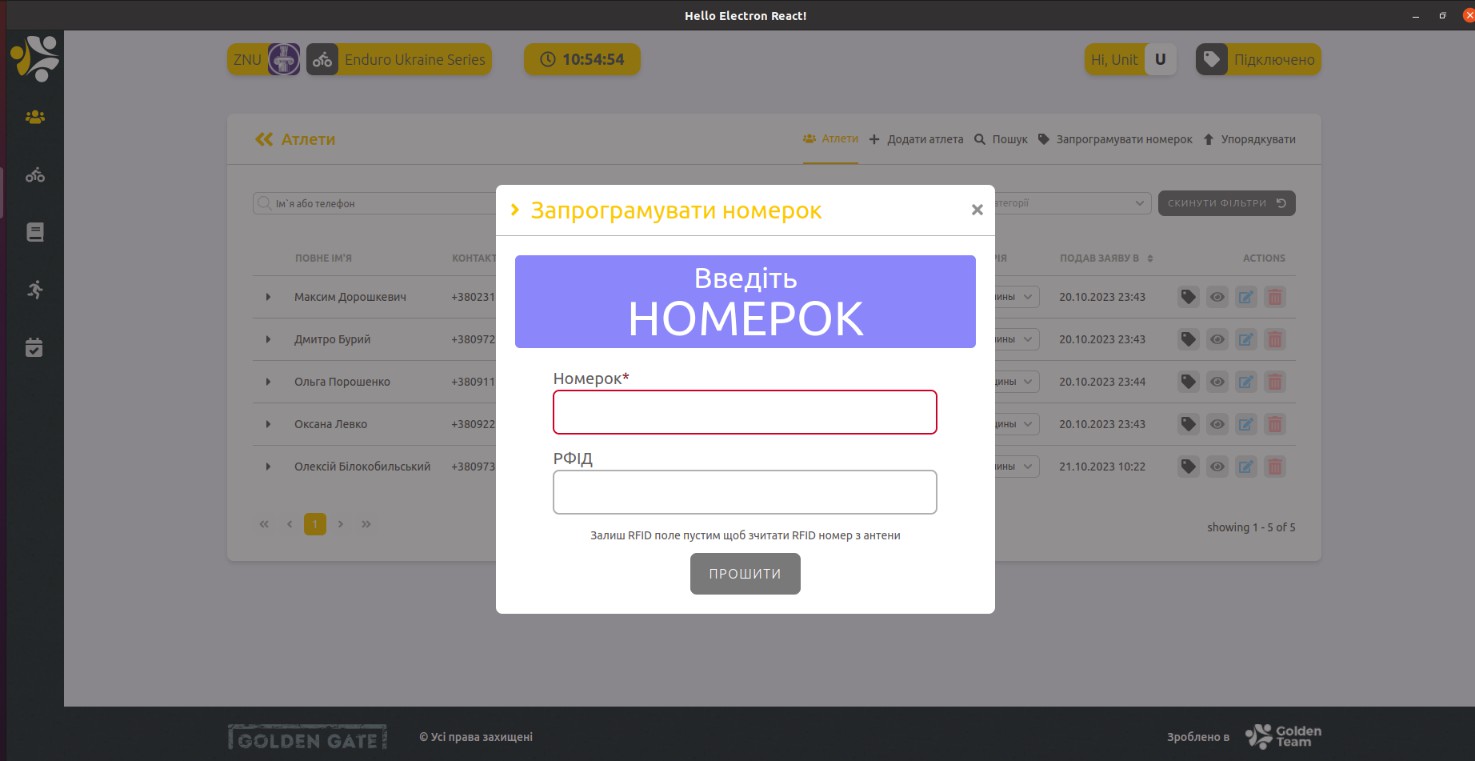


Рисунок 12 — Модальне вікно для присвоєння номерка атлету

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 но- мерка з мітки цією антенною. Перевірка обладнання заключається в тому, що існує 5 індикаторів, які по замовчуванню відображені червоним кольором. На- тиснувши 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 воро- тами старта, їхня кількість залежить від кількості стейджів у даному змаганні.

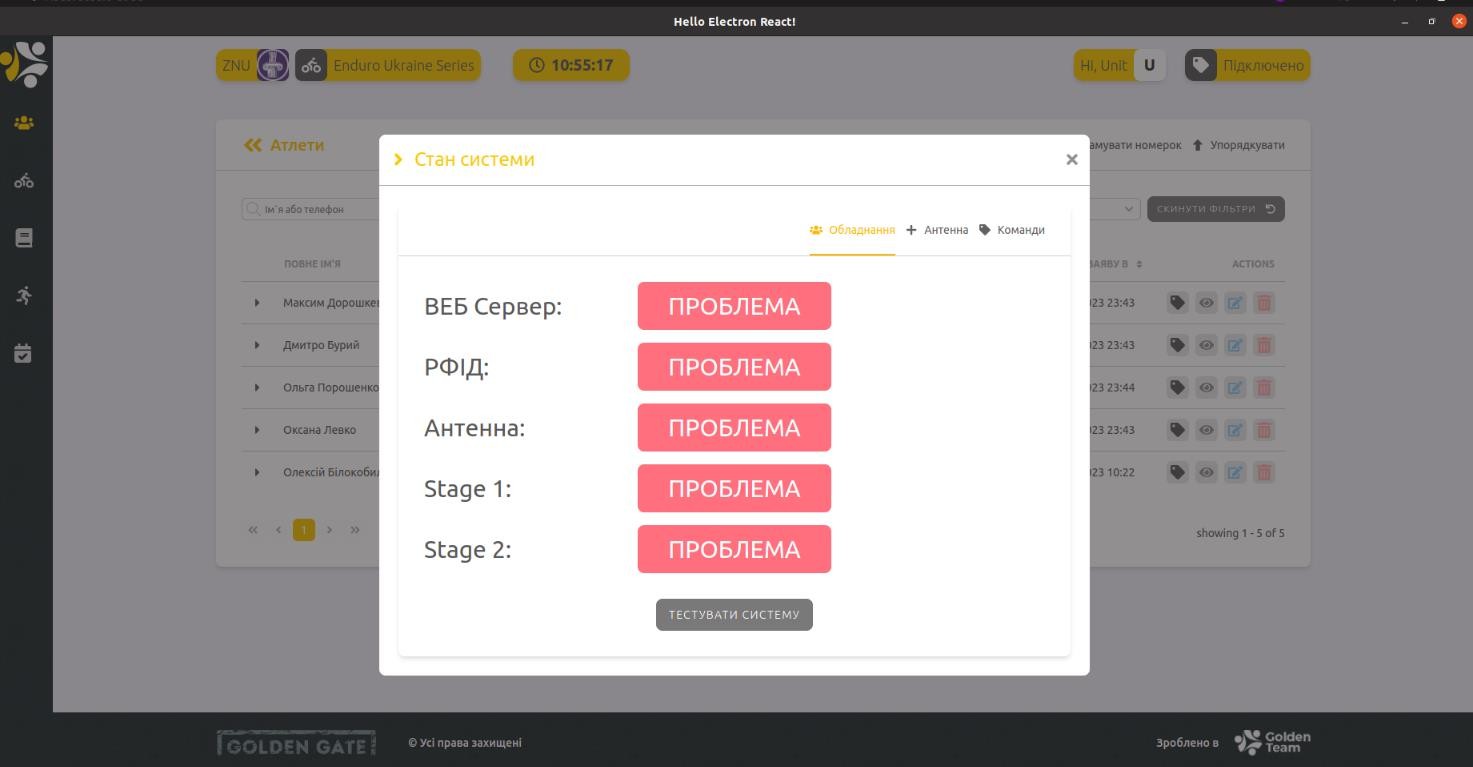


Рисунок 13 — Модальне вікно перевірки системи

## Розробка програмного забезпечення

## Архітектура системи

Архітектура даної системи має розподілений характер і складається з кіль- кох ключових компонентів: веб-серверу, веб-сайту, мобільного додатку, що спі- лкується з Arduino та десктопного додатку, що спілкується з Impinj.(див. рис. 14) Сервер обмінюється даними з клієнтами за допомогою HTTP запитів, а для мо- ментального оновлення оперативної інформації використовуються вебсокети[7]. Для зв’язку між мобільним додатком і Arduino (воротами старту) використову- ється 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.

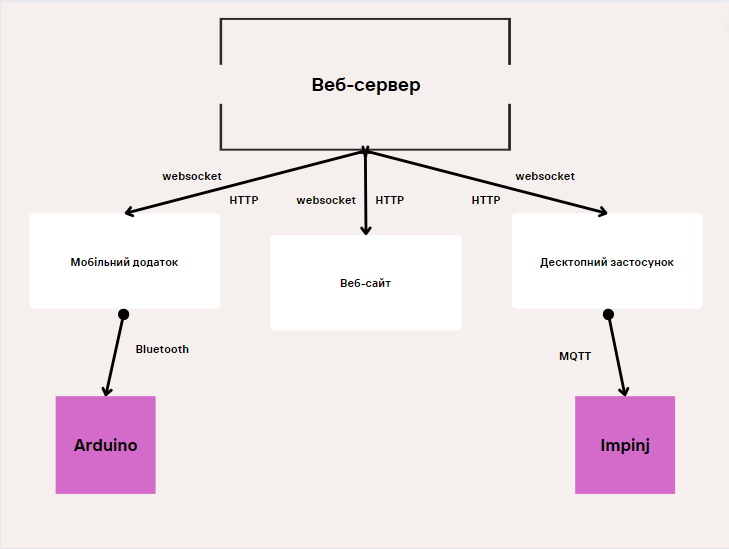


Рисунок 14 — Архітектура системи

## Розробка серверної частини

Основним моментом у створенні системи було зробити її зручною для ма- сштабування 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 взає- модії із клієнтським додатком, який використовує Electron. При створенні екзе- мпляра класу вказується головне вікно BrowserWindow, і відбувається ініціаліза- ція 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 виклика- ється 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 голов- ного процесу Electron. Клас також надає методи start, stop та isListening для руч- ного 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 реа- лізовує 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 кліє- нтським додатком на Electron. Клас успадковує методи для обробки з'єд- нань, запуску та зупинки сервера. Також він містить додатковий метод

`checkIfConnected`, який обіцяє повернути булеве значення та може вико- ристовуватися 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 до- помогою IPC-повідомлень.

Лістинг 2 — Реалізація класу-спадкоємця для спілкування за допо- могою 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 викорис- товує 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 ме- тод 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 про- цесу Electron. Метод stop відписується від теми та при необхідності відключа- ється від брокера. Метод checkIfConnected перевіряє, чи зберігається з'єднання з брокером, відправляючи тестове повідомлення. Загалом, клас MQTTServer реа- лізує інтеграцію з MQTT-брокером для отримання та обробки даних RFID в кон- тексті 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');

}

});}}

## Вебсокети

Всі клієнти спілкуються з сервером за допомогою вебсокетів. Лістинг 4 де- сонструє підключення вебсокетів на стороні сервера. Цей код реалізує клас SocketServer, що діє як WebSocket-сервер для забезпечення двостороннього зв'я- зку між клієнтами та сервером. У конструкторі виконується ініціалізація сервер- них властивостей, а метод start() запускає HTTP- та WebSocket-сервери, налаш- товані 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 за- критті з'єднання викликається onCloseConnection(), який встановлює таймер для затримки видалення з'єднання. Крім того, є метод onAccept(), що реагує на отри- мання сигналу 'accept' від клієнта. Метод send() відправляє повідомлення всім підключеним клієнтам. Усі дії спрямовані на підтримку безперервної та ефекти- вної 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) );

}

}

}

## Висновки до розділу 3

1. Було налаштоване середовище для розробки програмного забезпе- чення за допомогою ReactJs, NodeJs, NextJs, React Native та середи розробки Visual Studio Code.
2. Було розроблено інтерфейс та створено відповідні компоненти для ре- алізації однотипного UI на всіх клієнтах.
3. Виконано задачу створення коректної архітектури для розподілених додатків. Окремо створено архітектуру веб-сайту, десктопного застосунку і мо- більного застосунку.
4. Інтегровано необхідні бібліотеки для спілкування застосунків з Ar- duino, Impinj, Alien.
5. Розроблено програму для кожного з застосунків.
6. Інтегровано функціонал спілкування між сервером та клієнтами за до- помогою вебсокетів за допомогою бібліотеки Socket.io.
7. На етапі оптимізації було створено базовий клас netServer для масшта- бування 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 мо- більного застосунку.

# РОЗДІЛ 4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ РОБОТИ КОМП’ЮТЕРНОЇ СИСТЕМИ РОЗПОДІЛЕНИХ ДОДАТКІВ

## 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 сере- довищі для того, щоб точно визначити чи правильно працюють зчитування мі- ток, чи моментально оновлюються усі дані за допомогою вебсокетів і ,зви- чайно, знайти можливі помилки або проблеми, що неможливо визначити у ла- бораторних умовах.

Тож було створено новий захід, зареєстровано учасників та присвоєно ко- жному 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 викорис- товувати 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 вер- шині гори. Ворота фінішу встановлено внизу гори (див. рис. 16). Веб-сайт відк- ритий на ноутбуці адміністратора, що знаходиться у палатці адміністрації. Зма- гання 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 насту- пним не прийшов, бо запізнюється чи щось сталося на старті і потрібно поста- вити 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 ві- дображення таймеру і пересування черги на відповідну кількість часу. При пер- ших ітераціях такі дії призводили до руйнування системи, бо відбувався конф- лікт. Тоді на ітерації оптимізації було змінено процес паузи і на змагання тест пройшов успішно. Стейджі було відновлено, час пересунувся коректно і зма- гання було продовжено.

Далі переходимо до процесу фінішу. Було протестовано швидкість реакції системи на перетинання фінішу на великій швидкості (див. рис. 17). При перети- нанні атлетом воріт фінішу RFID зчитувач забирає інформацію з мітки та відп- равляє 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 зма- гань було проведено тестовий заїзд, де першими їхали ті, хто має низьку швид- кість. Тоді система Alien відпрацювала відмінно, дані передалися на сервер і від- правилися клієнтам. Можна було робити висновок, що даний зчитувач підходить для системи. Він дешевий, швидко програмується виконує свої функції. Але по- тім 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 дета- льно видно наскільки правильним було рішення створити базовий клас для до- давання різних зчитувачів, бо тепер Impinj можна використовувати для високо- швидкісних 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 ін- терфейсі. Це пришвидшить розробку та робить систему надійною. Також з отри- маних 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 різ- них 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.

## Висновки до розділу 4

1. Використання Impinj для зчитування міток є більш оптимальним варі- антом, 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 таб- лиць 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 га- рно володіти системою.
4. Змагання пройшли успішно, тож система є робочою і готовою до ви- користання. 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 ти- пами RFID зчитувачів Impinj та Alien, створено функціонал роботи з Arduino. Система є повністю готовою для використання на будь-яких операційних систе- мах.
3. Досліджені результати роботи системи на реальних змаганнях з велоси- педного спорту. Проаналізовано роботу Impinj та Alien: перший є дорожчим і складнішим у програмуванні, але зчитує дані з міток на великій швидкості; дру- гий є дешевший і простіший, але не працює правильно при великій швидкості.
4. Визначено, що використання вебсокетів у розподілених додатках є ідеа- льним рішення для моментального оновлення інформації. Завдяки швидкості ро- боти ReactJs інформація на сторінках оновлюється дуже швидко.
5. Поставлена проблема використання ReactJs та NodeJs як єдиної платфо- рми 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 Me- dia, 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 JavaS- cript. 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 за- стосування 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 вико- ристання ReactJs і NodeJs для створення розподілених додатків. «Актуальні пи- тання сталого науково-технічного та соціально-економічного розвитку регіонів

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

1. Franks M., Dely M., Meyarivan T. Professional WebSocket Programming: Developing Real-Time Applications with HTML5 and WebSockets. Wiley, 2013.
2. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
3. Hohpe G., Woolf B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2003.
4. WebSockets - A Conceptual Deep Dive. MDN Web Docs. URL: https://de- veloper.mozilla.org/en-US/docs/Web/API/WebSockets\_API (дата звернення: 10.06.2023).
5. Newman S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2015. С. 120–125.
6. Freeman E., Robson D. Head First Design Patterns. O'Reilly Media, 2004. С.110-112.
7. Flanagan D. JavaScript: The Definitive Guide. O'Reilly Media, 2020. P. 52–

63.

1. Hughes B. Pro Node.js for Developers. Apress, 2013. С. 120–125.
2. Resig J., Bibeault B., Maras B. Secrets of the JavaScript Ninja. Manning

Publications, 2016. P. 52–63.

1. Sommerville I. Software Engineering. Addison-Wesley, 2011. С. 120–125.
2. Hunter, Douglas, and Lima, Juliana. "Fullstack Open: The Definitive Guide for Modern JavaScript Web Development." URL: https://fullstackopen.com/ (дата звернення: 12.03.2023).
3. Документація ReactJS URL: <https://react.dev/> (дата звернення : 10.10.2022).
4. Документація NodeJS URL: https://nodejs.org/en (дата звернення : 09.11.2022).
5. Документація Redux URL: https://redux.js.org/ (дата звернення : 22.12.2022).
6. Adolph, Scott, et al. "Building Evolutionary Architectures: Support Constant Change." O'Reilly Media, 2017.
7. Brown, Kyle. "Node.js Design Patterns." Packt Publishing, 2014.
8. Goncalves, Arun. "WebSocket: Lightweight Client-Server Communica- tions." O'Reilly Media, 2015.
9. "Architectural Styles and Patterns - Tutorialspoint." URL: https://[www.tutorialspoint.com/software\_architecture\_design/architectural\_styles\_an](http://www.tutorialspoint.com/software_architecture_design/architectural_styles_an) d\_patterns.htm (Дата звернення: 25.11.2022)
10. "RESTful API Designing guidelines — The best practices." URL: https://hackernoon.com/restful-api-designing-guidelines-the-best-practices- 60e1d954e7c9 (Дата звернення: 03.02.2023)
11. "WebSocket API - Web APIs | MDN." URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket (Дата звернення: 03.02.2023)
12. "Microservices Architecture: What Is, Benefits & Examples." URL: https://[www.microfocus.com/documentation/silk-performer/205/en/silkperformer-](http://www.microfocus.com/documentation/silk-performer/205/en/silkperformer-) 205-webhelp-en/GUID-C09A7DCD-666D-4E7D-AE9D-C147FD9F8A72.html (Дата

звернення: 13.04.2023).

1. "React Native Documentation." URL: https://reactnative.dev/docs/getting- started (Дата звернення: 22.12.2022).
2. "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).

**ДОДАТОК А**

**Код 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

}

}

**Код 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);

}

}

**Код 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');

}

});

}

**Код 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)

}

}

}

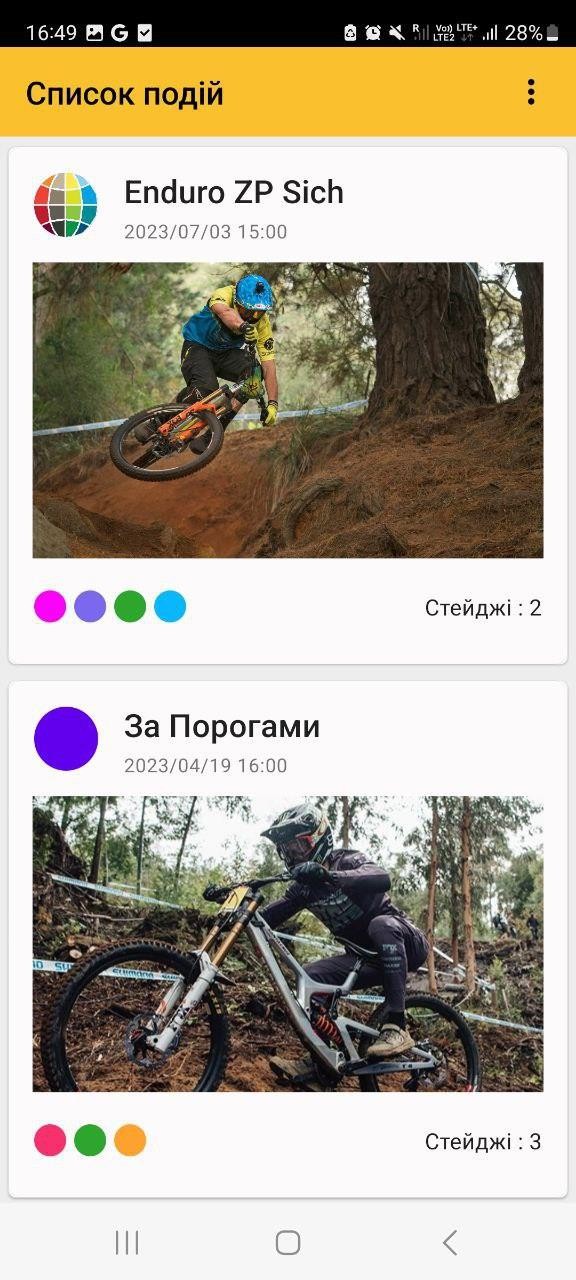
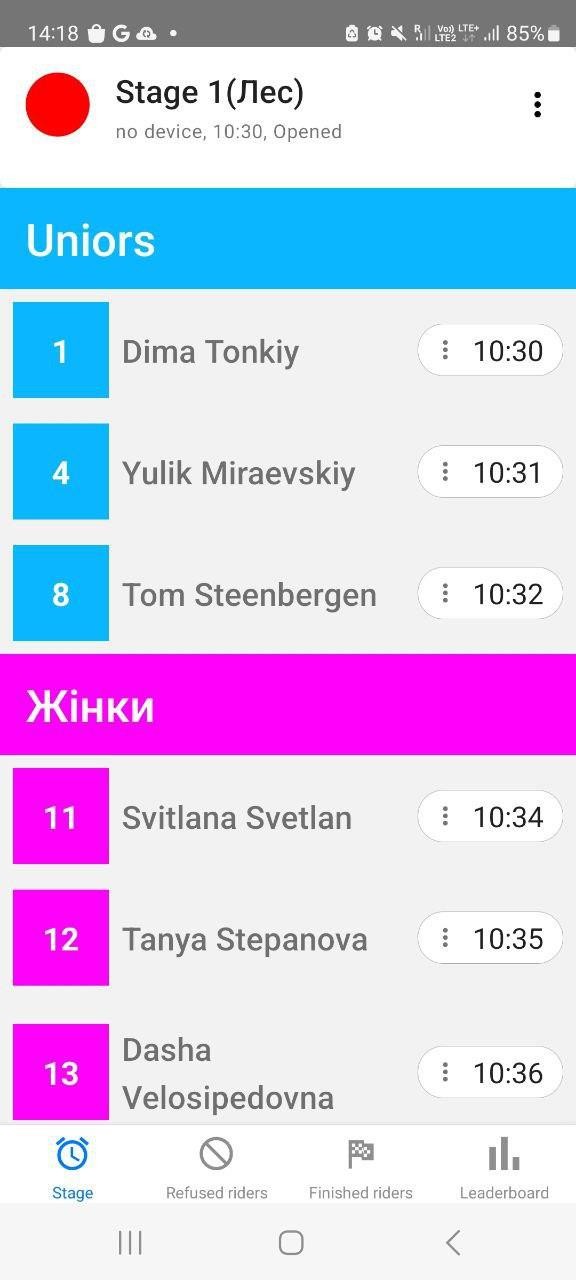
public send(entityName: string, msg: any) { if (!isEmpty(this.connects)) {

Object.keys(this.connects).map((key) => this.sendToKey(entityName, msg, key) );

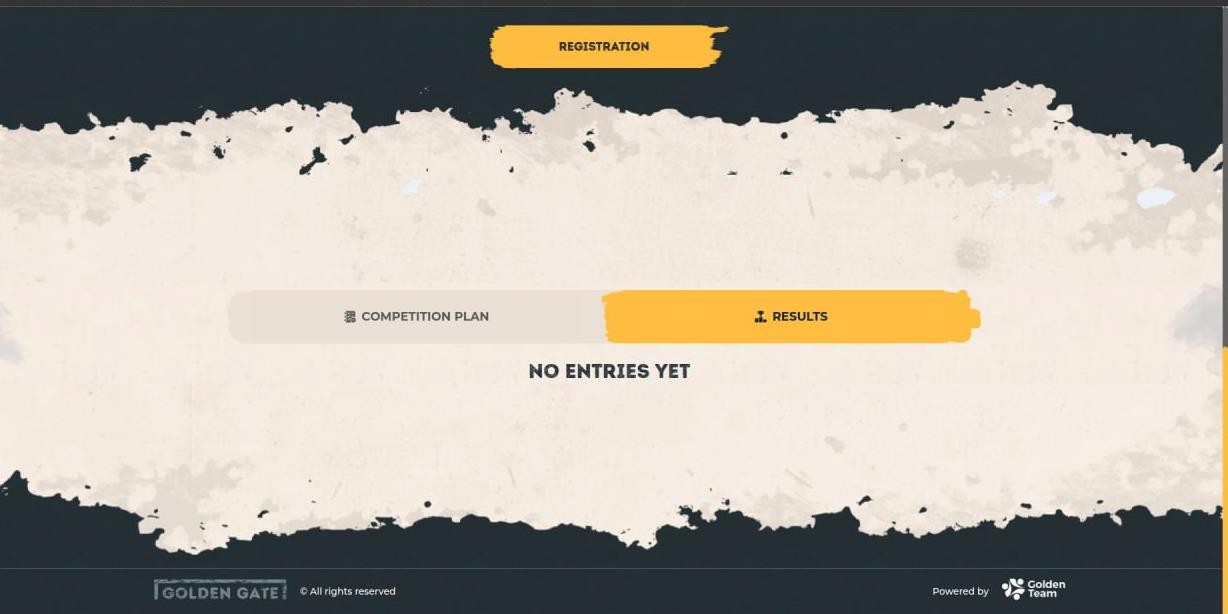
}

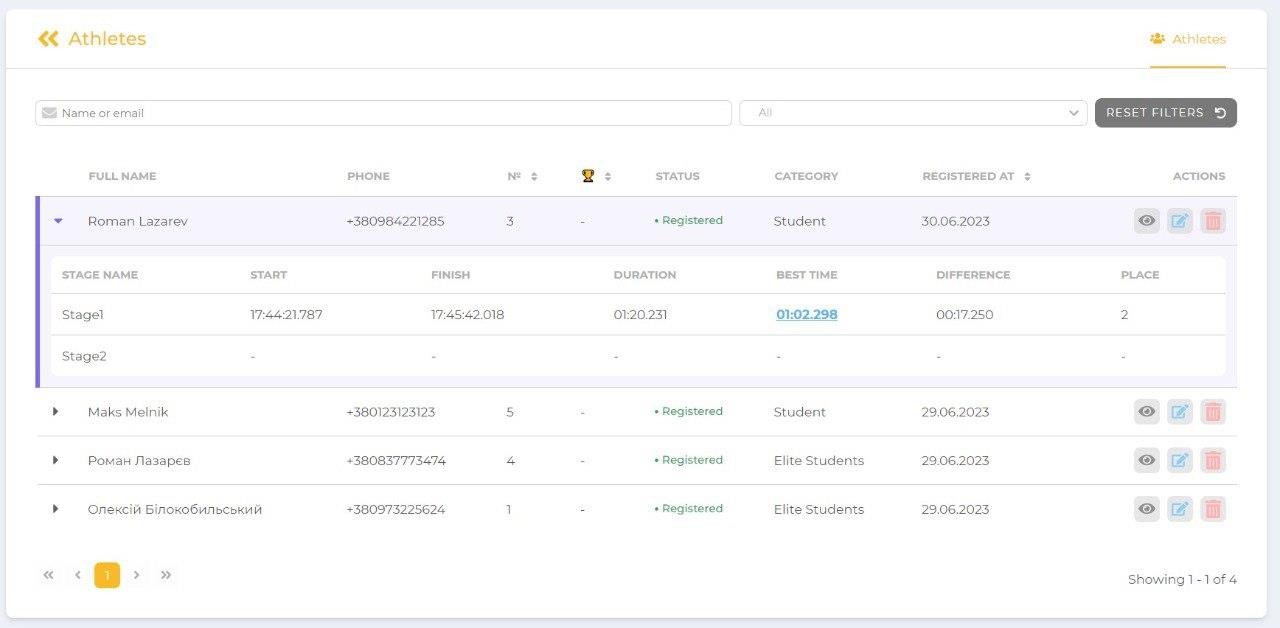
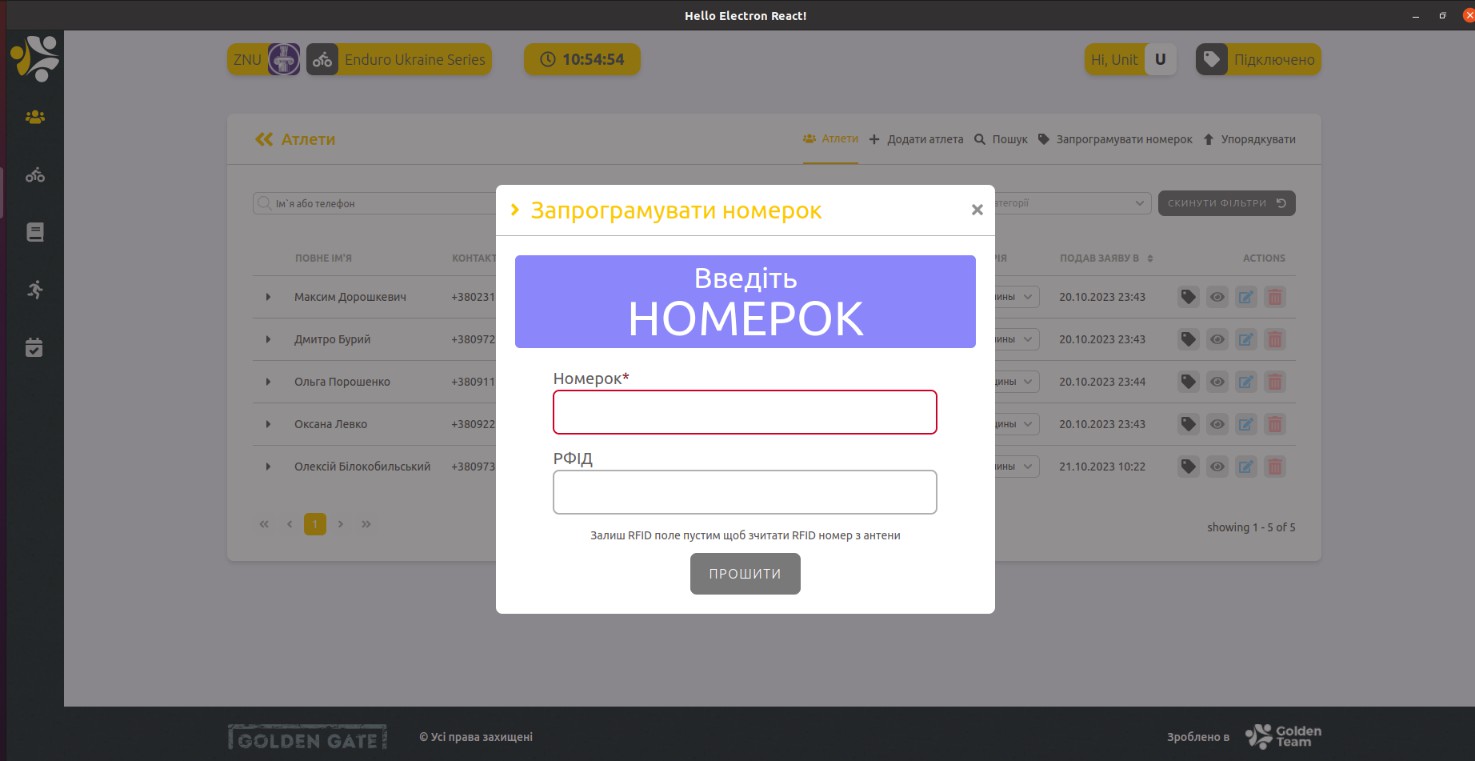
}

**ДОДАТОК Б  
  
ІНТЕРФЕЙС МОБІЛЬНОГО ТА ДЕСКТОПНОГО ЗАСТОСУНКУ** Головна сторінка мобільного застосунку

  
  
Сторінка Стейджу та черги мобільного застосунку  
Головна сторінка сайту  
  


Сторінка результатів



Таблиця атлетівМодальне вікно для присвоєння номерка атлету  
  


Модальне вікно перевірки системи