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

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

на тему: ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ТРЕНУВАННЯ НАВИЧОК ШВИДКІСНОГО ДРУКУ НА КЛАВІАТУРІ

Проєктував _____ Д. М. Бабенко

Керівник роботи _____ І. О. Музика

Нормоконтроль _____ Д. І. Кузнєцов

Завідувач кафедри _____ А. І. Купін

Кривий Ріг
2025

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

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

ЗАТВЕРДЖУЮ
Завідувач кафедри, голова циклової комісії

_____ А. І. Купін

"____" _____ 20__ року

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

_____Бабенко Дмитро Максимович_____
(прізвище, ім'я, по батькові)

1. Тема роботи _____програмне забезпечення для тренування_____ навичок швидкісного друку на клавіатурі_____
_____
керівник роботи __Музика Іван Олегович, кандидат технічних наук, доцент_,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "___"_____ 20__року №__
2. Строк подання студентом роботи _____
3. Вихідні дані до роботи __розроблене програмне забезпечення для_____ тренування навичок швидкісного друку на клавіатурі_____
_____
_____
_____
_____
_____
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)_аналіз інструментів тренування, функціональні вимоги, технічні вимоги, архітектура веб-сервісу, вибір технологій, база даних, серверна частина, клієнтська частина, змагання в реальному часі, генерація текстових фрагментів, тестування функціональності розробленого проєкту__
_____
_____
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)_____
_____
_____
_____
_____
_____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Музика І. О., доцент кафедри КСМ		

7. Дата видачі завдання 28.01.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк виконання етапів роботи	Примітка
1	Отримання завдання	28.01	
2	Написання вступу	25.04	
3	Збір теоретичного матеріалу	15.03 – 30.03	
4	Написання І розділу	18.04 – 24.04	
5	Редагування І розділу	25.04	
6	Написання ІІ розділу	26.04 – 13.05	
7	Редагування ІІ розділу	14.05	
8	Написання ІІІ розділу	15.05 – 20.05	
9	Редагування ІІІ розділу	21.05 – 28.05	
10	Написання IV розділу	29.05	
11	Редагування IV розділу	30.05 – 01.06	
12	Написання висновків	01.06	
13	Редагування проєкту	02.06	

Студент _____ _____
(підпис) (прізвище та ініціали)

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

РЕФЕРАТ

Пояснювальна записка: 77 сторінок, 31 рисунок, 2 таблиці, 19 використаних джерел, 2 додатка.

Мета проєктування – проктування та розробка інтерактивного веб-сервісу «KeyRaces» для тренування навичок швидкого друку та проведення змагань, з використанням технологій .NET Core, Blazor Server, SignalR, Entity Framework Core, PostgreSQL та інтеграцією з локальною мовною моделлю Ollama.

Проєкт складається з вступу, чотирьох основних розділів та висновків.

Перший розділ присвячений аналізу предметної області, постановці задачі та вибору технологічного стеку. Описано актуальність проблеми вдосконалення навичок швидкого друку, проведено огляд існуючих рішень та обґрунтовано вибір платформи .NET, мови C#, фреймворку Blazor Server для реалізації користувацького інтерфейсу, а також PostgreSQL як системи управління базами даних та Ollama для генерації текстового контенту.

Другий розділ охоплює детальне проєктування архітектури системи. Описано багатошарову архітектуру додатку (Core, Infrastructure, Server), розроблено схему бази даних з використанням Entity Framework Core, визначено основні сутності, сервіси та їх взаємодію. Розглянуто принципи автентифікації та авторизації користувачів за допомогою ASP.NET Core Identity.

У третьому розділі детально описано процес реалізації ключових функціональних модулів веб-сервісу та підготовку до розгортання. Розглянуто розробку API ендпоінтів для управління даними, впровадження технології SignalR для забезпечення взаємодії в реальному часі під час змагань та в чаті лобі, інтеграцію з мовною моделлю Ollama для динамічної генерації текстів, створення компонентів користувацького інтерфейсу на Blazor та адміністративної панелі. Окремо висвітлено питання налаштування та розгортання додатку за допомогою Docker.

Четвертий розділ присвячений демонстрації та перевірці функціональних можливостей розробленого веб-сервісу KeyRaces. Наведено огляд основного інтерфейсу та навігації, продемонстровано процеси створення та управління обліковим записом, випробувано функціонал індивідуального тренування та режиму змагань у реальному часі.

ВЕБ-СЕРВІС, ШВИДКИЙ ДРУК, BLazor SERVER, .NET CORE, SIGNALR, ENTITY FRAMEWORK CORE, POSTGRESQL, OLLAMA, DOCKER, АВТЕНТИФІКАЦІЯ, API, МОДУЛЬНЕ ТЕСТУВАННЯ, АДМІНІСТРАТИВНА ПАНЕЛЬ, РЕАЛЬНИЙ ЧАС, ДЕМОНСТРАЦІЯ ФУНКЦІОНАЛУ.

Explanatory note: 77 pages, 31 figures, 2 tables, 19 references, 2 appendices.

The purpose of the project is to design and develop an interactive web service “KeyRaces” for training speed typing skills and competitions, using .NET Core, Blazor Server, SignalR, Entity Framework Core, PostgreSQL technologies and integration with the local language model Ollama.

The project consists of an introduction, four main chapters and conclusions.

The first section is devoted to the analysis of the subject area, problem statement, and selection of the technology stack. It describes the relevance of the problem of improving speed typing skills, reviews existing solutions, and justifies the choice of the .NET platform, C# language, Blazor Server framework for implementing the user interface, as well as PostgreSQL as a database management system and Ollama for generating text content.

The second section covers the detailed design of the system architecture. It describes the multilayer architecture of the application (Core, Infrastructure, Server), develops a database schema using the Entity Framework Core, and defines the main entities, services, and their interaction. The principles of user authentication and authorization using ASP.NET Core Identity are considered.

The third section describes in detail the process of implementing the key functional modules of the web service and preparing for deployment. It covers the development of endpoint APIs for data management, the implementation of SignalR technology to provide real-time interaction during competitions and in the lobby chat, integration with the Ollama language model for dynamic text generation, creation of user interface components on Blazor and the administrative panel. The chapter also covers how to set up and deploy an application using Docker.

The fourth chapter is devoted to demonstrating and testing the functionality of the developed KeyRaces web service. It provides an overview of the main interface and navigation, demonstrates the processes of creating and managing an account, and tests the functionality of individual training and real-time competition mode.

WEB SERVICE, QUICK PRINT, BLAZOR SERVER, .NET CORE, SIGNALR, ENTITY FRAMEWORK CORE, POSTGRESQL, OLLAMA, DOCKER, AUTHENTICATION, API, MODULE TESTING, ADMINISTRATIVE PANEL, REAL TIME, FUNCTIONAL DEMONSTRATION.

ПЕРЕЛІК СКОРОЧЕНЬ

БД – База Даних;
СУБД – Система Керування Базами Даних;
API – Application Programming Interface (укр. інтерфейс програмного додатку);
CPM – Characters Per Minute (укр. символи за хвилину);
CRUD – Create Read Update Delete;
CSRF – Cross-Site Request Forgery (укр. міжсайтова підробка запиту);
HTTP – HyperText Transfer Protocol (укр. протокол передачі гіпертексту);
HTTPS – HyperText Transfer Protocol Secure;
JSON – JavaScript Object Notation (укр. запис об’єктів JavaScript);
JWT – JSON Web Token;
LLM – Large Language Model (укр. велика мовна модель);
WPM – Words Per Minute (укр. слова за хвилину);
XSS – Cross-Site Scripting (укр. міжсайтовий скриптинг);

ЗМІСТ

ВСТУП

У сучасному інформаційному суспільстві вміння швидко й точно вводити текст на клавіатурі є одним із ключових елементів цифрової грамотності. Зростання обсягів текстової інформації, необхідність оперативного обміну повідомленнями та масове застосування комп'ютерних систем у професійній і освітній сферах вимагають від користувачів високої швидкості набору та мінімізації кількості помилок. Ручне опанування навичок друку без спеціалізованих засобів часто виявляється тривалим і малоефективним, що знижує продуктивність праці та мотивацію до вдосконалення.

Аналіз існуючих програмних рішень показує, що традиційні тренажери друку здебільшого побудовані за статичною моделлю: вони пропонують набори текстів фіксованої складності й недостатньо враховують індивідуальні особливості користувача. Для досягнення високих результатів навчання необхідний адаптивний підхід, який автоматично коригує вправи залежно від поточних показників швидкості та точності введення. Розроблено інтерактивну веб-платформу для тренування навичок швидкісного друку на клавіатурі, яка включає модулі тренувань різного рівня складності, онлайн-змагання в реальному часі та аналітичний компонент для детального аналізу продуктивності користувача .

Архітектура системи передбачає клієнт-серверну взаємодію з використанням ASP.NET Core та механізму SignalR для організації двосторонньої комунікації в онлайн-змаганнях. Дані користувацької статистики кешуються в Redis для прискорення відображення графіків та зниження навантаження на базу даних. Контейнеризація за допомогою Docker забезпечує гнучке розгортання й масштабування сервісу, а реалізація модульного тестування гарантує високу якість та надійність програмного забезпечення [11, 13].

Метою кваліфікаційної роботи є розробка програмного забезпечення для тренування навичок швидкісного друку, що реалізує адаптивну систему вправ, конкурси в реальному часі та збір статистики продуктивності. Для досягнення поставленої мети необхідно вирішити такі завдання:

1. Провести аналіз вимог і огляд існуючих рішень із тренування клавіатурного набору.
2. Розробити загальну архітектуру системи та деталізовані схеми взаємодії компонентів.
3. Реалізувати клієнтську частину з інтерфейсом для тренувань, змагань та відображення статистики.

4. Забезпечити серверну обробку з використанням SignalR, Redis та бази даних.
5. Провести тестування продуктивності та надійності, оцінити результати в умовах реального навантаження.

1. АНАЛІЗ ІНСТРУМЕНТІВ ТРЕНУВАННЯ

1.1 Існуючі веб- и декстоп- тренажери для швидкісного набору

У цьому підрозділі наведемо детальний технічний опис найбільш відомих веб- і десктоп-тренажерів, опираючись на конкретні метрики й технології.

Typing

Аудиторія: понад 50 млн зареєстрованих користувачів.

Метрики: середня швидкість 45 WPM, точність ~92 %.

Функціональність: адаптивний вибір вправ (рівні складності від «Новачок» до «Експерт»), інтегровані уроки з підказками, історія результатів із графіками.

Технології: фронтенд на React, бекенд на Node.js, WebSocket для реального часу [1].

10FastFingers

Аудиторія: понад 20 млн тестів на місяць.

Метрики: середня швидкість 50 WPM, точність ~90 %.

Функціональність: тести на швидкість (60 секунд), підтримка 30 мов, щоденні рейтинги.

Технології: чистий HTML5/JS, PHP/MySQL, відсутність адаптивності — усі користувачі проходять однакові тексти [2].

Keybr

Аудиторія: ~5 млн користувачів.

Метрики: середня швидкість 48 WPM, точність ~94 %.

Функціональність: алгоритм розподілу символів із фокусом на часто помилкові, персоналізовані «хмаринки» для ICU (individual character units).

Технології: Python/Flask, SQLite, WebSocket [3].

TypeRacer

Аудиторія: понад 10 млн активних гравців.

Метрики: середня швидкість 60 WPM, точність ~88 %.

Функціональність: мультиплеєрні гонки в реальному часі до 8 гравців, рейтинги, текстові бібліотеки.

Технології: ASP.NET, SignalR, SQL Server. [4]

TypingMaster (десктоп)

Аудиторія: комерційний продукт, близько 1 млн ліцензій.

Метрики: середня швидкість 55 WPM, точність ~93 %.

Функціональність: офлайн-рівні, вбудований аналізатор слабких клавіш, підтримка багатьох мов, персональні тренування.

Технології: .NET Framework, локальна база даних [5].

Klavaro (десктоп)

Аудиторія: відкрите ПЗ, ~200 тис. завантажень.

Метрики: середня швидкість 50 WPM, точність ~95 %.

Функціональність: модульні вправи (позиції рук, комбінації клавіш), локалізація на 15 мов.

Технології: C++/Qt, XML-налаштування тренувань [6].

На графіку (рис. 1.1) показано середню швидкість (WPM) для кожної платформи. Як бачимо, TypeRacer та TypingMaster мають найвищі середні показники швидкості, проте їхні рішення поступаються в точності та адаптивності (TypeRacer – 88 % точності, TypingMaster – відсутність онлайн-змагань).

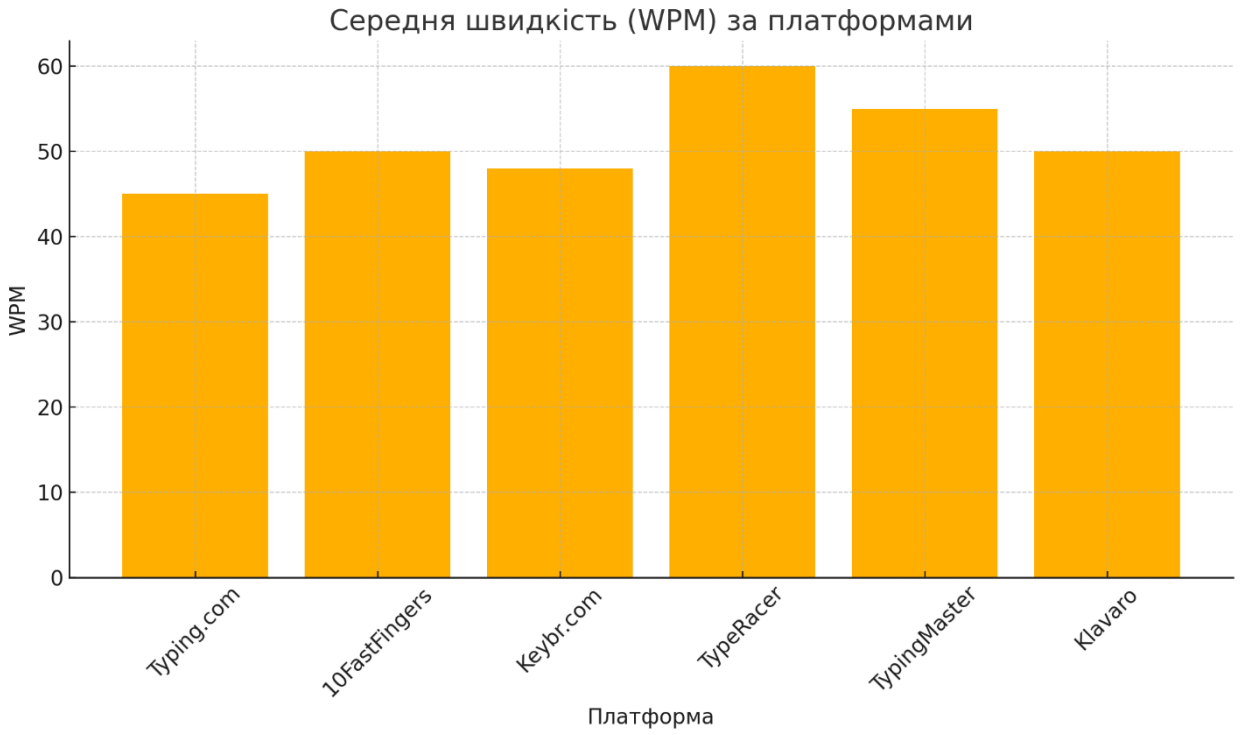

Рисунок 1.1 – Графік порівняння середньої швидкості набору тексту на різних платформах

Незважаючи на значну популярність та доступність онлайн-тренажерів, вони часто страждають від надмірної одноманітності вправ — тексти повторюються у різних комбінаціях, але рідко враховують індивідуальні слабкі місця користувача. Більшість платформ побудовано за моделлю «один розмір — для всіх», через що неможливо глибоко опрацювати характерні для кожного помилки або адаптувати складність завдань під темп навчання.

Часте використання однакових наборів слів і фраз спричиняє звикання, через що знижується мотивація та ефективність тренувань.

Десктопні рішення, хоч і пропонують гнучкі налаштування курсів та локальні словники, мають суттєві недоліки у виробничому середовищі. Обов'язкова інсталяція та ліцензійні обмеження TypingMaster можуть відлякати користувачів без можливості встановлювати ПЗ або проводити оновлення. Klavaro, хоча й відкрите ПЗ, має застарілий інтерфейс і обмежену підтримку мов, що ускладнює його використання на багатомовних проєктах.

1.2 Аналіз методик оцінювання швидкості та точності набору тексту

Основними метриками оцінювання швидкості та точності набору тексту є:

1. WPM – кількість слів, набраних за хвилину. Стандартне слово в цій метриці складається з 5 символів, включаючи пробіли та знаки пунктуації. Формула розрахунку:

```
WPM = (загальна кількість символів / 5) / (час у хвилинах)
```

2. CPM – кількість символів, набраних за хвилину:

```
CPM = загальна кількість символів / час у хвилинах
```

3. Точність — відсоток правильно набраних символів:

```
Точність = (1 - (кількість помилок / загальна кількість символів)) *
                                    100%
```

4. Коефіцієнт ефективності — комбінована метрика, що враховує як швидкість, так і точність:

```
Ефективність = WPM * Точність / 100
```

Порівняльний аналіз методик оцінювання в різних системах наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз методик оцінювання

Система	Основна метрика	Особливості розрахунку	Додаткові показники
TypeRacer	WPM	Враховує лише завершені слова	Рейтинг Elo [7] для змагань
Keybr	WPM + точність	Адаптивне зважування символів	Теплова карта клавіатури
10FastFingers	WPM	Фіксований час (60 секунд)	Порівняння з середніми показниками
Розроблювана система	WPM + точність	Динамічний розрахунок з урахуванням складності тексту	Історичні тренди, аналіз помилок

Для нашої системи обрано комбінований підхід, що дозволяє не лише вимірювати швидкість, але й аналізувати типові помилки користувача. Алгоритм розрахунку включає:

1. Фіксацію часу початку та завершення сесії
2. Підрахунок загальної кількості символів
3. Виявлення та класифікацію помилок (пропуски, заміни, перестановки)
4. Розрахунок WPM та точності
5. Збереження даних для подальшого аналізу

Такий підхід забезпечує більш точну оцінку прогресу користувача та дозволяє формувати персоналізовані рекомендації для покращення навичок.

1.3 Аналіз психофізіологічних аспектів швидкісного набору тексту

Швидкість набору тексту залежить від кількох ключових факторів, які можна кількісно виміряти та оптимізувати:

1. Моторна пам'ять – автоматизація рухів пальців, що вимірюється через: час реакції на символ (мс), стабільність інтервалів між натисканнями клавіш, кількість помилок при прискоренні

2. Когнітивне навантаження – здатність обробляти текст під час набору: буфер випередження (кількість символів, які користувач запам'ятовує наперед), час затримки між читанням та набором, вплив складності тексту на швидкість

Згідно з дослідженням Фейс та Логана (2018) [8], яке вивчало когнітивні та моторні аспекти набору тексту серед 1000+ учасників різного рівня досвіду, було виявлено чіткі закономірності у показниках швидкості та точності. Результати цього дослідження наведено в таблиці 1.2.

Таблиця 1.2 – Показники швидкості набору в залежності від досвіду

Рівень досвіду	Середня швидкість (WPM)	Міжклавішний інтервал (мс)	Бувер випередження (символів)	Точність (%)
Початківець	15-25	480-720	1-2	78-85
Середній	35-55	240-360	2-4	92-96
Досвідчений	65-85	120-180	5-7	96-98
Експерт	100+	80-110	8-14	98-99.5

Дослідження показало, що експерти з швидкістю понад 100 WPM демонструють значно коротші міжклавішні інтервали (80-110 мс) порівняно з початківцями (480-720 мс). Крім того, експерти здатні утримувати в робочій пам'яті значно більший буфер випередження — до 14 символів, тоді як початківці обробляють лише 1-2 символи наперед.

Аналіз електроміографічних (ЕМГ) даних, проведений Салтхаусом (2020) [9], показує, що при швидкості набору понад 60 WPM спостерігається значне зниження м'язової напруги порівняно з повільнішим набором, що свідчить про автоматизацію рухів (рис. 1.2).

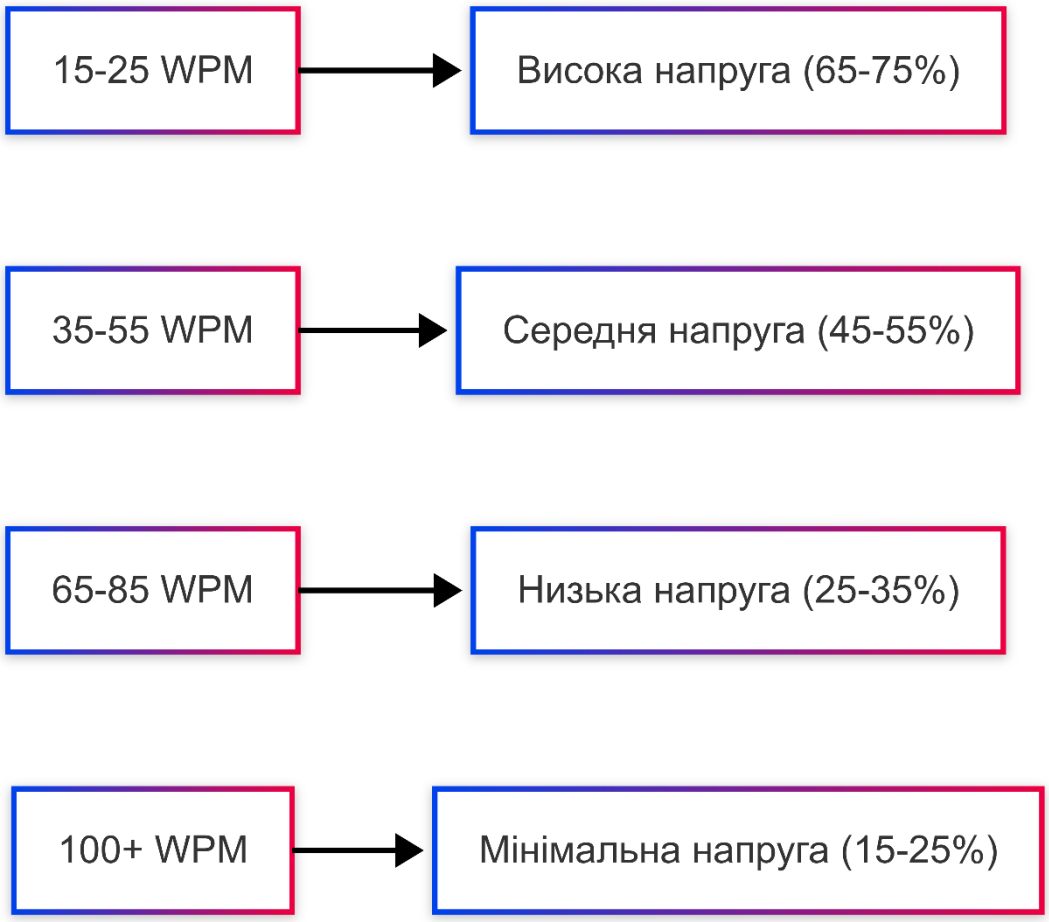

Рисунок 1.2 – Аналіз електроміографічних даних

Дослідження Яманаші та співавторів (2022) [10] щодо розподілу помилок за типами показує, що найчастіше зустрічаються:

1. Заміни символів (48.2%) — натискання неправильної клавіші
2. Пропуски символів (27.5%) — ненатискання потрібної клавіші
3. Перестановки (14.8%) — зміна порядку символів
4. Вставки (9.5%) — додавання зайвих символів

Ці дані є критично важливими для розробки адаптивних алгоритмів навчання, які повинні фокусуватися на найбільш проблемних аспектах для конкретного користувача.

1.4 Аналіз вимог до сучасних систем тренування швидкісного друку

Сучасні системи тренування швидкісного друку повинні відповідати комплексу технічних, функціональних та ергономічних вимог, що забезпечують ефективне навчання та підтримку користувачів різного рівня підготовки. На основі аналізу технічної документації, міжнародних стандартів та існуючих програмних рішень можна сформулювати систематизований перелік вимог до таких систем.

1.4.1 Функціональні вимоги

Базові функції тренування.

Сучасні системи тренування швидкісного друку мають забезпечувати різноманітні режими практики для задоволення потреб користувачів різного рівня підготовки. Аналіз існуючих рішень показує, що найбільш ефективні системи пропонують режим вільного набору без обмежень, режим з обмеженням часу (від 1 до 10 хвилин), режим з обмеженням кількості символів (від 100 до 5000 символів) та режим з фіксованою кількістю помилок.

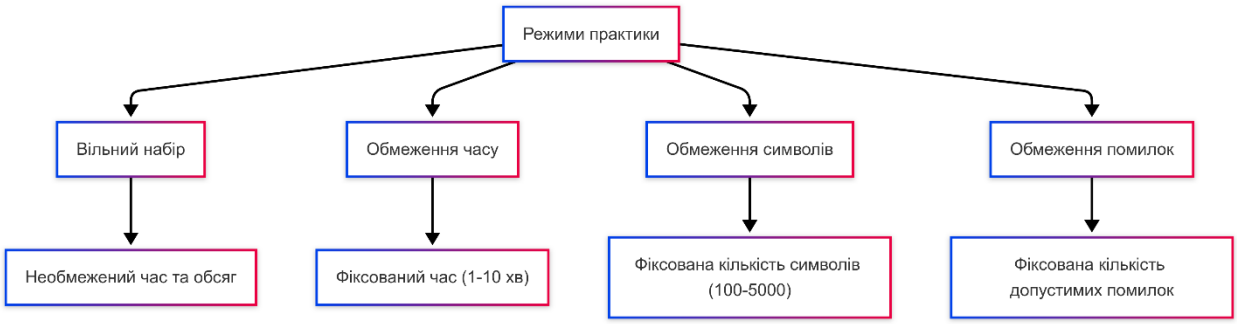

Рисунок 1.3 – Режими практики в сучасних системах

Важливою вимогою є підтримка різних розкладок клавіатури, включаючи QWERTY (американська, британська), AZERTY (французька), QWERTZ (німецька), Dvorak, Colemak, а також національні розкладки, зокрема українську та російську. Аналіз показує, що більшість існуючих

систем підтримують лише обмежений набір розкладок, що створює незручності для користувачів з нестандартними потребами.

Візуалізація процесу набору є критично важливою для ефективного навчання. Технічно досконалі системи забезпечують інтерактивну віртуальну клавіатуру з підсвічуванням клавіш, індикацію правильних/неправильних натискань, відображення поточної позиції в тексті, візуалізацію прогресу виконання завдання та індикатори швидкості й точності в реальному часі.

Управління текстовим контентом передбачає наявність бібліотеки готових текстів різної складності, можливість додавання власних текстів, автоматичну генерацію тренувальних текстів, фільтрацію текстів за тематикою, складністю, довжиною та підтримку різних мов і кодувань. Технічно розвинені системи містять бібліотеки з понад 1000 текстів різної складності та тематики.

Порівняльний аналіз існуючих систем показує значні відмінності у реалізації базових функцій. TypeRacer пропонує лише два режими практики та обмежену підтримку розкладок, тоді як TypingMaster забезпечує п'ять режимів та підтримку семи розкладок. Keybr вирізняється розширеною віртуальною клавіатурою та алгоритмами генерації текстів, але має обмежену бібліотеку готових текстів.

Аналітичні можливості.

Аналітичні можливості системи є ключовим фактором, що відрізняє ефективні тренажери від простих текстових редакторів. Технічно досконалі системи забезпечують збір та аналіз статистики, включаючи відстеження швидкості набору (WPM, CPM), вимірювання точності, аналіз часу реакції на символи, відстеження міжклавішних інтервалів та аналіз розподілу швидкості по різних ділянках клавіатури.

Класифікація та аналіз помилок є критично важливими для персоналізованого навчання. Технічно розвинені системи забезпечують виявлення найбільш проблемних символів, класифікацію помилок за типами (заміни, пропуски, перестановки, вставки), аналіз частотності помилок, виявлення патернів помилок та створення теплових карт помилок на клавіатурі.

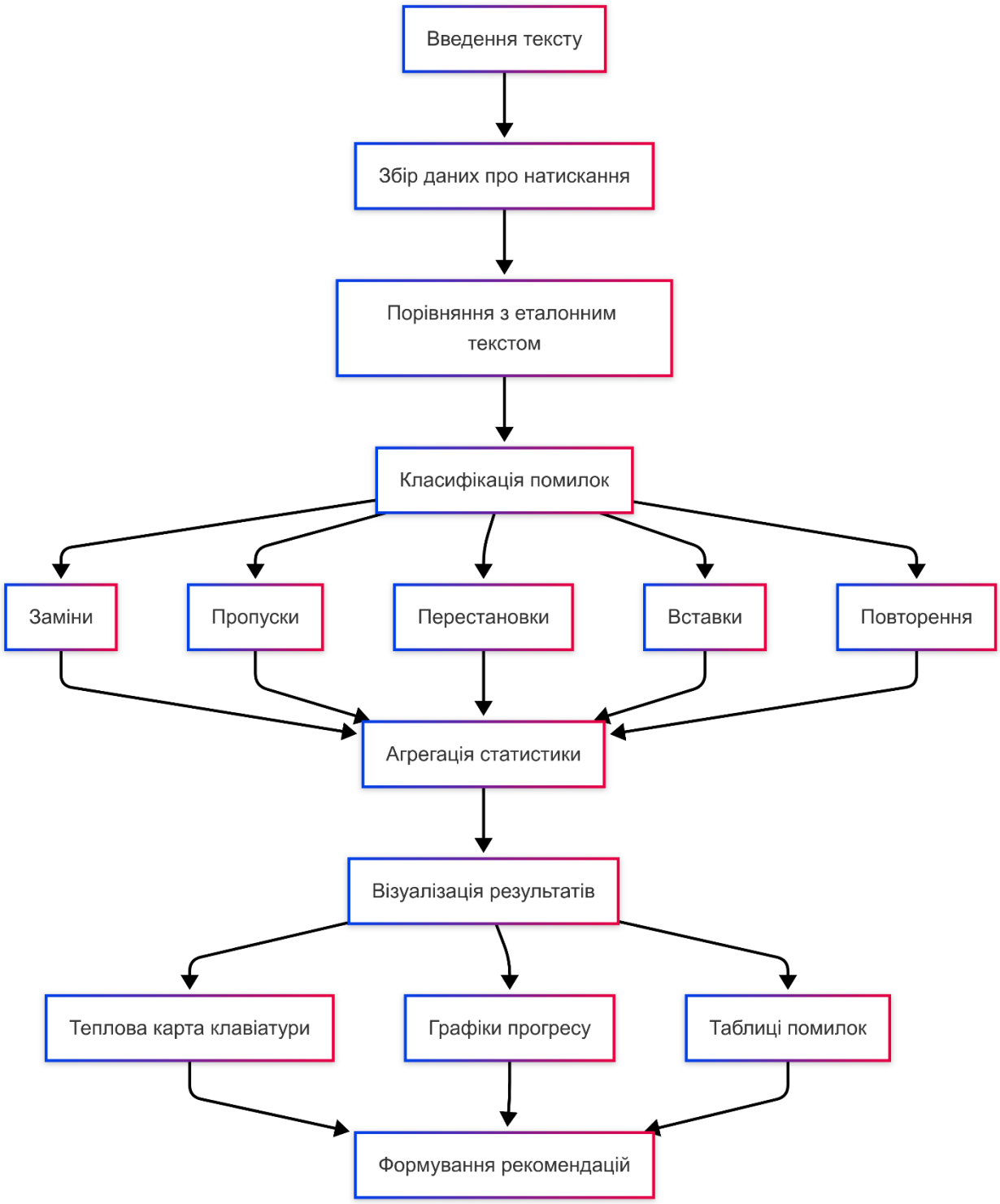

Рисунок 1.4 – Схема аналізу помилок

Історія та відстеження прогресу включає збереження результатів усіх сесій, порівняння результатів у часі, аналіз трендів покращення/погіршення, прогнозування майбутніх результатів та календар активності.

Порівняльний аналіз існуючих систем показує, що Keybr має найбільш розвинені аналітичні можливості, включаючи детальний аналіз помилок та теплові карти клавіатури. TypingMaster пропонує розширені метрики точності та детальний аналіз помилок, але має обмежені можливості

візуалізації. 10FastFingers має мінімальні аналітичні можливості, обмежуючись базовими метриками швидкості та точності.

Соціальні функції та гейміфікація.

Соціальні функції відіграють важливу роль у підтримці мотивації користувачів та створенні конкурентного середовища, що стимулює вдосконалення навичок. Технічно розвинені системи забезпечують змагання в реальному часі з іншими користувачами, асинхронні змагання з "привидами" (записами інших користувачів), створення приватних змагань з обмеженим доступом, турніри з елімінацією та командні змагання.

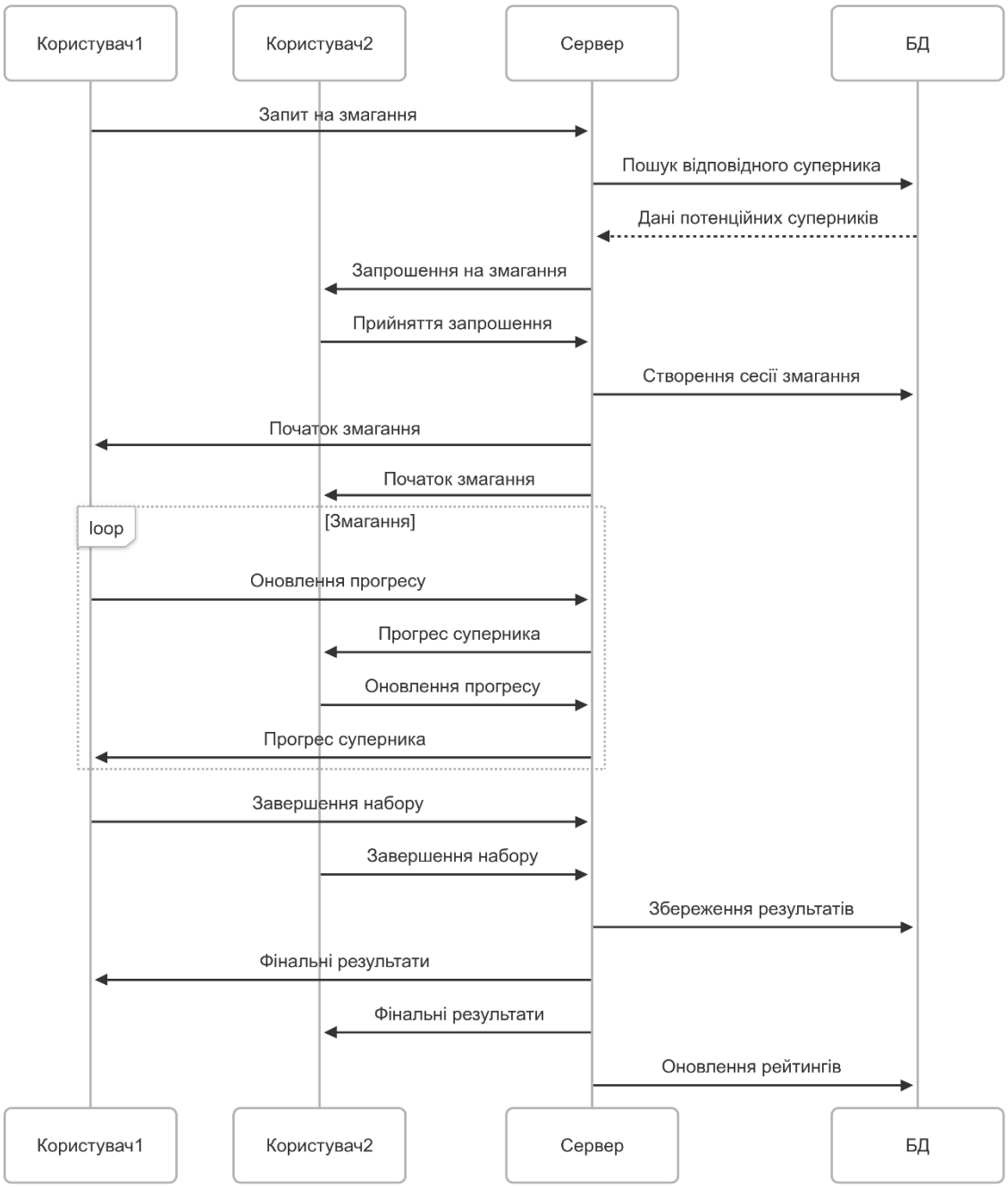

Рисунок 1.5 – Архітектура системи змагань

Рейтингова система включає глобальні таблиці лідерів, рейтинги за категоріями (швидкість, точність, прогрес), регіональні та мовні рейтинги, рейтинг Elo для змагань та сезонні рейтинги з оновленням.

Система досягнень передбачає наявність значків за досягнення певних показників, рівнів майстерності з візуальною індикацією, щоденних, тижневих та місячних викликів, спеціальних досягнень за унікальні результати та системи прогресу з відображенням відсотка завершення.

Соціальна взаємодія включає профілі користувачів з публічною статистикою, можливість додавання друзів, систему повідомлень між користувачами, інтеграцію з соціальними мережами та спільноти за інтересами.

Порівняльний аналіз існуючих систем показує, що TypeRacer має найбільш розвинені соціальні функції, включаючи змагання в реальному часі та глобальні таблиці лідерів. 10FastFingers пропонує обмежені змагання та глобальні рейтинги, але має обмежені можливості соціальної взаємодії. Keybr та Klavaro майже не мають соціальних функцій, фокусуючись на індивідуальному тренуванні.

Персоналізація та адаптивність.

Персоналізація є ключовим фактором, що дозволяє адаптувати систему до індивідуальних потреб та особливостей користувача, підвищуючи ефективність тренувань. Технічно розвинені системи забезпечують адаптивне навчання, включаючи автоматичне коригування складності завдань, фокусування на проблемних символах та комбінаціях, генерацію персоналізованих вправ, адаптивні алгоритми підбору текстів та динамічне коригування темпу навчання.

Налаштування інтерфейсу передбачає вибір теми оформлення, налаштування розміру та типу шрифту, зміну кольорової схеми, налаштування звукових ефектів та адаптацію для користувачів з обмеженими можливостями.

Персоналізовані плани тренувань включають створення індивідуальних планів тренувань, встановлення короткострокових та довгострокових цілей, відстеження прогресу відносно цілей, нагадування та сповіщення, а також рекомендації щодо оптимального режиму тренувань.

Налаштування складності передбачає регулювання складності текстів, фільтрацію за частотністю символів, налаштування обмежень часу та помилок, вибір тематики текстів та прогресивне збільшення складності.

Порівняльний аналіз існуючих систем показує, що Keybr має найбільш розвинені можливості адаптивного навчання, фокусуючись на проблемних символах. TypingMaster пропонує персоналізовані плани тренувань та розширені налаштування інтерфейсу, але має обмежені можливості адаптивного навчання. TypeRacer та 10FastFingers мають мінімальні можливості персоналізації, пропонуючи однакові тексти та завдання для всіх користувачів.

1.4.2 Технічні вимоги до архітектури системи

Архітектура системи тренування швидкісного друку повинна забезпечувати оптимальну продуктивність, масштабованість та надійність. На основі аналізу технічних характеристик існуючих рішень можна сформулювати наступні вимоги.

Клієнт-серверна архітектура.

Клієнтська частина повинна забезпечувати оптимізацію для швидкого відображення та обробки введення, мінімізацію затримок між натисканням клавіші та відображенням, ефективне використання кешування для зменшення мережевих запитів, адаптивний інтерфейс для різних пристроїв та оптимізацію використання пам'яті та процесора.

Серверна частина повинна забезпечувати багаторівневу архітектуру з розділенням відповідальності, ефективну обробку запитів з мінімальною затримкою, оптимізацію роботи з базою даних, кешування часто запитуваних даних та асинхронну обробку некритичних операцій.

Комунікація між клієнтом та сервером повинна забезпечувати використання WebSocket для комунікації в реальному часі, оптимізацію розміру пакетів даних, ефективні протоколи серіалізації, стиснення даних при передачі та механізми відновлення з'єднання.

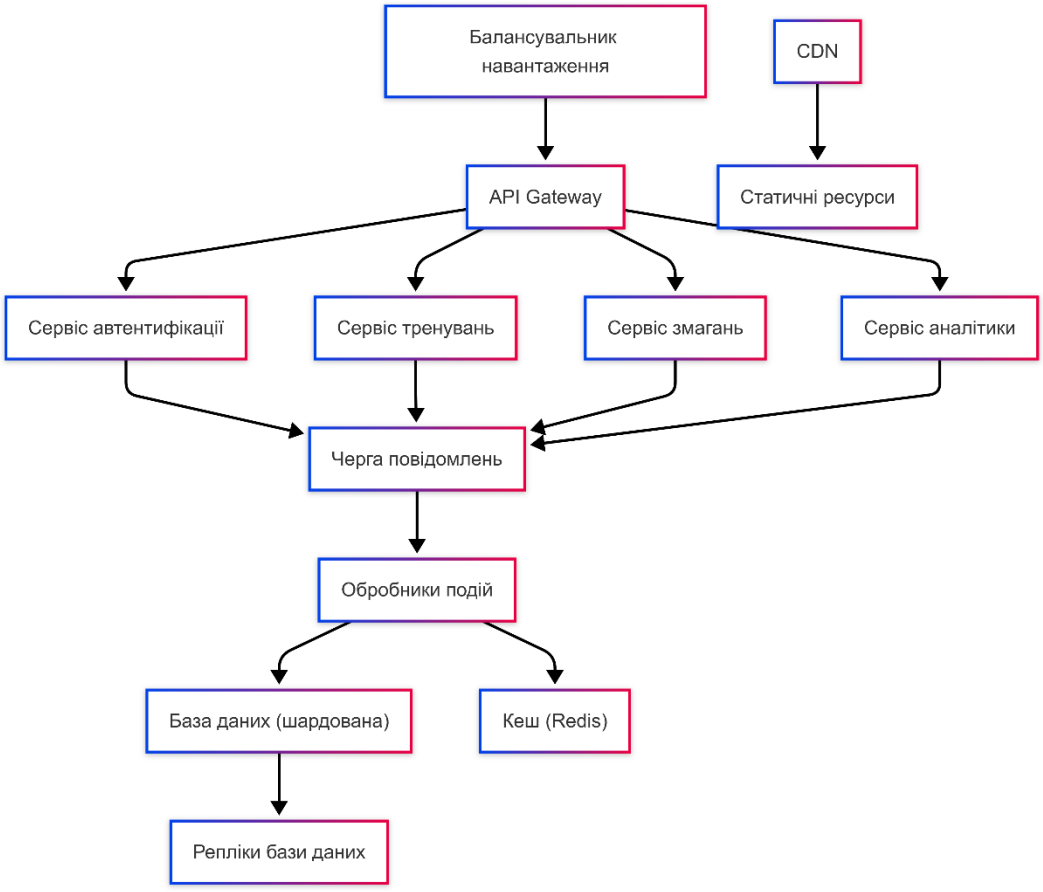

Рисунок 1.5 – Архітектура масштабованої системи

Вимоги до бази даних.

Структура даних повинна забезпечувати оптимізовану схему для швидкого доступу до часто використовуваних даних, ефективне зберігання історичних даних, нормалізацію для мінімізації дублювання, індексування критичних полів для швидкого пошуку та партиціонування для розподілу навантаження.

Продуктивність бази даних повинна забезпечувати оптимізацію запитів для мінімізації часу відгуку, кешування результатів частих запитів, ефективне використання індексів, оптимізацію з'єднань таблиць та моніторинг і оптимізацію продуктивності.

Масштабованість бази даних повинна забезпечувати горизонтальне масштабування для обробки зростаючого обсягу даних, шардинг для розподілу навантаження, реплікацію для підвищення доступності, стратегії архівування історичних даних та оптимізацію для роботи з великими обсягами даних.

Вимоги до інфраструктури.

Розгортання повинно забезпечувати контейнеризацію для забезпечення ізоляції та портативності, оркестрацію контейнерів для автоматичного масштабування, автоматизоване розгортання з мінімальним простоєм, стратегії розгортання з мінімізацією ризиків (blue-green, canary) та інфраструктуру як код для відтворюваності середовища.

Моніторинг та логування повинні забезпечувати збір метрик продуктивності в реальному часі, централізоване логування з можливістю пошуку та аналізу, сповіщення про аномалії та інциденти, трасування запитів через різні компоненти системи та візуалізацію метрик для аналізу трендів.

Відмовостійкість повинна забезпечувати автоматичне відновлення після збоїв, розподіл навантаження між кількома серверами, географічну реплікацію для захисту від локальних збоїв, автоматичне масштабування при зростанні навантаження та деградацію функціональності замість повної відмови.

1.4.3 Порівняльний аналіз відповідності існуючих рішень вимогам

На основі проведеного аналізу функціональних та нефункціональних вимог можна оцінити відповідність існуючих рішень цим вимогам. Аналіз показує, що жодне з існуючих рішень не задовольняє всі ключові вимоги користувачів, особливо в частині персоналізації та адаптивного навчання.

Десктопні рішення (TypingMaster, Klavaro) мають перевагу в швидкодії та автономності, але поступаються в соціальних функціях та доступності. Веб-рішення (TypeRacer, Keybr, 10FastFingers) мають кращу доступність та соціальні функції, але часто поступаються в персоналізації та адаптивності.

Найбільш суттєві недоліки існуючих рішень включають обмежені можливості аналізу помилок та персоналізації тренувань, недостатню інтеграцію з іншими системами, обмежену підтримку різних мов та

розкладок клавіатури, а також недостатню адаптацію для користувачів з обмеженими можливостями.

Розроблювана система спрямована на усунення цих недоліків шляхом впровадження адаптивних алгоритмів навчання, розширення можливостей аналізу та візуалізації даних, забезпечення гнучкої інтеграції з іншими системами, підтримки широкого спектру мов та розкладок, а також дотримання стандартів доступності.

Таким чином, аналіз вимог підтверджує актуальність розробки нової системи тренування швидкісного друку, що поєднує переваги існуючих рішень та усуває їхні недоліки.

Висновки за розділом

Проведений аналіз сучасних систем тренування швидкісного друку дозволяє сформувати комплексне розуміння вимог до таких систем та оцінити відповідність існуючих рішень цим вимогам. Дослідження показало, що ефективна система тренування швидкісного друку повинна забезпечувати баланс між функціональністю, продуктивністю, надійністю та зручністю використання.

У функціональному аспекті критично важливими є різноманітні режими практики, підтримка різних розкладок клавіатури, візуалізація процесу набору, управління текстовим контентом, аналітичні можливості, соціальні функції та персоналізація. Особливу увагу слід приділити адаптивним алгоритмам навчання, які дозволяють фокусуватися на проблемних символах та комбінаціях, а також аналізу помилок для формування персоналізованих рекомендацій.

Нефункціональні вимоги, такі як швидкодія інтерфейсу, масштабованість, надійність, безпека та доступність, відіграють не менш важливу роль у забезпеченні ефективного навчання. Мінімальна затримка відображення введених символів, стабільна робота під навантаженням, захист від шахрайства та підтримка різних платформ є необхідними умовами для створення конкурентоспроможної системи.

Технічна архітектура системи повинна забезпечувати оптимальну продуктивність, масштабованість та надійність. Клієнт-серверна архітектура з використанням сучасних технологій, оптимізована база даних та надійна інфраструктура є основою для реалізації всіх функціональних та нефункціональних вимог.

Порівняльний аналіз існуючих рішень виявив, що жодне з них не задовольняє всі ключові вимоги користувачів. Десктопні рішення мають перевагу в швидкодії та автономності, але поступаються в соціальних функціях та доступності. Веб-рішення мають кращу доступність та соціальні функції, але часто поступаються в персоналізації та адаптивності.

Найбільш суттєві недоліки існуючих рішень включають обмежені можливості аналізу помилок та персоналізації тренувань, недостатню інтеграцію з іншими системами, обмежену підтримку різних мов та

розкладок клавіатури, а також недостатню адаптацію для користувачів з обмеженими можливостями.

Розроблювана система KeyRaces спрямована на усунення цих недоліків шляхом впровадження адаптивних алгоритмів навчання, розширення можливостей аналізу та візуалізації даних, забезпечення гнучкої інтеграції з іншими системами, підтримки широкого спектру мов та розкладок, а також дотримання стандартів доступності.

Архітектурні рішення, представлені в діаграмах, демонструють підхід до забезпечення масштабованості, надійності та безпеки системи. Модульна структура з чітким розділенням відповідальності дозволяє забезпечити гнучкість та розширюваність системи, а використання сучасних технологій та підходів до розробки забезпечує відповідність сучасним стандартам якості програмного забезпечення.

Таким чином, аналіз вимог підтверджує актуальність розробки нової системи тренування швидкісного друку KeyRaces, що поєднує переваги існуючих рішень та усуває їхні недоліки. Система має потенціал стати ефективним інструментом для навчання та вдосконалення навичок швидкісного друку для широкого кола користувачів, від початківців до професіоналів.

2. ПРОЄКТУВАННЯ ВЕБ-СЕРВІСУ ДЛЯ ТРЕНУВАННЯ

2.1 Архітектура веб-сервісу та вибір технологій розробки

Архітектура веб-сервісу KeyRaces розроблена з урахуванням вимог до функціональності, продуктивності, масштабованості та безпеки, визначених у першому розділі. Проєкт побудований на основі сучасних технологій та архітектурних підходів, що забезпечують ефективну реалізацію всіх необхідних функцій системи тренування швидкісного друку.

2.1.1 Загальна архітектура системи

Веб-сервіс KeyRaces реалізований на основі багаторівневої архітектури з чітким розділенням відповідальності між компонентами. Система складається з трьох основних рівнів: рівень ядра (Core), рівень інфраструктури (Infrastructure) та рівень представлення (Server).

Рівень ядра містить бізнес-сутності, інтерфейси та моделі даних, що визначають основну функціональність системи. Цей рівень є незалежним від конкретних технологій реалізації та визначає абстрактні інтерфейси для взаємодії з зовнішніми ресурсами.

Рівень інфраструктури реалізує інтерфейси, визначені на рівні ядра, забезпечуючи доступ до бази даних, зовнішніх сервісів та інших ресурсів. Тут розміщуються конкретні реалізації репозиторіїв, сервісів та інших компонентів, що взаємодіють з зовнішніми системами.

Рівень представлення забезпечує взаємодію з користувачем через веб-інтерфейс та API. Цей рівень включає контролери, представлення, моделі представлення та інші компоненти, необхідні для забезпечення користувацького інтерфейсу та API.

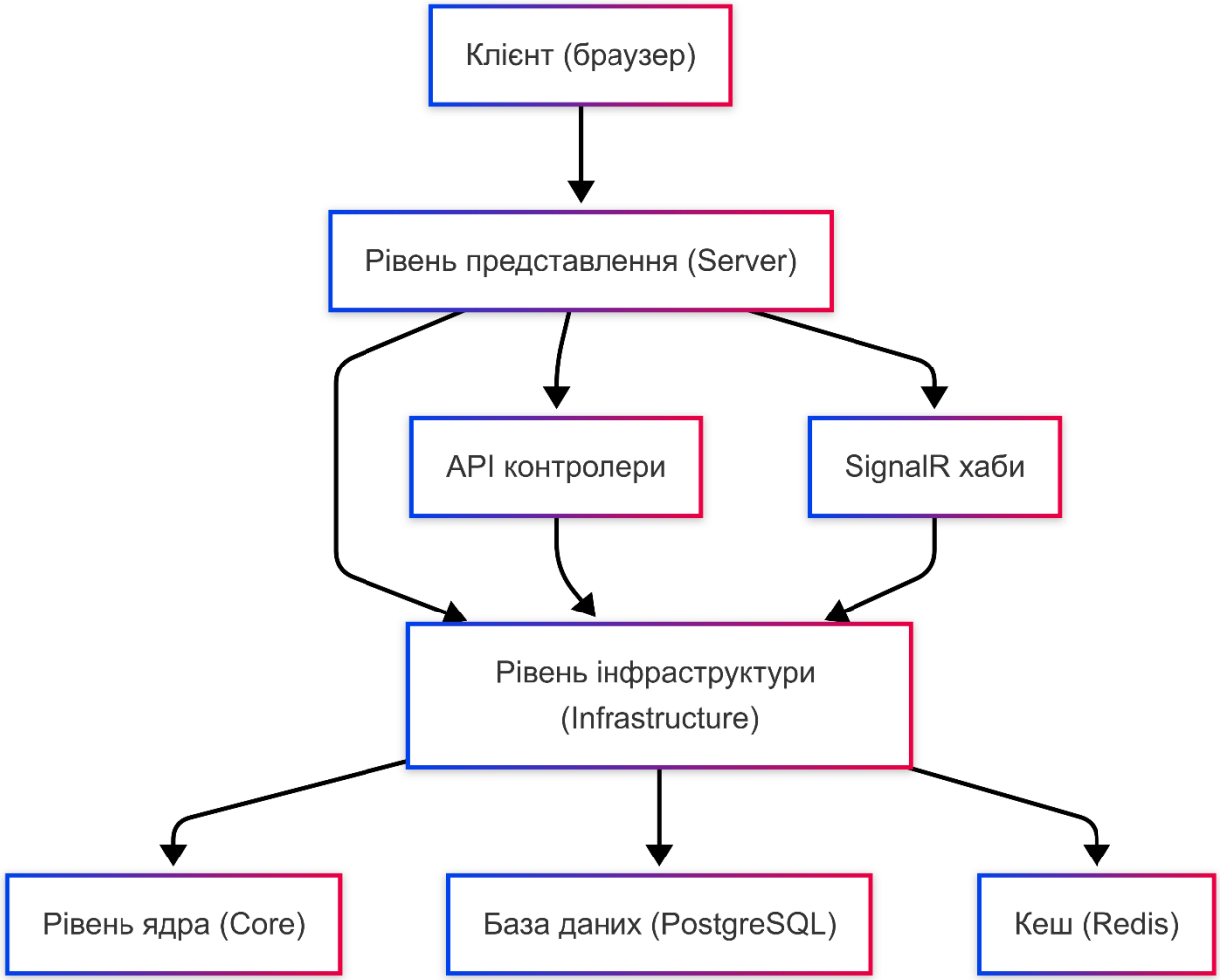

Рисунок 2.1 – Багаторівнева архітектура проєкту

Така архітектура забезпечує модульність, розширюваність, тестованість та підтримуваність системи. Компоненти можуть розроблятися та тестуватися незалежно, нові функції можуть бути додані без значних змін в існуючому коді, а чітке розділення відповідальності спрощує написання модульних та інтеграційних тестів та підтримку коду.

2.1.2 Вибір технологій серверної частини

Для реалізації серверної частини веб-сервісу KeyRaces обрано платформу .NET 8.0 з мовою програмування C#. Цей вибір обґрунтований високою продуктивністю, кросплатформністю, багатим екосистемом, сильною типізацією та підтримкою асинхронного програмування [12].

.NET 8.0 забезпечує високу швидкодію, що критично важливо для системи тренування швидкісного друку, де затримки можуть негативно впливати на користувацький досвід. Кросплатформність дозволяє розгортати систему на різних операційних системах, включаючи Windows, Linux та macOS, що розширює можливості для хостингу та розгортання.

Багата екосистема .NET включає велику кількість бібліотек та інструментів для розробки, що прискорює процес створення та підтримки системи. Сильна типізація C# забезпечує надійність та безпеку коду,

зменшуючи кількість помилок під час виконання. Підтримка асинхронного програмування через механізми async/await дозволяє ефективно обробляти запити без блокування потоків виконання.

Серверна частина реалізована з використанням ASP.NET Core, що забезпечує високу продуктивність, модульність та безпеку. ASP.NET Core оптимізований для обробки великої кількості одночасних запитів, що важливо для системи з потенційно великою кількістю користувачів. Система залежностей ASP.NET Core дозволяє гнучко конфігурувати компоненти та їх взаємодію. Вбудовані механізми безпеки захищають від поширених вразливостей, таких як CSRF, XSS та інші.

Для забезпечення комунікації в реальному часі, особливо важливої для змагань з набору тексту, використовується SignalR. Ця бібліотека дозволяє створювати додатки з двостороннім зв'язком між клієнтом та сервером, забезпечуючи миттєве оновлення прогресу учасників змагань та інтерактивну взаємодію.

2.1.3 Вибір технологій зберігання даних

Для зберігання даних у системі KeyRaces використовується комбінація технологій: PostgreSQL як основна реляційна база даних та Redis як система кешування та сховище ключ-значення [14].

PostgreSQL забезпечує надійне зберігання структурованих даних, підтримку складних запитів та транзакцій, масштабованість та високу продуктивність, а також розширені можливості для роботи з JSON та іншими типами даних. Ця СУБД є оптимальним вибором для зберігання даних користувачів, сесій тренувань, змагань, текстових даних та статистики.

Redis використовується для кешування часто запитуваних даних, зберігання сесійних даних, забезпечення швидкого доступу до даних змагань у реальному часі та реалізації черг повідомлень для асинхронної обробки. Висока швидкодія Redis дозволяє забезпечити миттєвий доступ до даних, що критично важливо для інтерактивних змагань та аналізу в реальному часі.

Доступ до бази даних реалізований з використанням Entity Framework Core – ORM-фреймворку, що забезпечує абстракцію від конкретної СУБД, типобезпечний доступ до даних, автоматичне відстеження змін та міграції для керування схемою бази даних [15].

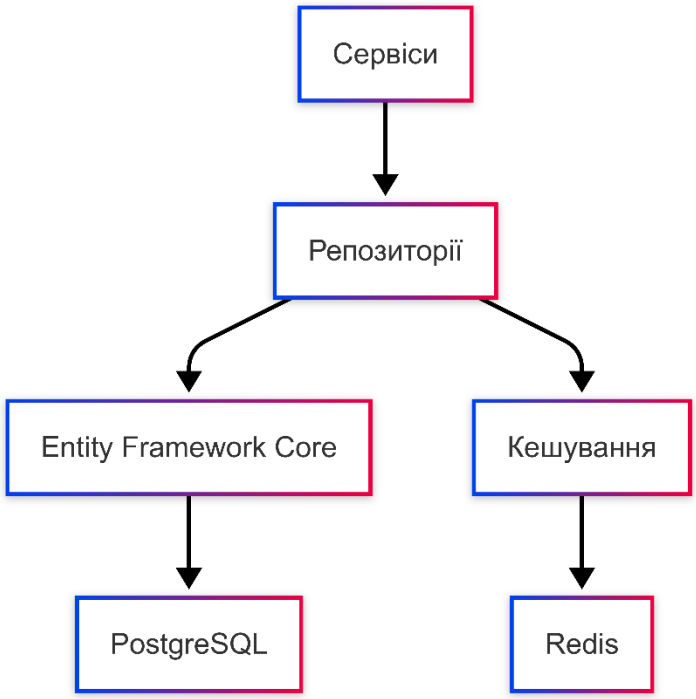

Рисунок 2.2 – Архітектура доступу до даних

2.1.4 Контейнеризація та розгортання

Для забезпечення стабільного та передбачуваного розгортання веб-сервісу KeyRaces використовується контейнеризація на основі Docker. Контейнеризація забезпечує ізоляцію компонентів системи, стандартизацію середовища розробки та виробництва, спрощення масштабування системи та автоматизацію процесу розгортання.

Система розгортається з використанням Docker Compose, що дозволяє оркеструвати взаємодію між контейнерами. Архітектура контейнеризації включає контейнер з веб-сервісом KeyRaces, контейнер з базою даних PostgreSQL, контейнер з Redis для кешування та контейнер з Ollama для генерації текстів.

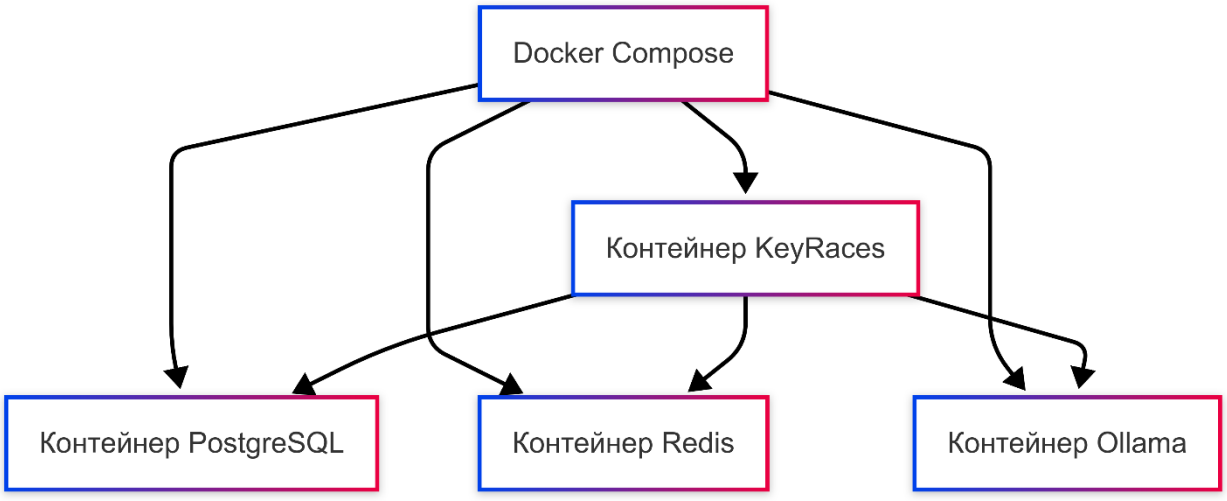

Рисунок 2.3 – Архітектура контейнеризації

Такий підхід до розгортання забезпечує гнучкість, масштабованість та надійність системи, а також спрощує процес розробки та тестування.

2.1.5 Генерація текстів з використанням LLM

Однією з інноваційних особливостей веб-сервісу KeyRaces є використання локальної моделі машинного навчання (LLM) для генерації тренувальних текстів. Це реалізовано через інтеграцію з Ollama – інструментом для запуску великих мовних моделей локально.

Архітектура генерації текстів включає сервіс «LocalLLMTextGenerationService», що реалізує інтерфейс «ITextGenerationService», HTTP-клієнт для взаємодії з API Ollama, механізми обробки помилок та резервні варіанти генерації тексту, а також кешування згенерованих текстів для оптимізації продуктивності.

Використання локальної LLM забезпечує генерацію унікальних текстів різної складності та тематики, підтримку різних мов (англійська, українська, російська), адаптацію складності тексту до рівня користувача та незалежність від зовнішніх API та сервісів.

Сервіс генерації текстів взаємодіє з Ollama через HTTP API, відправляючи запити з параметрами генерації (тема, складність, довжина, мова) та отримуючи згенерований текст. У разі недоступності Ollama або помилок генерації, сервіс використовує резервний механізм генерації текстів на основі заздалегідь підготовлених шаблонів.

2.1.6 Безпека та автентифікація

Система безпеки веб-сервісу KeyRaces реалізована з використанням сучасних підходів та технологій: JWT) для автентифікації та авторизації користувачів, ASP.NET Core Identity для управління користувачами та ролями, HTTPS для захищеного з'єднання, а також захист від CSRF, XSS та інших поширених вразливостей.

Процес автентифікації включає реєстрацію користувача з валідацією даних, вхід з генерацією JWT-токена та refresh-токена, авторизацію запитів на основі JWT-токенів, оновлення токенів для забезпечення безперервної сесії та вихід з системи з інвалідацією токенів.

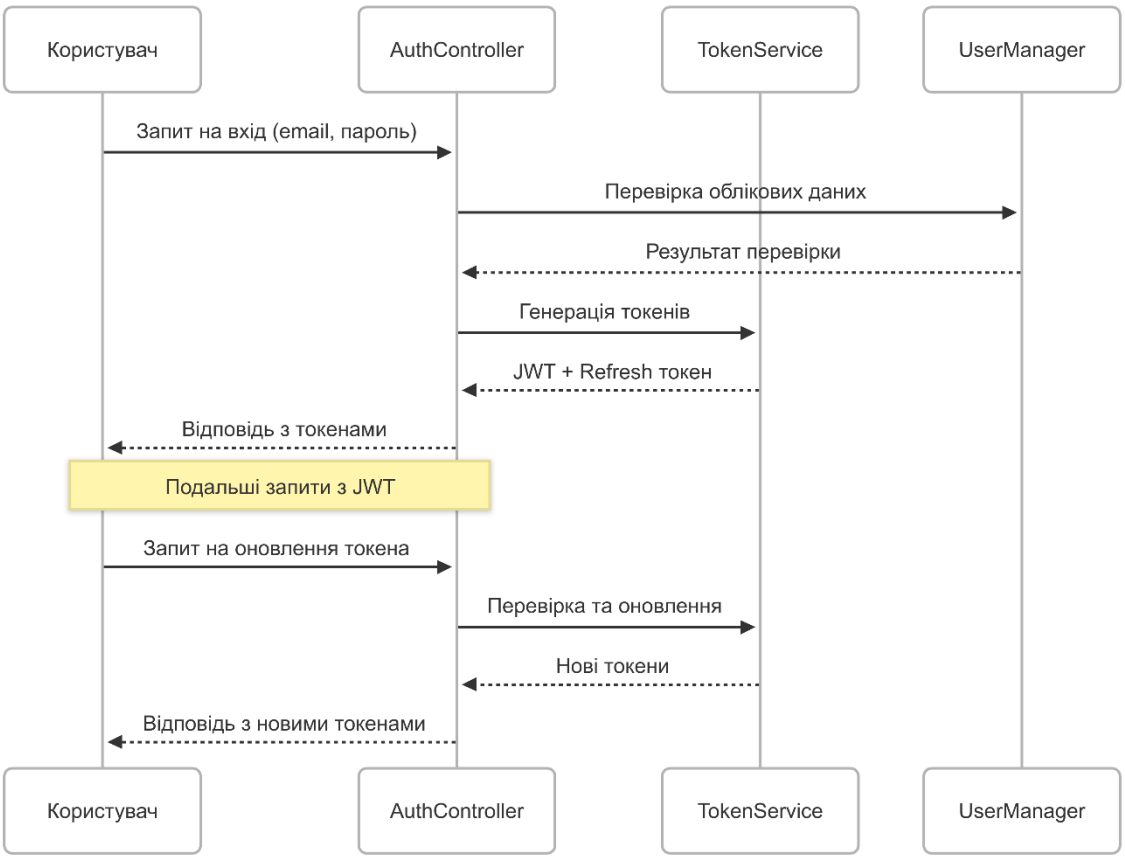

Рисунок 2.4 – Процес автентифікації

Така система автентифікації забезпечує безпеку облікових даних користувачів, захист від несанкціонованого доступу та зручність використання системи.

2.1.7 Комунікація в реальному часі

Для забезпечення інтерактивних змагань з набору тексту та миттєвого оновлення прогресу учасників використовується технологія SignalR, що дозволяє встановлювати двосторонній зв'язок між клієнтом та сервером.

Архітектура комунікації в реальному часі включає «TypingHub» центральний компонент для обміну повідомленнями, групи для організації учасників змагань, методи для приєднання до змагань та звітування про прогрес, а також механізми трансляції оновлень всім учасникам змагання.

Використання SignalR забезпечує мінімальну затримку при передачі даних, автоматичне відновлення з'єднання при втраті зв'язку, масштабованість для підтримки великої кількості одночасних змагань та сумісність з різними браузерами та клієнтами.

Під час змагання кожен учасник приєднується до групи, що відповідає ідентифікатору змагання, через метод «JoinRace». Прогрес учасника (швидкість набору в WPM) передається на сервер через метод «ReportProgress», який потім транслює цю інформацію всім учасникам групи.

Це дозволяє в реальному часі відображати позиції учасників та створювати ефект присутності в змаганні.

Таким чином, архітектура веб-сервісу KeyRaces та обрані технології розробки забезпечують реалізацію всіх функціональних та нефункціональних вимог, визначених у першому розділі, створюючи надійну, масштабовану та продуктивну платформу для тренування швидкісного друку.

2.2 Проєктування бази даних та моделей даних

Проєктування бази даних для веб-сервісу KeyRaces здійснювалося з урахуванням вимог до функціональності, продуктивності та масштабованості системи. База даних повинна забезпечувати ефективне зберігання та доступ до даних користувачів, текстових фрагментів, сесій тренувань, змагань та статистики.

2.2.1 Концептуальна модель даних

Концептуальна модель даних KeyRaces включає наступні основні сутності: користувачі (UserProfile), текстові фрагменти (TextSnippet), сесії тренування (TypingSession), змагання (Competition), учасники змагань (CompetitionParticipant), статистика набору (TypingStatistic), досягнення (Achievement) та записи таблиці лідерів (LeaderboardEntry).

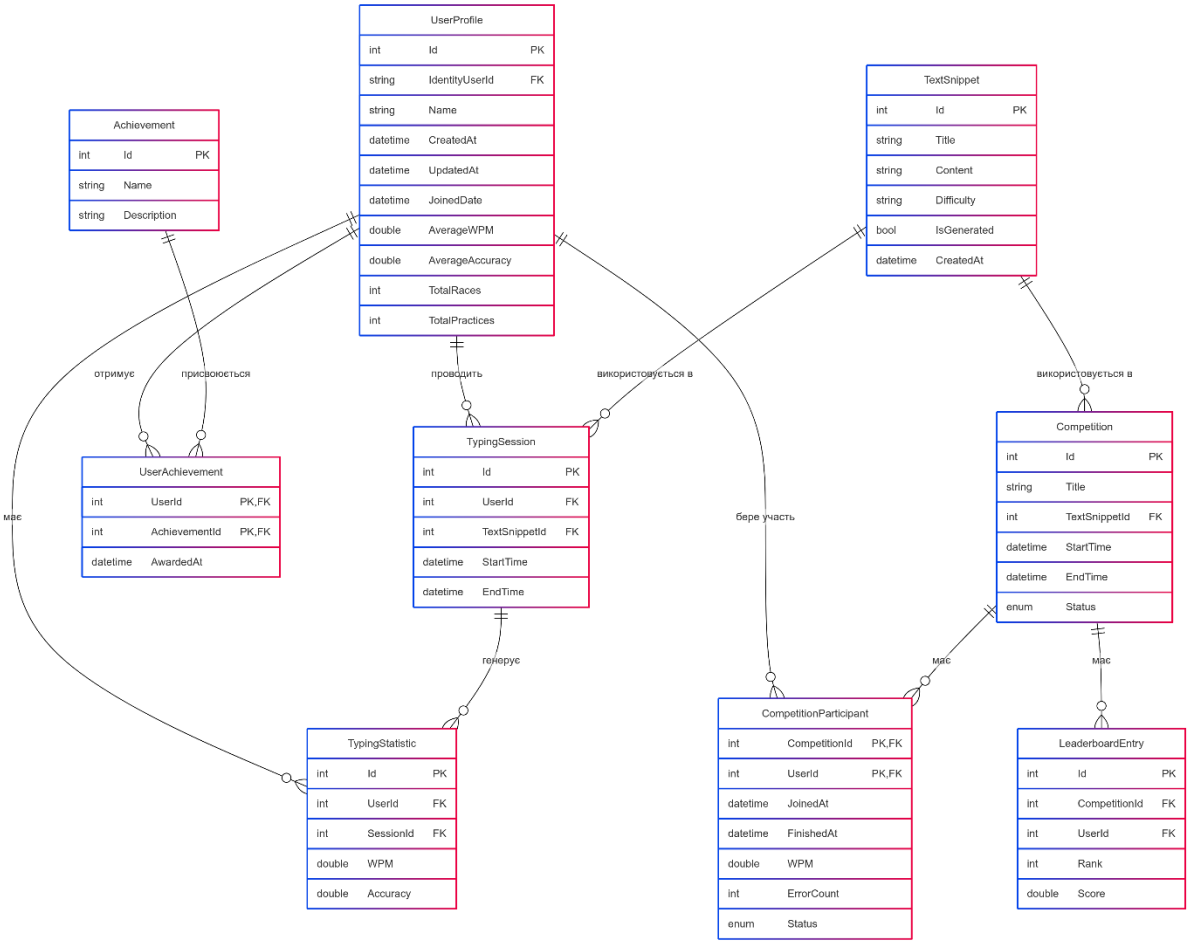

Рисунок 2.5 – ER- діаграма проєкту

2.2.2 Опис основних сутностей

UserProfile представляє користувача системи та містить інформацію про його профіль, статистику та прогрес. Сутність пов'язана з ідентифікатором користувача в системі ASP.NET Core Identity через поле IdentityUserId. Профіль користувача зберігає загальну статистику (середня швидкість, точність, кількість змагань та тренувань), а також дати створення та оновлення.

TextSnippet представляє текстовий фрагмент для тренування або змагання. Кожен фрагмент має заголовок, вміст, рівень складності та ознаку, чи був він згенерований автоматично. Текстові фрагменти використовуються як в індивідуальних тренуваннях, так і в змаганнях.

TypingSession представляє сесію тренування користувача. Сесія пов'язана з користувачем та текстовим фрагментом, має час початку та завершення. На основі сесій формується статистика набору тексту.

TypingStatistic зберігає статистику набору тексту для конкретної сесії тренування. Статистика включає швидкість набору (WPM) та точність (відсоток правильно набраних символів).

Competition представляє змагання з набору тексту. Змагання має заголовок, пов'язаний текстовий фрагмент, час початку та завершення, а також статус (заплановане, в процесі, завершене).

CompetitionParticipant представляє учасника змагання. Ця сутність є зв'язуючою між користувачем та змаганням, містить інформацію про час приєднання, завершення, швидкість набору, кількість помилок та статус участі.

Achievement представляє досягнення, яке може бути присвоєне користувачу. Досягнення має назву та опис.

UserAchievement є зв'язуючою сутністю між користувачем та досягненням, містить інформацію про дату присвоєння досягнення.

LeaderboardEntry представляє запис у таблиці лідерів для конкретного змагання. Запис містить інформацію про користувача, змагання, ранг та рахунок.

2.2.3 Реалізація бази даних

База даних KeyRaces реалізована з використанням PostgreSQL та Entity Framework Core. Для кожної сутності створено відповідний клас моделі даних, що відображається на таблицю в базі даних. Відношення між сутностями реалізовані через зовнішні ключі та навігаційні властивості.

Для забезпечення цілісності даних використовуються обмеження на рівні бази даних, включаючи первинні та зовнішні ключі, унікальні індекси та перевірки значень. Для підвищення продуктивності запитів створено індекси для полів, що часто використовуються у фільтрації та сортуванні.

Міграції бази даних керуються через Entity Framework Core, що забезпечує контрольоване оновлення схеми бази даних при розгортанні нових версій системи. Історія міграцій зберігається в коді проєкту, що

дозволяє відстежувати зміни схеми та забезпечує можливість відкату до попередніх версій.

Таким чином, спроєктована база даних забезпечує ефективне зберігання та доступ до даних веб-сервісу KeyRaces, підтримуючи всі необхідні функціональні можливості системи тренування швидкісного друку.

2.3 Розробка серверної частини та API

Серверна частина веб-сервісу KeyRaces реалізована з використанням ASP.NET Core та забезпечує взаємодію клієнтської частини з базою даних та іншими компонентами системи. API сервісу спроєктовано за принципами REST, що забезпечує стандартизований та передбачуваний інтерфейс для взаємодії з системою.

2.3.1 Структура API

API веб-сервісу KeyRaces організовано навколо основних ресурсів системи: користувачі, текстові фрагменти, сесії тренування, змагання та статистика. Для кожного ресурсу реалізовано відповідний контролер, що забезпечує операції створення, читання, оновлення та видалення (CRUD).

2.3.2 Автентифікація та авторизація

Автентифікація в API реалізована з використанням JWT (JSON Web Tokens). AuthController забезпечує ендпоінти для реєстрації, входу, оновлення токенів та виходу з системи.

Процес автентифікації починається з реєстрації користувача через ендпоінт «/api/auth/register», де передаються ім'я, електронна пошта та пароль. Після успішної реєстрації користувач може увійти в систему через ендпоінт «/api/auth/login», отримавши у відповідь JWT-токен та refresh-токен.

JWT-токен використовується для автентифікації запитів до захищених ендпоінтів API. Токен передається в заголовку Authorization у форматі Bearer. Термін дії токена обмежений, тому для забезпечення безперервної сесії використовується механізм оновлення токенів через ендпоінт «/api/auth/refresh»

2.3.3 Управління текстовими фрагментами

TextSnippetController забезпечує ендпоінти для роботи з текстовими фрагментами, що використовуються для тренувань та змагань. API дозволяє отримувати список фрагментів з фільтрацією за складністю, додавати нові фрагменти та оновлювати існуючі.

Особливістю API є можливість генерації нових текстових фрагментів за допомогою моделі машинного навчання через ендпоінт «/api/textgeneration/generate». Цей ендпоінт приймає параметри генерації (тема, складність, довжина) та повертає згенерований текст, який може бути збережений як новий текстовий фрагмент.

2.3.4 Управління сесіями тренування

SessionController забезпечує ендпоінти для роботи з сесіями тренування. API дозволяє створювати нові сесії, отримувати інформацію про сесію, завершувати сесію та отримувати список сесій користувача.

Процес тренування починається зі створення нової сесії через ендпоінт «/api/session», де передаються ідентифікатори користувача та текстового фрагмента. Після завершення тренування результати відправляються на ендпоінт «/api/session/{id}/complete», де зберігаються швидкість набору та кількість помилок.

2.3.5 Управління змаганнями

CompetitionController та ParticipantController забезпечують ендпоінти для роботи зі змаганнями та учасниками змагань. API дозволяє створювати нові змагання, отримувати інформацію про змагання, приєднуватися до змагань та отримувати список учасників змагання.

Процес змагання починається зі створення нового змагання через ендпоінт «/api/competition», де передаються заголовок, ідентифікатор текстового фрагмента та час початку. Користувачі можуть приєднатися до змагання через ендпоінт «/api/participant/join», де передається ідентифікатор змагання.

Під час змагання прогрес учасників відстежується через WebSocket з'єднання, реалізоване за допомогою SignalR. Клієнти підключаються до хабу «TypingHub» та відправляють оновлення прогресу, які транслюються всім учасникам змагання.

2.3.6 Обробка помилок та валідація

API веб-сервісу KeyRaces включає механізми обробки помилок та валідації вхідних даних. Для валідації використовуються атрибути валідації моделей даних та фільтри дій, що забезпечують перевірку вхідних даних перед їх обробкою.

Обробка помилок реалізована через глобальний обробник винятків, що перехоплює всі необроблені винятки та повертає відповідні HTTP-статуси та повідомлення про помилки. Це забезпечує уніфікований формат відповідей про помилки та запобігає витоку чутливої інформації.

Таким чином, API веб-сервісу KeyRaces забезпечує повний набір функціональності для взаємодії клієнтської частини з системою, включаючи автентифікацію, управління текстовими фрагментами, сесіями тренування, змаганнями та досягненнями. API спроєктовано за принципами REST, що забезпечує стандартизований та передбачуваний інтерфейс для взаємодії з системою.

Висновок за розділом

У другому розділі спроєктовано архітектуру веб-сервісу KeyRaces для тренування швидкісного друку відповідно до вимог, визначених у першому розділі. Запропонована багаторівнева архітектура з чітким розділенням відповідальності забезпечує модульність та масштабованість системи, необхідні для ефективної роботи сервісу.

Спроєктована структура бази даних ефективно моделює предметну область, включаючи сутності користувачів, текстових фрагментів, сесій тренувань та змагань. Розроблена архітектура API з використанням REST принципів та SignalR для комунікації в реальному часі забезпечує всі необхідні функції для взаємодії клієнтської частини з системою.

Обрані технології (.NET 8.0, PostgreSQL, Redis, Docker) та архітектурні рішення створюють надійну основу для подальшої реалізації веб-сервісу KeyRaces, що відповідає всім функціональним та нефункціональним вимогам.

3. РОЗРОБКА ВЕБ-СЕРВІСУ

3.1 Ініціалізація проєкту та налаштування середовища розробки

На початковому етапі розробки програмного забезпечення KeyRaces було закладено фундамент для подальшої реалізації функціональності. Цей етап включав створення структури проєкту, налаштування системи контролю версій, конфігурацію середовища розробки та підготовку інструментів для контейнеризації.

Розробка проєкту розпочалася зі створення рішення (solution) у середовищі .NET. Було обрано платформу .NET 8 та мову програмування C# з огляду на їхню високу продуктивність, кросплатформність та розвинену екосистему для веб-розробки. Структура рішення була організована за принципами багатошарової архітектури для забезпечення чіткого розподілу відповідальності та підвищення модульності коду. Було створено наступні ключові проєкти:

– «keyraces.Core»: призначений для розміщення бізнес-логіки, сутностей домену, інтерфейсів сервісів та моделей передачі даних. Цей проєкт є ядром системи і не залежить від конкретних технологій інфраструктури чи представлення.

– «keyraces.Infrastructure»: містить реалізації інтерфейсів, визначених у «keyraces.Core», зокрема для взаємодії з базою даних (репозиторії), зовнішніми сервісами (наприклад, генерація текстів) та іншими інфраструктурними компонентами.

– «keyraces.Server»: відповідає за рівень представлення, включаючи реалізацію API ендпоінтів за допомогою ASP.NET Core, обробку HTTP-запитів, автентифікацію, авторизацію та взаємодію з клієнтською частиною. Також цей проєкт містить конфігурацію веб-додатку та налаштування Blazor компонентів для користувацького інтерфейсу.

Одночасно зі створенням структури прокту було ініціалізовано систему контролю версій Git. Для централізованого зберігання коду, відстеження змін та спільної роботи (навіть у випадку індивідуальної розробки для забезпечення резервного копіювання та історії змін) було використано платформу GitHub. Кожна значуща зміна або додавання нової функціональності фіксувалися у вигляді комітів, що дозволяло за потреби відкочуватися до попередніх стабільних версій проєкту у випадку виникнення критичних помилок або непередбачуваної поведінки системи.

Налаштування середовища розробки передбачало встановлення актуальної версії .NET 8 SDK, що включає необхідні бібліотеки та інструменти командного рядка (dotnet CLI) для компіляції, тестування та запуску проєкту.

Як основне інтегроване середовище розробки (IDE) використовувався інструмент, що підтримує розробку на .NET – Visual Studio, забезпечуючи зручні засоби для написання коду, налагодження та управління проєктом.

Для забезпечення узгодженого середовища розробки та підготовки до подальшого розгортання було прийнято рішення про використання технології контейнеризації Docker. На ранньому етапі було створено початковий «Dockerfile» для проєкту «keyraces.Server». Цей файл містив інструкції для побудови образу додатку, включаючи копіювання вихідного коду, відновлення залежностей, компіляцію проєкту та визначення команди для запуску серверної частини.

Приклад частини «Dockerfile»:

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY ["keyraces.Server/keyraces.Server.csproj", "keyraces.Server/"]
# ... копіювання інших .csproj файлів та відновлення залежностей ...
RUN dotnet restore "keyraces.Server/keyraces.Server.csproj"
COPY . .
WORKDIR "/src/keyraces.Server"
RUN dotnet publish "keyraces.Server.csproj" -c Release -o /app/publish

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "keyraces.Server.dll"]
```

Також було створено файл «docker-compose.ym» для оркестрації взаємодії між контейнером додатку та контейнерами залежностей, такими як система управління базами даних PostgreSQL та система кешування Redis. У «docker-compose.yml» було визначено сервіси для «keyraces_app», «keyraces_db» (PostgreSQL) та «keyraces_redis», налаштовано мережеву взаємодію між ними, визначено порти та томи для збереження даних. Важливим аспектом конфігурації через «docker-compose.yml» стало визначення змінних середовища, зокрема рядків підключення до бази даних та Redis, що дозволило відокремити конфігурацію від коду додатку. Це забезпечило гнучкість налаштувань для різних середовищ (розробка, тестування, продуктив).

На цьому етапі також було додано початковий набір NuGet-пакетів, необхідних для базової функціональності, таких як «Microsoft.AspNetCore.OpenApi» для інтеграції Swagger/OpenAPI, «Microsoft.EntityFrameworkCore.Npgsql» для роботи з PostgreSQL через Entity Framework Core, та «Microsoft.Extensions.Caching.StackExchangeRedis» для інтеграції з Redis.

Таким чином, завершення етапу ініціалізації проєкту та налаштування середовища розробки дозволило отримати структурований проєктний каркас, інтегрований з системою контролю версій, та налаштоване контейнеризоване середовище, готове для подальшої розробки основних функціональних модулів веб-сервісу KeyRaces.

3.2 Реалізація базової архітектури та моделей даних

Після успішної ініціалізації проєкту та налаштування середовища розробки, наступним ключовим етапом стала реалізація базової архітектури та визначення моделей даних. Цей етап був спрямований на створення міцного фундаменту для бізнес-логіки системи, визначення структури даних та способів взаємодії з ними.

Розробка розпочалася з детального проєктування сутностей предметної області в проєкті «keyraces.Core». Було визначено основні класи, що відображають ключові об'єкти системи. До таких класів належать:

«UserProfile» – представляє профіль користувача, зберігаючи його ідентифікаційні дані, статистику продуктивності (середня швидкість набору, точність), кількість пройдених тренувань та змагань, а також дату реєстрації та останньої активності. Ця сутність пов'язана з користувачем системи автентифікації ASP.NET Core Identity.

«TextSnippet» – описує текстовий фрагмент, що використовується для тренувань або змагань. Кожен фрагмент характеризується унікальним ідентифікатором, назвою, власне текстом, рівнем складності (наприклад, «легкий», «середній», «важкий»), мовою та ознакою, чи був він згенерований автоматично системою.

«TypingSession» – фіксує інформацію про окрему тренувальну сесію користувача. Ця сутність містить посилання на «UserProfile» та «TextSnippet», час початку та завершення сесії, а також розраховані показники, такі як швидкість набору (WPM) та точність.

«Competition» – визначає параметри змагання між користувачами. Включає назву змагання, посилання на «TextSnippet», запланований час початку, фактичний час завершення та поточний статус змагання (наприклад, «очікує», «в процесі», «завершено»).

«CompetitionParticipant» – зв'язуюча сутність, що фіксує участь конкретного «UserProfile» у певному «Competition». Зберігає інформацію про час приєднання до змагання, прогрес набору, фінальний результат (WPM, точність, кількість помилок) та статус участі.

«Achievement» – описує можливі досягнення, які користувач може отримати в системі (наприклад, «Досягнуто 50 WPM», «10 тренувань поспіль»). Кожне досягнення має унікальний ключ, назву, опис та, можливо, іконку.

«UserAchievement» – фіксує факт отримання користувачем певного досягнення та дату його отримання.

При проєктуванні сутностей особлива увага приділялася інкапсуляції бізнес-логіки безпосередньо в класах сутностей, дотримуючись принципів Domain-Driven Design. Наприклад, сутність «UserProfile» містила методи для оновлення статистики користувача після завершення тренування або змагання, автоматично перераховуючи середні показники. Приклад методу оновлення статистики в «UserProfile»:

```
public void UpdateStatistics(double wpm, double accuracy, bool isRace)
```

```
{
    double totalRacesAndPractices = TotalRaces + TotalPractices;

    if (totalRacesAndPractices > 0)
    {
        AverageWPM = ((AverageWPM * totalRacesAndPractices) + wpm) /
(totalRacesAndPractices + 1);
        AverageAccuracy = ((AverageAccuracy * totalRacesAndPractices)
+ accuracy) / (totalRacesAndPractices + 1);
    }
    else
    {
        AverageWPM = wpm;
        AverageAccuracy = accuracy;
    }

    if (isRace)
        TotalRaces++;
    else
        TotalPractices++;

    UpdatedAt = DateTime.UtcNow;
}
```

Для взаємодії з базою даних було обрано технологію Entity Framework Core (EF Core) як об'єктно-реляційний маппер (ORM). У проєкті «keyraces.Infrastructure» було створено контекст бази даних «AppDbContext», що успадковується від «IdentityDbContext» для інтеграції з ASP.NET Core Identity. У цьому контексті було зареєстровано всі визначені сутності як «DbSet» та налаштовано їх відображення на таблиці бази даних, включаючи визначення первинних та зовнішніх ключів, індексів та зв'язків між сутностями (один-до-одного, один-до-багатьох, багато-до-багатьох). Наприклад, зв'язок багато-до-багатьох між «UserProfile» та «Achievement» було реалізовано через проміжну сутність «UserAchievement».

Для забезпечення доступу до даних з бізнес-логіки було реалізовано патерн «Repository». Для кожної ключової сутності було створено відповідний інтерфейс репозиторію в «keyraces.Core» (наприклад, «IUserProfileRepository», «ITextSnippetRepository») та його конкретну реалізацію в «keyraces.Infrastructure», що використовувала «AppDbContext» для виконання операцій CRUD (Create, Read, Update, Delete) та інших специфічних запитів до бази даних.

Процес зміни схеми бази даних керувався за допомогою механізму міграцій EF Core. Перша міграція, названа «InitialCreate», була згенерована після визначення початкового набору сутностей та їхніх зв'язків. Ця міграція містила C# код, що описував створення відповідних таблиць, колонок, ключів та індексів у базі даних PostgreSQL. Усі наступні зміни в моделях даних (додавання нових полів, сутностей або зміна існуючих) також фіксувалися через створення нових міграцій, що забезпечувало версіонування схеми бази даних та можливість її послідовного оновлення або відкату. Наприклад, додавання поля «Language» до сутності «TextSnippet»

вимагало створення нової міграції, яка б модифікувала відповідну таблицю в базі даних.

Таким чином, на цьому етапі було закладено основи для роботи з даними в системі KeyRaces, визначено структуру сутностей, налаштовано взаємодію з базою даних через EF Core та реалізовано патерн «Repository» для інкапсуляції логіки доступу до даних. Це створило надійну основу для подальшої розробки сервісного шару та API.

3.3 Розробка серверної частини та API

Після визначення моделей даних та налаштування взаємодії з базою даних, основна увага була зосереджена на розробці серверної частини та інтерфейсу програмного додатку (API) веб-сервісу KeyRaces. Цей етап передбачав створення контролерів для обробки HTTP-запитів, реалізацію бізнес-логіки в сервісах, налаштування автентифікації та авторизації, а також забезпечення механізмів валідації даних та обробки помилок.

Розробка API здійснювалася в проєкті «keyraces.Server» з використанням фреймворку ASP.NET Core. Було прийнято рішення про побудову API за принципами REST (Representational State Transfer), що забезпечує стандартизований підхід до взаємодії між клієнтською та серверною частинами через HTTP-методи (GET, POST, PUT, DELETE) та чітко визначені ендпоінти для кожного ресурсу.

Для кожної ключової предметної області було створено окремий API контролер:

«AuthController» було розроблено для управління процесами автентифікації та реєстрації користувачів. Він містив ендпоінти для реєстрації нового користувача («/api/auth/register»), входу існуючого користувача («/api/auth/login»), оновлення JWT-токенів за допомогою refresh-токена («/api/auth/refresh») та виходу користувача з системи («/api/auth/logout»).

«TextSnippetController» було реалізовано для управління текстовими фрагментами. Цей контролер надавав ендпоінти для отримання списку всіх текстових фрагментів з можливістю фільтрації за складністю та мовою («/api/textsnippet»), отримання випадкового текстового фрагменту для тренування («/api/textsnippet/random»), додавання нового текстового фрагменту адміністратором («/api/textsnippet» – POST) та оновлення існуючого («/api/textsnippet/id» - PUT).

«SessionController» було створено для управління тренувальними сесіями користувачів. Він включав ендпоінти для початку нової тренувальної сесії («/api/session/start»), фіксації результатів завершеної сесії («/api/session/complete»), отримання статистики конкретної сесії («/api/session/id») та списку всіх сесій поточного користувача («/api/session/user»).

«CompetitionController» та «ParticipantController» було розроблено для управління змаганнями та учасниками. «CompetitionController» дозволяв

створювати нові змагання, отримувати список активних та завершених змагань. «ParticipantController» обробляв запити на приєднання користувачів до змагань.

«AchievementController» надавав ендпоінти для отримання списку всіх доступних досягнень та списку досягнень, отриманих конкретним користувачем.

«TextGenerationController» було додано для інтеграції з сервісом генерації текстів на основі LLM. Ключовий ендпоінт «/api/textgeneration/generate» приймав параметри (тема, складність, довжина, мова) та повертав згенерований текст.

Автентифікація користувачів в API була реалізована на основі JWT (JSON Web Tokens). При успішному вході користувача «AuthController» генерував пару токенів: короткоживучий access-токен (наприклад, з терміном дії 15-60 хвилин) та довгоживучий refresh-токен (наприклад, з терміном дії 7-30 днів). Access-токен використовувався для авторизації кожного запиту до захищених ендпоінтів API і передавався в HTTP-заголовку «Authorization» з префіксом «Bearer». Refresh-токен зберігався на клієнті (наприклад, у безпечному httpOnly cookie) і використовувався для отримання нової пари токенів без необхідності повторного введення логіна та пароля. Для управління користувачами, ролями та процесом генерації токенів використовувалася бібліотека ASP.NET Core Identity.

Авторизація доступу до певних ендпоінтів API здійснювалася за допомогою атрибутів, таких як «[Authorize]». Наприклад, ендпоінти для створення змагань або додавання текстових фрагментів могли бути доступні лише користувачам з роллю «Адміністратор», що перевірялося за допомогою «[Authorize(Roles = "Admin")]».

Для забезпечення коректності даних, що надходять від клієнта, було впроваджено механізми валідації. На рівні моделей DTO (Data Transfer Objects), що використовуються для передачі даних в запитах, застосовувалися атрибути валідації з простору імен «System.ComponentModel.DataAnnotations» (наприклад, «[Required]», «[StringLength]», «[EmailAddress]»). Додатково, для більш складних сценаріїв валідації, можна було б використовувати бібліотеку FluentValidation. Перед виконанням логіки контролера, стан моделі («ModelState») перевірявся, і у випадку невалідних даних клієнту поверталася відповідь з HTTP-статусом 400 (Bad Request) та списком помилок валідації.

Обробка помилок та виняткових ситуацій була централізована за допомогою спеціального middleware компонента. Цей компонент перехоплював усі необроблені винятки, що виникали під час обробки запиту, логував їх та повертав клієнту стандартизовану відповідь про помилку у форматі JSON. Залежно від типу винятку, встановлювався відповідний HTTP-статус (наприклад, 500 Internal Server Error для непередбачених помилок, 404 Not Found, якщо ресурс не знайдено). Це дозволило уникнути

витоку деталей реалізації та надавати клієнту зрозумілу інформацію про проблему.

Бізнес-логіка, пов'язана з обробкою запитів, була винесена з контролерів у окремі сервіси, розташовані в проєкті «keyraces.Infrastructure» (або «keyraces.Core» для інтерфейсів). Наприклад, «TypingSessionService» інкапсулював логіку створення, завершення та отримання тренувальних сесій, взаємодіючи з відповідними репозиторіями для доступу до даних. Контролери викликали методи цих сервісів, передаючи їм необхідні дані та отримуючи результати для формування HTTP-відповіді. Такий підхід сприяв дотриманню принципу єдиної відповідальності (Single Responsibility Principle) та покращував тестованість коду.

Для взаємодії в реальному часі, зокрема для функціоналу змагань, було інтегровано SignalR. У проєкті «keyraces.Server» було створено «TypingHub», який обробляв підключення клієнтів, дозволяв їм приєднуватися до груп (кімнат змагань) та транслював оновлення про прогрес учасників (поточний WPM, відсоток пройденого тексту) усім клієнтам відповідної групи. Це забезпечувало динамічне відображення перебігу змагання на клієнтській стороні.

3.4 Розробка клієнтського інтерфейсу

Після розробки серверної частини та API, наступним етапом стало створення клієнтського інтерфейсу для взаємодії користувачів з веб-сервісом KeyRaces. Для реалізації клієнтської частини було обрано технологію Blazor Server. Цей вибір був зумовлений можливістю розробки інтерактивних веб-інтерфейсів з використанням мови C# та платформи .NET, що дозволило зберегти єдиний технологічний стек для всього додатку та спростити взаємодію між клієнтською та серверною логікою. Blazor Server забезпечує рендеринг інтерфейсу на сервері та передачу оновлень DOM клієнту через з'єднання SignalR, що мінімізує кількість JavaScript коду, необхідного на клієнті.

Розробка клієнтського інтерфейсу розпочалася зі створення основних сторінок (Razor компоненти з директивою «@page») та навігаційних елементів. Було створено наступні ключові сторінки:

Сторінка «Index.razor» – головна сторінка сайту, що надає загальну інформацію про сервіс та можливості для швидкого старту тренування.

Сторінка «Practice.razor» – призначена для індивідуальних тренувань. Цей компонент містив логіку для завантаження текстового фрагменту, відстеження введення тексту користувачем в реальному часі, розрахунку швидкості набору (WPM), точності та відображення помилок.

Сторінка «Competitions.razor» – відображала список доступних змагань, дозволяла користувачам створювати нові лобі (якщо є відповідні права) та приєднуватися до існуючих.

Сторінка «CompetitionLobby.razor» – представляла лобі конкретного змагання, де учасники могли бачити список інших гравців, їхній статус

готовності, а також отримувати оновлення про перебіг змагання в реальному часі.

Сторінка «Auth.razor» – містила компоненти для реєстрації та входу користувачів.

Сторінка «Profile.razor» – відображала профіль користувача, його статистику, список досягнень та історію тренувань/змагань.

Сторінка «AdminPanel.razor» – надавала адміністраторам інструменти для управління користувачами, текстовими фрагментами та іншими аспектами системи.

Для кожної сторінки було розроблено відповідну розмітку HTML з використанням синтаксису Razor та стилізацію за допомогою CSS. Для забезпечення адаптивного дизайну та швидкого прототипування інтерфейсу використовувався CSS-фреймворк Bootstrap 5. Було створено кастомні CSS-файли для специфічних стилів компонентів та загального вигляду сайту, забезпечуючи узгодженість дизайну на всіх сторінках.

Ключовим компонентом клієнтської частини стала сторінка «Practice.razor». Її реалізація включала:

Механізм завантаження текстового фрагменту з API.

Обробку подій клавіатури («@onkeydown», «@onkeypress») для відстеження введення символів користувачем.

Порівняння введеного тексту з еталонним для визначення правильності набору та підсвічування помилок. Наприклад, кожен символ еталонного тексту відображався як окремий «», клас якого динамічно змінювався залежно від введення користувача («correct», «incorrect», «current»).

Розрахунок метрик WPM та точності в реальному часі. Швидкість розраховувалася за формулою: «WPM = (кількість_правильних_слів / час_в_хвилинах)», де стандартне слово приймалося за 5 символів. Точність визначалася як відсоток правильно набраних символів від загальної кількості введених.

Візуалізацію прогресу, наприклад, за допомогою прогрес-бару, що показував відсоток пройденого тексту.

Приклад обробки введення в «Practice.razor»:

```
private async Task HandleKeyPress(KeyboardEventArgs e)
{
    if (isPracticeActive && !isCompleted && currentPosition < 
practiceText.Length)
    {
        char typedChar = e.Key[0]; // Припускаємо, що e.Key 
містить один символ
        char expectedChar = practiceText[currentPosition];

        // Логіка порівняння, оновлення typedText, підрахунку 
помилок
        // ...

        currentPosition++;
        typedTextBuilder.Append(typedChar); // Використання 
StringBuilder для ефективності
```

```
            // Оновлення відображення та метрик
            await InvokeAsync(StateHasChanged);

            if (currentPosition == practiceText.Length)
            {
                await CompletePractice();
            }
        }
    }
```

Для взаємодії з серверним API було створено набір сервісів на клієнтській стороні. Ці сервіси інкапсулювали логіку HTTP-запитів до відповідних ендпоінтів API з використанням «HttpClient». Наприклад, «AuthService» містив методи для реєстрації, входу та виходу користувача, «TextSnippetService» – для отримання текстових фрагментів. Ці сервіси реєструвалися в контейнері залежностей Blazor та інжектувалися в компоненти сторінок.

Для управління станом автентифікації користувача на клієнті було реалізовано «AuthenticationStateProvider». Цей провайдер відстежував наявність та валідність JWT-токена (зберігався, наприклад, у «localStorage» браузера через JS Interop, або у безпечнішому сховищі) та надавав інформацію про поточного користувача іншим компонентам.

Для сторінок змагань («Competitions.razor» та «CompetitionLobby.razor») було реалізовано інтеграцію з SignalR хабом «TypingHub». При вході на сторінку лобі змагання встановлювалося з'єднання з хабом, клієнт приєднувався до відповідної групи змагання. Компонент підписувався на серверні методи хабу для отримання оновлень про стан лобі, список учасників, їхній прогрес та результати. Також клієнт викликав серверні методи хабу для відправки власного прогресу або зміни статусу готовності.

Особливу увагу було приділено користувацькому досвіду (UX). Інтерфейс проєктувався з урахуванням інтуїтивності навігації, чіткого відображення інформації та мінімізації затримок при взаємодії. Наприклад, при наборі тексту забезпечувався миттєвий зворотний зв'язок щодо правильності введення.

3.5 Інтеграція змагань в реальному часі з використанням SignalR

Для реалізації функціоналу змагань в реальному часі, що передбачає миттєве оновлення стану гри та взаємодію між учасниками, було інтегровано бібліотеку ASP.NET Core SignalR. Ця технологія забезпечує двонаправлений зв'язок між сервером та підключеними клієнтами.

Призначення SignalR та архітектура взаємодії в реальному часі.

SignalR використовується для передачі даних в реальному часі між сервером та клієнтами, що є необхідним для відображення актуального стану лобі, прогресу гравців та обміну повідомленнями в чаті.

Архітектура взаємодії включає наступні компоненти:

Клієнт («CompetitionLobby.razor»): Blazor Server компонент, що ініціює та підтримує з'єднання з SignalR хабом. Надсилає на сервер дії користувача (зміна статусу готовності, прогрес друку, повідомлення чату) та отримує оновлення стану лобі та гри.

SignalR Хаб («TypingHub.cs»): Серверний компонент, що успадковується від «Microsoft.AspNetCore.SignalR.Hub». Діє як центральний вузол для обробки повідомлень від клієнтів, виклику методів сервісу стану лобі та розсилки оновлень підключеним клієнтам у відповідних групах.

Сервіс стану лобі («RedisCompetitionLobbyService.cs»): Реалізація інтерфейсу «ICompetitionLobbyService». Відповідає за управління персистентним станом лобі (список учасників, їхній статус, обраний текст, результати гри). Використовує «IDistributedCache» (сконфігурований для Redis) для зберігання та синхронізації даних.

Серверна реалізація: «TypingHub.cs».

Клас «TypingHub» є основним елементом серверної логіки SignalR.

Базовий клас: «Microsoft.AspNetCore.SignalR.Hub».

Авторизація: Застосовано атрибут «[Authorize]», що вимагає автентифікації користувача для взаємодії з методами хабу.

Ідентифікація користувача: Ідентифікатор та ім'я користувача отримуються з «Context.User» (наприклад, «Context.User.FindFirstValue(ClaimTypes.NameIdentifier)» та «Context.User.FindFirstValue(ClaimTypes.Name)»).

Управління з'єднаннями:

«OnConnectedAsync()»: Викликається при встановленні нового клієнтського з'єднання. Використовується для логування («_logger.LogInformation»).

«OnDisconnectedAsync(Exception exception)»: Викликається при розриві з'єднання. Логує подію та може містити логіку для автоматичного виходу гравця з лобі.

Методи хабу:

«JoinLobby(string lobbyId)»: Додає з'єднання клієнта до групи SignalR, що відповідає «lobbyId» («await Groups.AddToGroupAsync(Context.ConnectionId, $«lobby-{lobbyId}»»)»). Викликає «_lobbyService.JoinLobbyAsync» для оновлення стану лобі в Redis. Сповіщає інших учасників групи про нового гравця («await Clients.Group($«lobby-{lobbyId}»»).SendAsync(«UserJoined», userName)»).

«LeaveLobby(string lobbyId)»: Видаляє з'єднання клієнта з групи («await Groups.RemoveFromGroupAsync(Context.ConnectionId, $«lobby-{lobbyId}»»)»). Викликає «_lobbyService.LeaveLobbyAsync». Сповіщає групу про вихід гравця («await Clients.Group($«lobby-{lobbyId}»»).SendAsync(«UserLeft», userName)»).

«UpdateReadyStatus(string lobbyId, bool isReady)»: Обробляє зміну статусу готовності гравця. Викликає

«_lobbyService.UpdateReadyStatusAsync». Сповіщає групу про зміну, надсилаючи оновлений об'єкт лобі («await Clients.Group($«lobby-{lobbyId}»).SendAsync(«ReadyStatusChanged», updatedLobby)»).

«StartGame(string lobbyId, int textSnippetId)»: Ініціюється хостом лобі. Перевіряє, чи є поточний користувач хостом («lobby.HostId == userId»). Викликає «_lobbyService.StartGameAsync». Сповіщає групу про початок гри («await Clients.Group($«lobby-{lobbyId}»).SendAsync(«GameStarted»)»).

«UpdateProgress(string lobbyId, int progress, int wpm, double accuracy)»: Приймає дані про поточний прогрес гравця. Викликає «_lobbyService.UpdatePlayerProgressAsync». Транслює ці дані іншим учасникам групи («await Clients.Group($«lobby-{lobbyId}»).SendAsync(«PlayerProgressUpdated», userId, progress, wpm, accuracy)»).

«FinishGame(string lobbyId, int finalWpm, double finalAccuracy)»: Фіксує завершення гри гравцем. Викликає «_lobbyService.CompleteGameAsync». Сповіщає групу про завершення гравцем («await Clients.Group($«lobby-{lobbyId}»).SendAsync(«PlayerFinished», userId, finalWpm, finalAccuracy)»). Якщо всі гравці завершили, надсилаються фінальні результати («await Clients.Group($«lobby-{lobbyId}»).SendAsync(«AllPlayersFinished», lobby)» або «await Clients.Group($«lobby-{lobbyId}»).SendAsync(«GameResults», results)»).

«SendChatMessage(string lobbyId, string message)»: Обробляє надсилання повідомлення в чат лобі. Викликає «_lobbyService.AddChatMessageAsync». Розсилає повідомлення («ChatMessage») всім учасникам групи («await Clients.Group($«lobby-{lobbyId}»).SendAsync(«ReceiveChatMessage», chatMessage)»).

Управління станом лобі: «RedisCompetitionLobbyService.cs».

Сервіс «RedisCompetitionLobbyService» відповідає за персистентне зберігання та управління станом об'єктів «CompetitionLobby».

Використання «IDistributedCache»: Взаємодіє з Redis (або іншим сконфігурованим розподіленим кешем) для операцій читання/запису стану лобі.

Зберігання даних: Об'єкти «CompetitionLobby» серіалізуються в JSON рядок за допомогою «System.Text.Json.JsonSerializer» перед збереженням в Redis («await _cache.SetStringAsync(key, json, options)»). При читанні відбувається десеріалізація («JsonSerializer.Deserialize<CompetitionLobby>(json)»).

Ключі Redis: Використовуються префікси для ключів, наприклад, «LOBBY_PREFIX = «lobby:»» (для окремих лобі) та «ACTIVE_LOBBIES_KEY = «active_lobbies»» (для списку активних лобі).

Час життя записів: «DistributedCacheEntryOptions» використовується для встановлення терміну дії записів у кеші (наприклад, «AbsoluteExpirationRelativeToNow = TimeSpan.FromHours(LOBBY_EXPIRY_HOURS)»).

Основні операції:

«CreateLobbyAsync»: Створює новий об'єкт «CompetitionLobby», серіалізує та зберігає його в Redis, додає ID лобі до списку активних лобі.

«GetLobbyAsync»: Отримує серіалізований об'єкт лобі з Redis за ID та десеріалізує його.

«UpdateLobbyAsync»: Серіалізує оновлений об'єкт «CompetitionLobby» та перезаписує його в Redis.

«DeleteLobbyAsync»: Видаляє запис лобі з Redis та з списку активних лобі.

Інші методи («JoinLobbyAsync», «LeaveLobbyAsync», «UpdateReadyStatusAsync» тощо) отримують об'єкт лобі, модифікують його та викликають «UpdateLobbyAsync» для збереження змін.

Клієнтська інтеграція: «CompetitionLobby.razor».

Компонент «CompetitionLobby.razor» на стороні клієнта (Blazor Server) управляє взаємодією з «TypingHub».

Ініціалізація «HubConnection»:

Використовується «HubConnectionBuilder» для конфігурації з'єднання.

URL хабу: «NavigationManager.ToAbsoluteUri(«/typingHub»)».

Токен доступу: JWT-токен отримується з «localStorage» (через «IJSRuntime JS.InvokeAsync<string>(«localStorage.getItem», «auth_token»)») та надається через «options.AccessTokenProvider».

Автоматичне перепідключення: «WithAutomaticReconnect()» налаштовує автоматичні спроби відновлення з'єднання.

Встановлення з'єднання: «await hubConnection.StartAsync()» в методі «OnInitializedAsync» або «SetupSignalR». Після успішного підключення викликається «hubConnection.SendAsync(«JoinLobby», LobbyId)».

Підписка на серверні події: Методи «hubConnection.On<T>()» використовуються для реєстрації обробників повідомлень від сервера. Приклади:

«hubConnection.On<Core.Models.CompetitionLobby>(«ReadyStatusChanged», (updatedLobby) => { lobby = updatedLobby; StateHasChanged(); });»

«hubConnection.On<string, int, int, double>(«PlayerProgressUpdated», (userId, progress, wpm, accuracy) => { /* оновлення UI для гравця */ StateHasChanged(); });»

«hubConnection.On<ChatMessage>(«ReceiveChatMessage», (message) => { chatMessages.Add(message); StateHasChanged(); /* логіка прокрутки чату */ });»

Інші обробники: «UserJoined», «UserLeft», «GameStarted», «PlayerFinished», «AllPlayersFinished», «LobbyUpdated».

Оновлення стану компонента та UI: Змінні компонента (наприклад, «lobby», «chatMessages») оновлюються отриманими даними, а «await InvokeAsync(StateHasChanged)» викликається для перерендеру інтерфейсу.

Виклик методів хабу з клієнта: Методи «hubConnection.SendAsync(«MethodName», ...args)» використовуються для надсилання запитів на сервер. Приклади:

«await hubConnection.SendAsync(«UpdateReadyStatus», LobbyId, newReadyStatus);»

«await hubConnection.SendAsync(«SendChatMessage», LobbyId, newMessage);»

Інші виклики: «StartGame», «UpdateProgress», «FinishGame».

Управління життєвим циклом:

Компонент реалізує «IAsyncDisposable». Метод «DisposeAsync()» викликає «hubConnection.SendAsync(«LeaveLobby», LobbyId)» (опціонально, для негайного виходу) та «await hubConnection.DisposeAsync()» для коректного закриття з'єднання та звільнення ресурсів.

Синхронізація даних та ігровий цикл.

Процес синхронізації даних під час змагання:

Початок гри: Хост ініціює старт. Клієнт викликає «StartGame» на хабі. «TypingHub» викликає «_lobbyService.StartGameAsync», який оновлює стан лобі в Redis (встановлює «LobbyStatus.InGame», «TextSnippetId», «StartedAt»). Хаб розсилає повідомлення «GameStarted» (або «LobbyUpdated») всім клієнтам у групі. Клієнти переходять в ігровий режим, завантажують текст змагання на основі «lobby.TextSnippetId».

Оновлення прогресу: Під час гри клієнт («CompetitionLobby.razor» у співпраці з «lobbyInterop.js») відстежує введення тексту, розраховує WPM, точність та відсоток пройденого тексту. Ці дані періодично (наприклад, кожну секунду або при значній зміні, як реалізовано в «HandleKeyDown» та «UpdateStats») надсилаються на сервер викликом «UpdateProgress» на хабі. «TypingHub» передає ці дані в «_lobbyService.UpdatePlayerProgressAsync» для оновлення стану гравця в Redis. Потім хаб транслює ці оновлення всім іншим учасникам лобі через «PlayerProgressUpdated».

Завершення гри гравцем: Коли гравець завершує набір тексту (визначається на клієнті, наприклад, в «CheckGameCompletion»), клієнт викликає «FinishGame» на хабі, передаючи фінальні WPM та точність. «TypingHub» викликає «_lobbyService.CompleteGameAsync» (або «PlayerFinishedAsync»), який встановлює «HasFinished = true», «FinalWPM», «FinalAccuracy», «FinishedAt», «Position» для гравця в Redis. Хаб сповіщає групу через «PlayerFinished» та «LobbyUpdated».

Завершення гри всіма гравцями: Коли «_lobbyService» визначає, що всі гравці в лобі завершили гру («lobby.Players.All(p => p.HasFinished)»), статус лобі змінюється на «LobbyStatus.Finished». «TypingHub» може розіслати повідомлення «AllPlayersFinished» з фінальними результатами або оновлений об'єкт «LobbyUpdated», що сигналізує про завершення змагання.

Обмін повідомленнями в чаті: Клієнт надсилає повідомлення через «SendChatMessage» на хабі. «TypingHub» викликає «_lobbyService.AddChatMessageAsync» для збереження повідомлення в

списку «ChatMessages» об'єкта лобі в Redis. Хаб розсилає отримане «ChatMessage» всім клієнтам у групі через «ReceiveChatMessage».

3.6 Генерація текстових фрагментів за допомогою LLM

Для розширення бази текстових фрагментів та надання користувачам більш різноманітного контенту для тренувань, в проєкт KeyRaces було інтегровано функціонал генерації текстів за допомогою великої мовної моделі (LLM). Це дозволяє створювати унікальні тексти на задані теми, різної складності та довжини, на додаток до попередньо завантажених статичних фрагментів.

Основним компонентом, що відповідає за генерацію тексту, є сервіс «LocalLLMTextGenerationService», який реалізує інтерфейс «ITextGenerationService». Цей інтерфейс визначає метод «GenerateTextAsync(string topic, string difficulty, int length, string language)», що приймає тему, бажану складність, довжину та мову тексту. «LocalLLMTextGenerationService» взаємодіє з локально розгорнутою моделлю LLM через API, за замовчуванням використовується Ollama з моделлю «llama2», доступною за адресою «http://localhost:11434/api/generate» Сервіс також включає механізми для перевірки доступності сервера LLM та генерації резервного тексту у випадку недоступності або помилки.

Процес генерації тексту починається з виклику ендпоінту «/api/TextGeneration/generate» в «TextGenerationController». Контролер приймає об'єкт «TextGenerationRequest», що містить параметри генерації. Перед безпосередньою генерацією, «LocalLLMTextGenerationService» виконує перевірку доступності сервера Ollama, звертаючись до його кореневого ендпоінту. Якщо сервер недоступний, використовується метод «GenerateFallbackText».

Якщо тема для генерації не вказана користувачем, сервіс обирає випадкову тему з попередньо визначеного списку для вказаної мови за допомогою методу «GetRandomTopic(language)». Ключовим етапом є формування запиту (промпту) до LLM, що виконується методом «CreatePrompt». Цей метод генерує текстовий запит, що включає інструкції для моделі: мову тексту, тему, рівень складності, приблизну довжину, а також низку обмежень, таких як заборона використання маркерів, списків, заголовків, вступних чи заключних фраз, згадок про «тренування друку», спеціальних символів (крім стандартної пунктуації) та розмітки. Для кожної мови (англійська, українська, російська) формується специфічний промпт, наприклад, для української мови додається вимога використовувати символи «є», «і», «ї» та «ґ». Структура промпту виглядає як

«### ІНСТРУКЦІЯ ### ... ### ВІДПОВІДЬ ###».

Перед кожним запитом до LLM виконується спроба очищення контексту моделі за допомогою методу «ResetOllamaContext». Цей метод надсилає спеціальний запит до Ollama з параметром «num_ctx = 0», щоб мінімізувати вплив попередніх взаємодій на поточну генерацію. Взаємодія з

API Ollama відбувається в методі «GenerateTextWithOllama». Формується JSON-запит, що включає назву моделі, згенерований промпт, параметр «stream = false» (для отримання повної відповіді одразу), налаштування температури, максимальної кількості токенів та інші опції, такі як «seed» для відтворюваності та «stop»-послідовності для обмеження генерації. Запит надсилається методом POST за допомогою «HttpClient». Отримана JSON-відповідь парситься для вилучення згенерованого тексту з поля «response». Передбачено до трьох спроб генерації, якщо перші спроби не дали результату або згенерований текст не пройшов валідацію.

Після отримання сирого тексту від LLM, він проходить етап постобробки в методі «ProcessGeneratedText». Цей метод послідовно викликає декілька функцій очищення та валідації. «CleanGeneratedText» видаляє стандартні артефакти LLM, такі як маркери «###ВІДПОВІДЬ###», типові вступні та заключні фрази, а також зайві пробіли. «ExtractMainText» намагається виділити основний змістовний текст, особливо якщо модель повернула декілька абзаців або зайвий контент, розділяючи текст на абзаци або рядки. Далі, метод «PurifyText» виконує більш глибоке очищення: «RemoveNonsensePatterns» використовує регулярні вирази для видалення поширених небажаних патернів (наприклад, порожніх квадратних дужок, конструкцій Markdown, HTML-тегів); «RemoveRepeatingFragments» виявляє та видаляє надмірно повторювані фрагменти тексту; «HandleMixedLanguagesCarefully» розділяє текст на речення та намагається відфільтрувати речення, що не відповідають цільовій мові, аналізуючи набори символів за допомогою функцій «IsRussianSentence», «IsUkrainianSentence», «IsEnglishSentence»; «FinalCleanup» нормалізує пробільні символи та видаляє короткі слова з інших мов (наприклад, короткі англійські слова в російському тексті).

Валідація очищеного тексту відбувається в методі «IsValidText». Перевіряється мінімальна довжина тексту (наприклад, більше 100 символів) та співвідношення літер (відповідних для цільової мови) до загальної кількості символів. Якщо це співвідношення занадто низьке (наприклад, менше 0.3), текст вважається невалідним (ймовірно, складається переважно з розділових знаків або символів). На завершення, метод «AdjustTextLength» коригує довжину тексту до цільового значення. Якщо текст довший, він обрізається по межі слова. Якщо коротший, до нього додається загальне речення (специфічне для мови) або повторюється останнє речення до досягнення приблизно потрібної довжини, після чого текст знову обрізається.

У випадку, якщо LLM недоступний або всі спроби генерації та очищення не дали валідного результату, активується механізм резервного копіювання «GenerateFallbackText». Цей метод надає попередньо визначені тексти, класифіковані за складністю (легкий, середній, важкий) для кожної підтримуваної мови. Текст обирається на основі запитаної складності та, якщо можливо, теми. Якщо відповідної теми немає, обирається випадковий

текст із потрібної категорії складності. Обраний резервний текст також проходить через «AdjustTextLength».

Успішно згенеровані та оброблені тексти зберігаються в базі даних як сутності «TextSnippet». Для таких фрагментів встановлюється прапорець «IsGenerated = true», а назва, складність та мова встановлюються відповідно до початкового запиту. Для діагностики з'єднання з сервером LLM, «TextGenerationController» надає ендпоінт «HttpGet(«check-connection»)», який перевіряє доступність Ollama та список доступних моделей.

3.7 Тестування, оптимізація та управління розробкою

Етапи тестування та оптимізації були невід'ємною частиною розробки веб-сервісу KeyRaces, спрямовані на забезпечення стабільності, продуктивності та надійності системи. Управління вихідним кодом проєкту, відстеження змін та координація розробки здійснювалися за допомогою системи контролю версій Git та платформи GitHub.

Тестування проєкту охоплювало декілька рівнів.

Модульне тестування (Unit Testing) було зосереджено на перевірці коректності окремих компонентів та логіки в ізольованому середовищі. Для цього використовувався фреймворк xUnit. Тести були написані для ключових класів в проєкті «keyraces.Core.Tests». Кожна нова функціональність або виправлення помилки супроводжувалися написанням або оновленням відповідних модульних тестів. Зміни в коді, включаючи тести, фіксувалися в локальному репозиторії Git та регулярно надсилалися до віддаленого репозиторію на GitHub, що забезпечувало історію змін та можливість повернення до попередніх версій.

Інтеграційне тестування мало на меті перевірку взаємодії між різними компонентами системи. Використання гілок (branches) в Git на GitHub дозволяло ізолювати розробку нових функцій або експериментальних змін. Перед злиттям (merge) таких гілок в основну гілку розробки (наприклад, «main» або «develop»), проводилося тестування для перевірки сумісності та коректності інтеграції.

Тестування користувацького інтерфейсу (UI Testing) проводилося переважно вручну. Виявлені помилки або недоліки реєструвалися як завдання (issues) на платформі GitHub, що дозволяло відстежувати їхній статус та призначати відповідальних за виправлення.

Тестування функціоналу реального часу, пов'язаного з SignalR, також координувалося через завдання на GitHub, особливо якщо виявлялися складні проблеми взаємодії, що вимагали спільного обговорення.

Оптимізація продуктивності була спрямована на декілька аспектів.

Оптимізація запитів до бази даних.

Оптимізація роботи з кешем.

Оптимізація клієнтської частини (Blazor Server).

Оптимізація роботи SignalR.

Оптимізація генерації тексту LLM.

Зміни, пов'язані з оптимізацією, також проходили через систему контролю версій. Це дозволяло аналізувати вплив оптимізаційних заходів на продуктивність, порівнюючи версії коду до та після змін. Коментарі до комітів на GitHub часто містили інформацію про причини змін та очікувані покращення.

Висновки за розділом

У даному розділі було детально розглянуто процес розробки веб-сервісу для тренування швидкісного друку KeyRaces. Розробка охоплювала низку послідовних етапів, починаючи від ініціалізації проєкту та закінчуючи його тестуванням та оптимізацією, з використанням сучасних технологій та підходів.

На початковому етапі було визначено основні технології: платформа .NET для серверної частини та Blazor Server для клієнтського інтерфейсу, що дозволило створити єдину технологічну екосистему на мові C#. Було налаштовано середовище розробки, включаючи систему контролю версій Git з використанням платформи GitHub для управління вихідним кодом та спільної роботи.

Проєктування архітектури системи базувалося на принципах чистої архітектури (Clean Architecture), з чітким розділенням відповідальності між шарами: ядро («Core») з бізнес-логікою та сутностями, інфраструктурний шар («Infrastructure») з реалізацією доступу до даних (Entity Framework Core, репозиторії) та інтеграцією зовнішніх сервісів (Redis, LLM), та презентаційний шар («Server») з API контролерами, SignalR хабами та Blazor компонентами. Було визначено ключові сутності даних, такі як «UserProfile», «TextSnippet», «TypingSession», «Competition», «Achievement», та їхні взаємозв'язки, що лягли в основу схеми бази даних PostgreSQL.

Розробка серверної частини включала створення RESTful API ендпоінтів для управління користувачами, автентифікації (на основі JWT), управління текстовими фрагментами, статистикою тренувань та змаганнями. Було реалізовано сервісний шар для інкапсуляції бізнес-логіки та репозиторії для абстрагування доступу до даних.

Клієнтський інтерфейс було створено за допомогою технології Blazor Server. Розроблено набір інтерактивних Razor-компонентів для основних сторінок сервісу: головна, тренування, змагання, профіль користувача, панель адміністратора. Взаємодія з API здійснювалася через клієнтські сервіси, а управління станом автентифікації – через кастомний «AuthenticationStateProvider».

Ключовим функціоналом стала інтеграція змагань в реальному часі, реалізована за допомогою ASP.NET Core SignalR. Було створено «TypingHub» для обробки взаємодій гравців, таких як приєднання до лобі, оновлення статусу готовності, передача прогресу друку та обмін повідомленнями в чаті. Стан лобі змагань зберігався та синхронізувався за

допомогою розподіленого кешу Redis, що забезпечило швидкий доступ та узгодженість даних.

Для розширення контенту було інтегровано функціонал генерації текстових фрагментів за допомогою великої мовної моделі (LLM), зокрема, через локально розгорнутий сервер Ollama. Розроблено сервіс, що формує промпти, взаємодіє з LLM API, виконує постобробку та валідацію згенерованого тексту, а також має механізм резервного копіювання.

Завершальним етапом розробки стали тестування та оптимізація. Проводилося модульне тестування ключових компонентів за допомогою xUnit, інтеграційне тестування взаємодії компонентів та ручне тестування користувацького інтерфейсу. Оптимізація включала покращення запитів до бази даних, ефективне використання кешу, оптимізацію Blazor компонентів та роботи SignalR, а також логіки генерації текстів LLM. Управління процесом розробки, включаючи відстеження завдань та версіонування коду, здійснювалося за допомогою GitHub.

4. ТЕСТУВАННЯ ФУНКЦІОНАЛЬНОСТІ РОЗРОБЛЕНОГО ПРОЄКТУ

4.1 Основний інтерфейс та навігаційні можливості

При першому вході на веб-сервіс користувач потрапляє на головну сторінку. Ця сторінка, реалізована компонентом «Index.razor», слугує інформаційним та навігаційним центром. Візуально вона представляє логотип сервісу, розташований у верхній частині, заголовок з назвою сервісу та короткий опис його призначення. Нижче розташовані картки, що ведуть до основних функціональних блоків: «Практика», «Тренування», «Змагання» та «Профіль», кожна з яких містить іконку, назву розділу, короткий опис та кнопку для переходу.

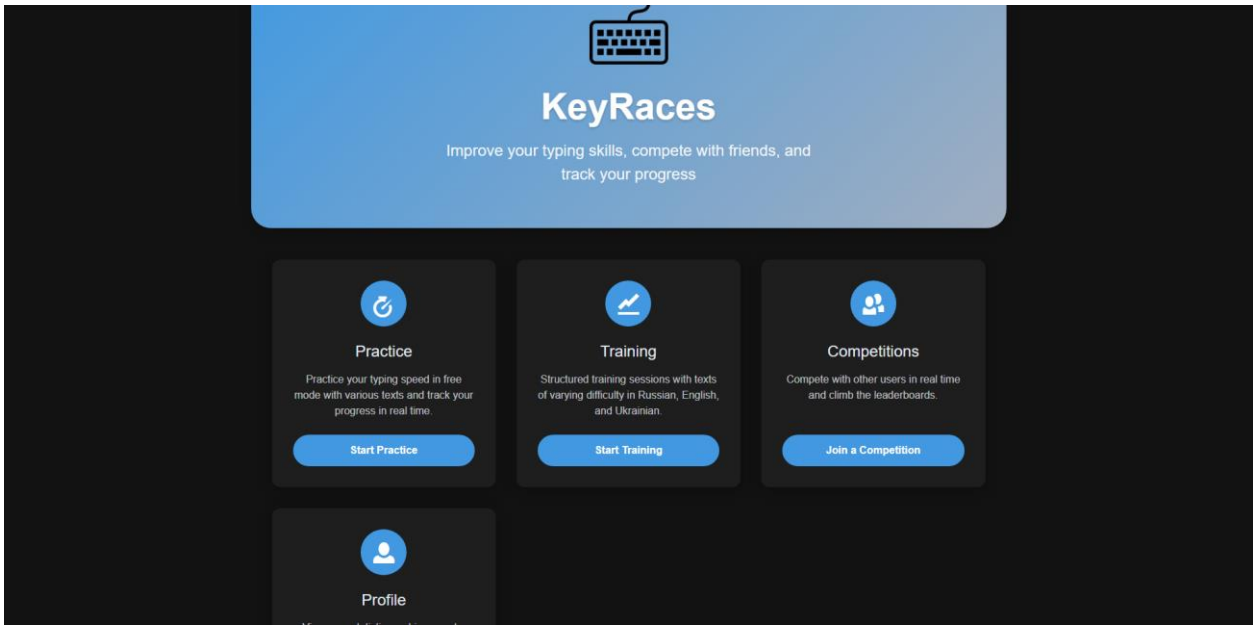

Рисунок 4.1 – Головна сторінка сайту

Основний навігаційний механізм реалізований за допомогою навігаційного меню, яке є компонентом «NavMenu.razor». Воно закріплене у верхній частині кожної сторінки в межах загального макету «MainLayout.razor» і залишається видимим при прокручуванні. Зліва в меню розташований логотип, який також є посиланням на головну сторінку. Центральну частину займають посилання на ключові розділи: «Тренування», «Практика», «Змагання».

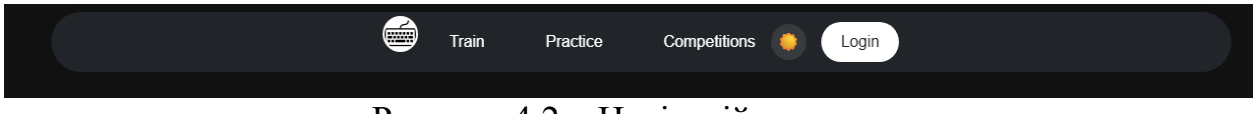

Рисунок 4.2 – Навігаційне меню

Праворуч у навігаційному меню знаходиться блок взаємодії з користувачем. Для неавтентифікованого користувача тут відображається кнопка «Login», що веде на сторінку автентифікації («Auth.razor»).

Після успішної автентифікації замість кнопки «Login» відображається ім'я користувача та випадаюче меню. Це меню містить посилання на «Профіль» («Profile/Profile.razor»), а також, за наявності відповідних прав, посилання «Admin Panel» («AdminPanel.razor») та пункт «Logout» для виходу з системи.

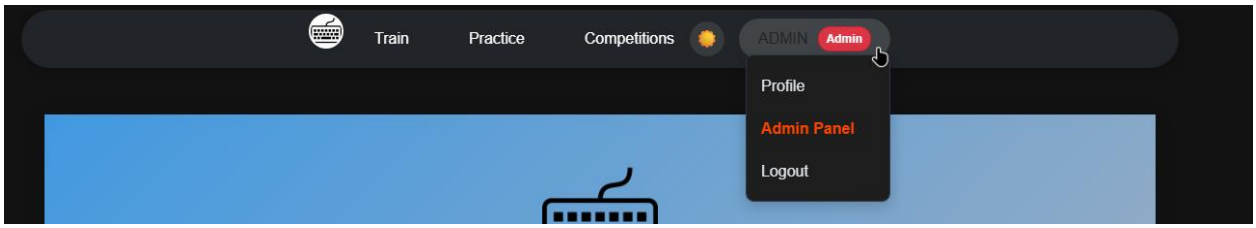

Рисунок 4.3 – Навігаційне меню користувача

Структура меню є стандартною для веб-додатків, що сприяє швидкій адаптації користувачів. Перевірка показала, що всі посилання в навігаційному меню є активними та коректно перенаправляють користувача до відповідних розділів сервісу. Наприклад, клік на пункт «Змагання» переводить користувача на сторінку «Competitions.razor», де відображається список доступних змагань.

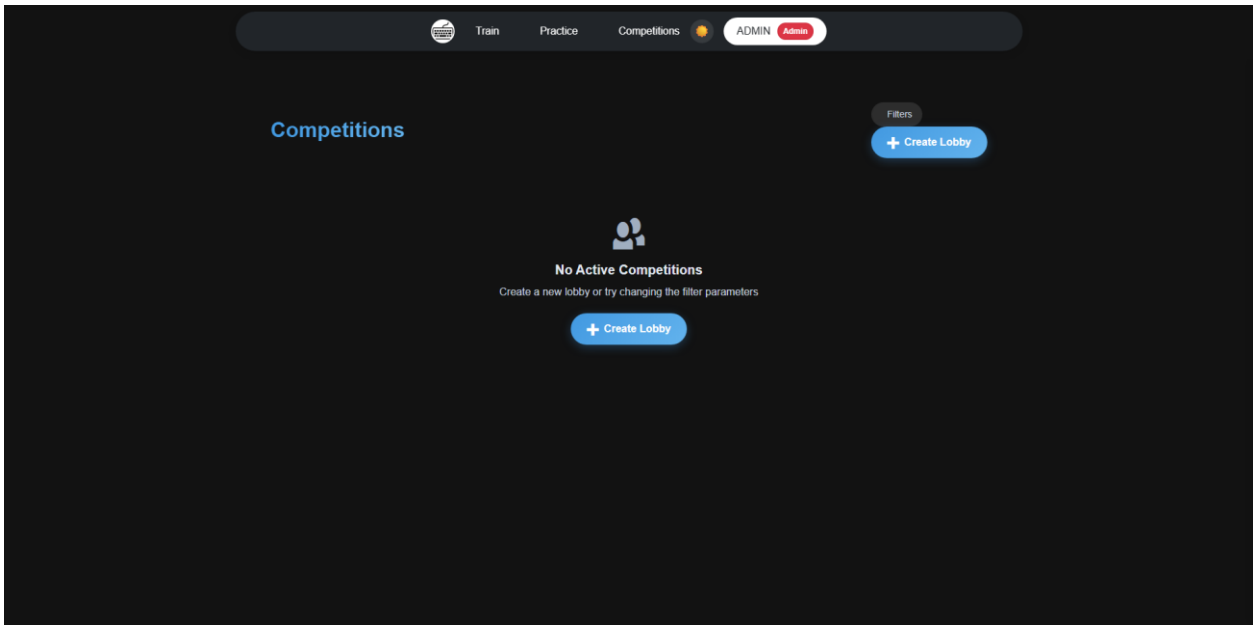

Рисунок 4.4 – Сторінка змагань без існуючих лоббі

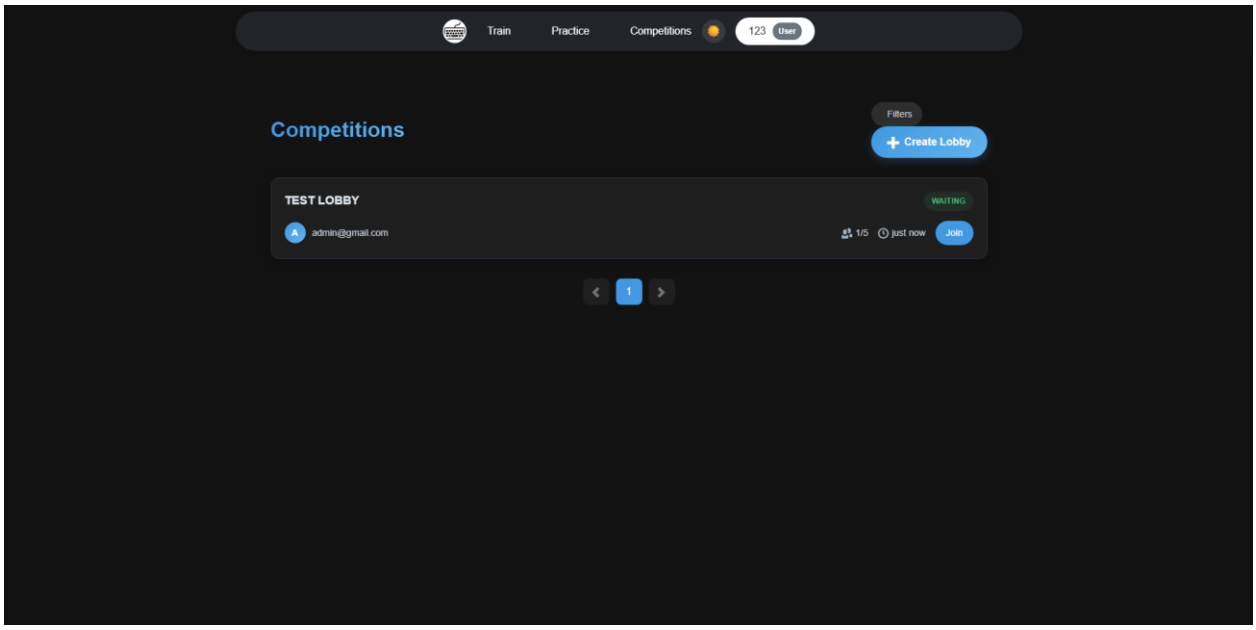

Рисунок 4.5 – Вигляд сторінки змагань з ракурсу іншого користувача та створеним лоббі

Доступ до основних функціональних блоків сервісу здійснюється через чітко визначені навігаційні шляхи. Інтерфейс надає візуальні підказки, такі як зміна вигляду посилань при наведенні курсору або виділення активного пункту меню, що полегшує орієнтацію користувача в структурі сайту.

Загальне компонування сторінок та візуальне оформлення, визначене глобальними стилями у файлах «site.css» та змінними темами у «theme.css», забезпечує консистентність відображення на різних сторінках сервісу. Елементи керування, такі як кнопки та поля вводу, мають єдиний стиль, що відповідає обраній темі оформлення.

В ході перевірки встановлено, що основний інтерфейс та навігаційна система веб-сервісу KeyRaces функціонують коректно. Користувач має змогу безперешкодно переходити між усіма ключовими розділами, а структура сайту є логічною та інтуїтивно зрозумілою. Це забезпечує ефективний доступ до заявлених функціональних можливостей сервісу.

4.2 Процес створення та управління обліковим записом користувача

Процес взаємодії з обліковим записом починається на сторінці автентифікації, яка реалізована компонентом «Auth.razor». Ця сторінка доступна за прямими посиланнями «/login» та «/register», а також через кнопку «Login» у навігаційному меню для неавтентифікованих користувачів. Сторінка автентифікації містить дві вкладки: «Login» для входу існуючих користувачів та «Register» для створення нового облікового запису. За замовчуванням активною може бути вкладка «Login».

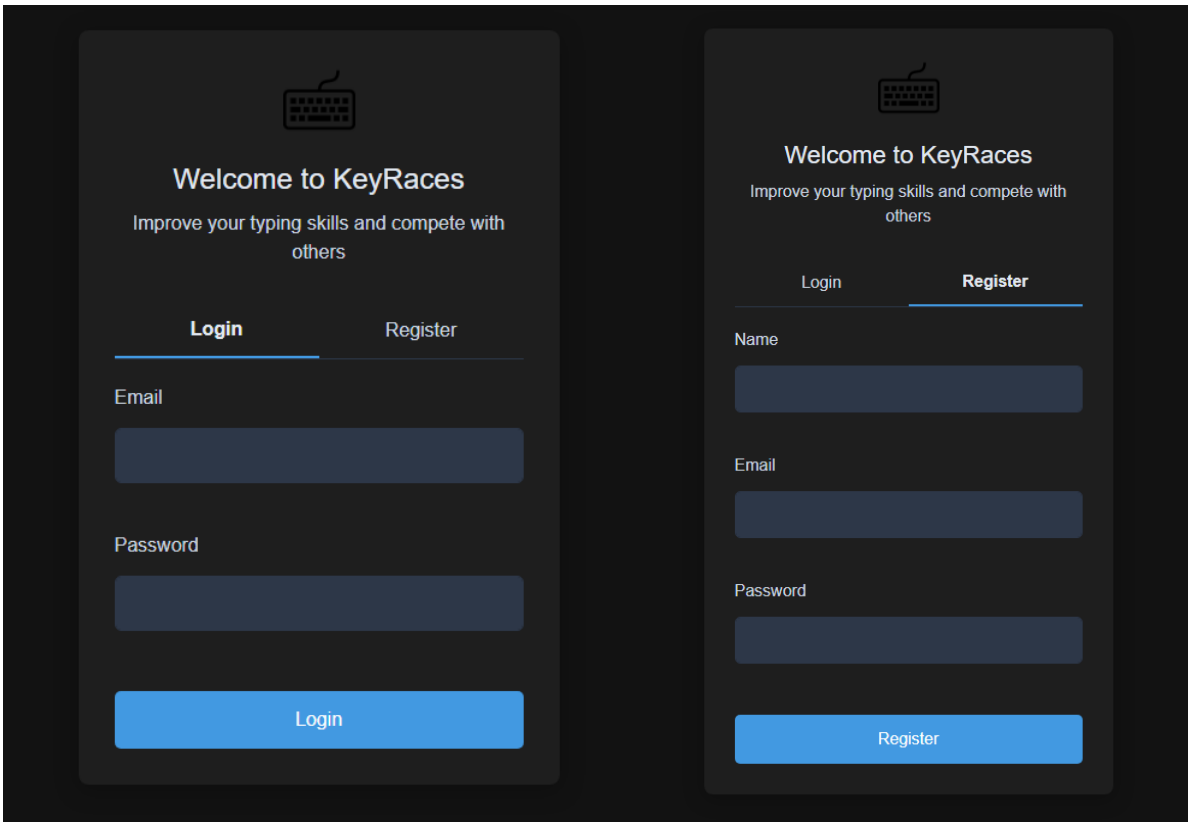

Рисунок 4.6 – Форми логіну та реєстрації користувача

Для створення нового облікового запису користувач переходить на вкладку «Register». Форма реєстрації, керована моделлю «RegisterDto.cs», вимагає введення наступних даних: ім'я користувача («Name»), адреса електронної пошти («Email») та пароль («Password»). Кожне поле має валідацію на стороні клієнта та сервера.

Після заповнення всіх полів та натискання кнопки «Register», дані надсилаються на серверний контролер «AuthController.cs», зокрема на метод «Register». У випадку успішної реєстрації, створюється новий користувач в системі ASP.NET Core Identity, йому призначається роль «User», створюється відповідний профіль користувача («UserProfile») та генеруються токени автентифікації. Користувач автоматично автентифікується та перенаправляється на головну сторінку сервісу або на сторінку, з якої він ініціював процес реєстрації (якщо був параметр «returnUrl»).

Для входу в систему існуючий користувач використовує вкладку «Login» на сторінці «Auth.razor». Форма входу, керована моделлю «LoginDto.cs», вимагає введення адреси електронної пошти («Email») та пароля («Password»).

Після натискання кнопки «Login», дані передаються на метод «Login» контролера «AuthController.cs». Сервер перевіряє відповідність наданих облікових даних. У разі успішної автентифікації, користувач отримує доступ до системи, встановлюється сесія, генеруються та надсилаються клієнту токени, і він перенаправляється на відповідну сторінку. Якщо дані невірні, відображається повідомлення про помилку.

Після автентифікації користувач отримує доступ до свого профілю. Перехід до профілю здійснюється через відповідний пункт у випадаючому меню під іменем користувача в навігаційній панелі. Сторінка профілю («Profile/Profile.razor») відображає основну інформацію про користувача, таку як ім'я, електронна пошта, а також може містити статистику його активності та досягнення.

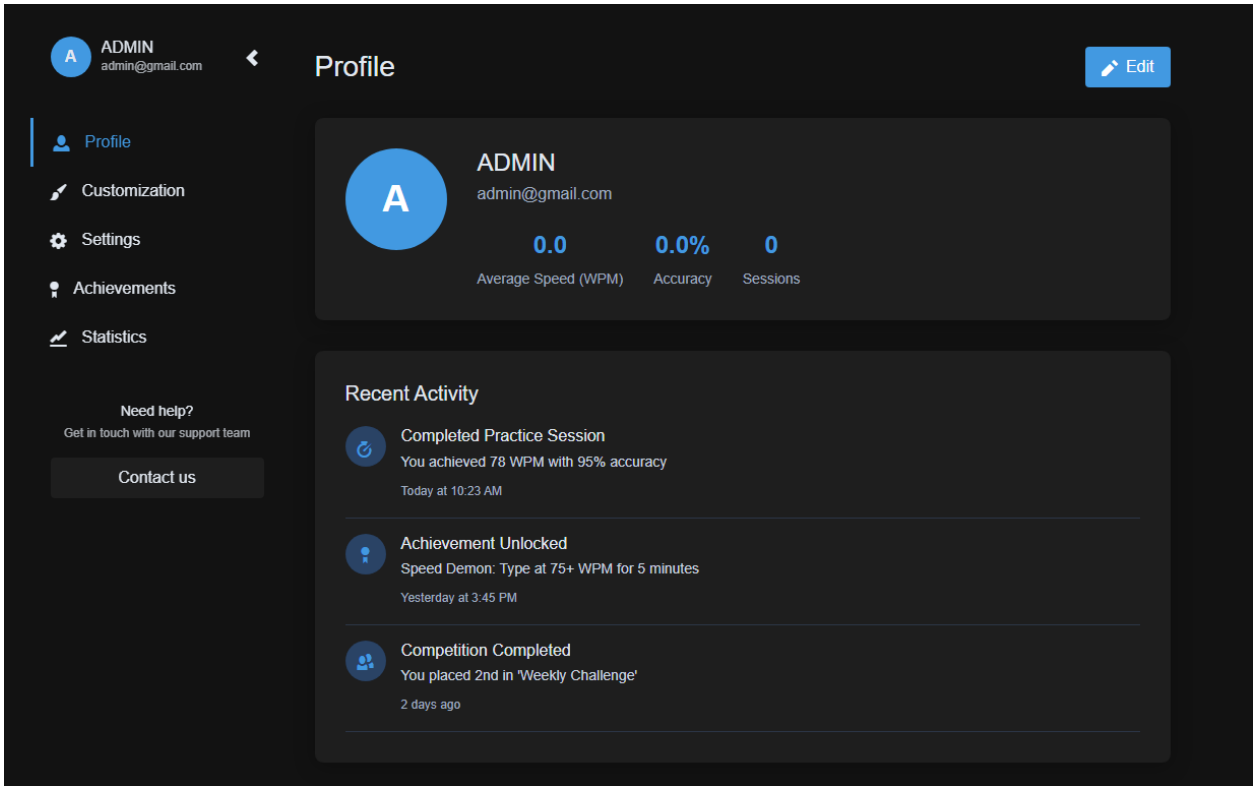

Рисунок 4.7 – Сторінка профілю користувача

Управління обліковим записом також включає можливість виходу з системи. Для цього користувач обирає пункт «Logout» у випадаючому меню навігаційної панелі. Ця дія викликає метод «Logout» в «AuthController.cs», який завершує сесію користувача, анулює токени та видаляє відповідні cookie. Після виходу користувач перенаправляється на головну сторінку або сторінку входу.

4.3 Функціонал індивідуального тренування

Після переходу до розділу індивідуального тренування, система спершу перевіряє, чи автентифікований користувач. Якщо користувач не увійшов до системи, йому буде запропоновано це зробити, оскільки результати тренувань зберігаються та прив'язуються до конкретного облікового запису. Для автентифікованого користувача відкривається інтерфейс налаштування тренувального тексту. Тут користувач може обрати джерело тексту: «Випадковий текст з бази даних» або «Згенерувати за допомогою ШІ». Також доступні опції вибору складності тексту («Легкий», «Середній», «Важкий») та мови («Російська», «Українська», «Англійська»).

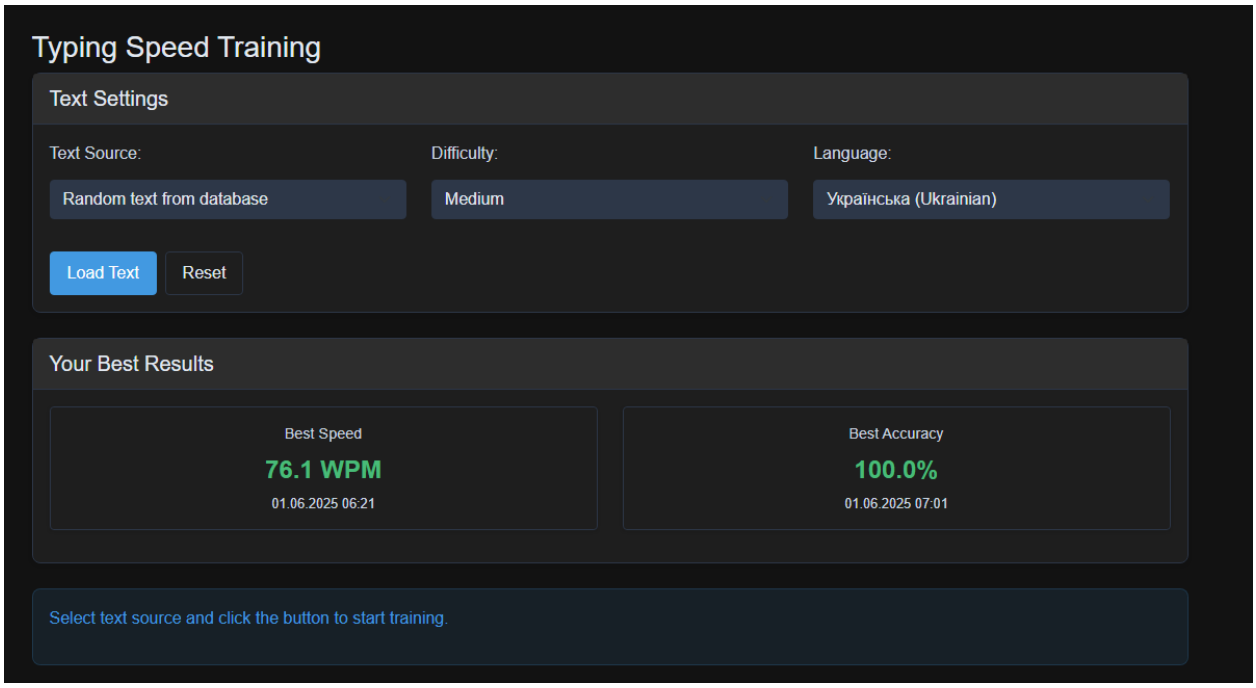

Рисунок 4.8 – Сторінка з одиночними тренуваннями

Якщо користувач обирає «Випадковий текст з бази даних» та натискає кнопку «Завантажити текст», система надсилає запит до серверного контролера «TextSnippetController.cs» на ендпоінт «/api/TextSnippet/random» з обраними параметрами складності та мови. Контролер, використовуючи «ITextSnippetRepository», вибирає відповідний текстовий фрагмент («TextSnippetDto») з бази даних.

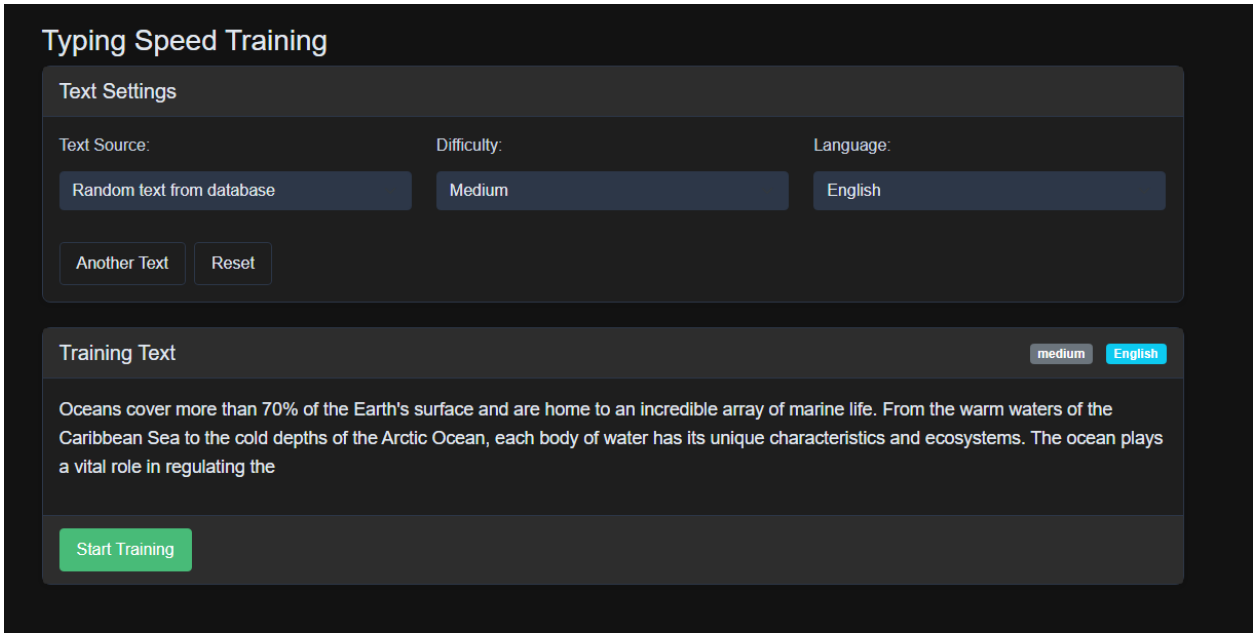

Рисунок 4.9 – Сторінка тренування з текстом

У випадку вибору «Згенерувати за допомогою ШІ», користувач додатково може вказати бажану довжину тексту в символах. Після натискання кнопки «Згенерувати текст», запит з параметрами складності, мови та довжини надсилається на ендпоінт «/api/TextGeneration/generate»

контролера «TextGenerationController.cs». Цей контролер взаємодіє з сервісом генерації тексту (наприклад, «LocalLLMTextGenerationService.cs»), який створює унікальний текстовий фрагмент. Згенерований текст також відображається для попереднього перегляду.

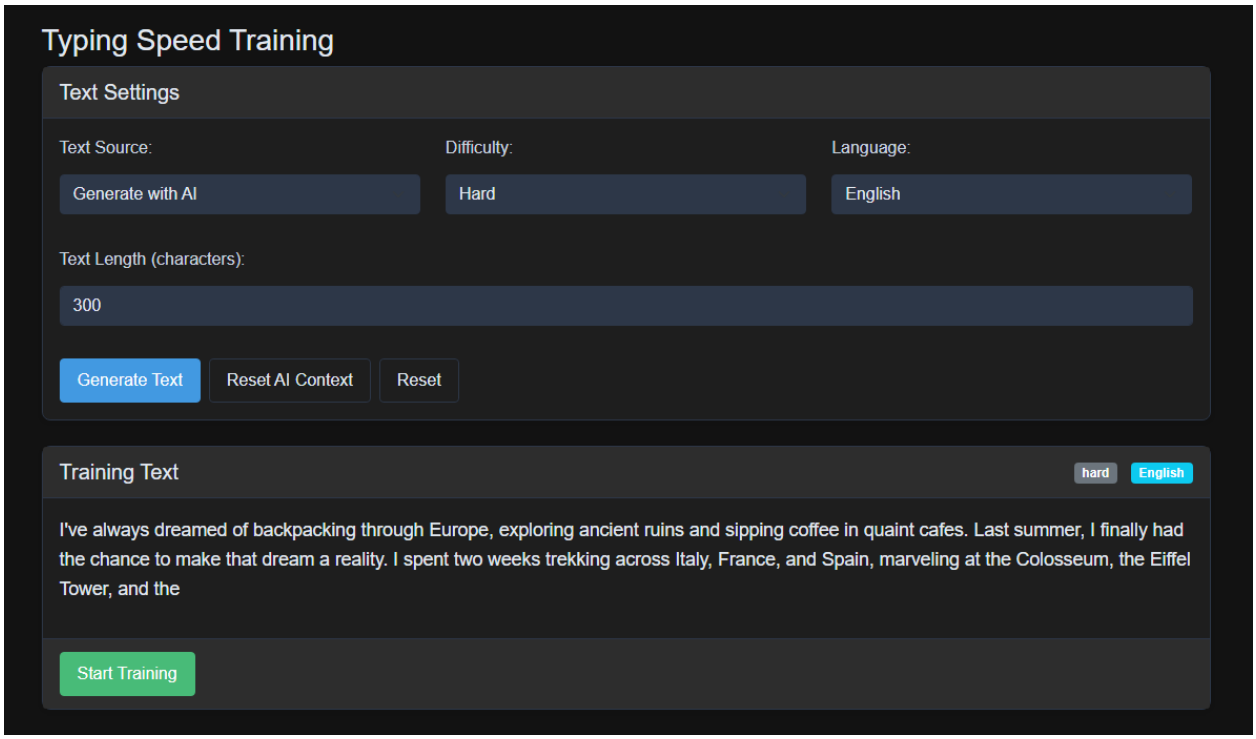

Рисунок 4.10 – Згенерований текст за допомогою ШІ

Після того, як текст обрано або згенеровано, користувач натискає кнопку «Почати тренування». Ця дія ініціює створення нової тренувальної сесії. Компонент «Train.razor» викликає метод «StartSessionAsync» сервісу «ITypingSessionService». На сервері створюється новий запис «TypingSession», що пов'язує користувача («UserProfile.Id») та обраний текстовий фрагмент («TextSnippet.Id»), і фіксується час початку сесії. Інтерфейс переходить у режим активного тренування.

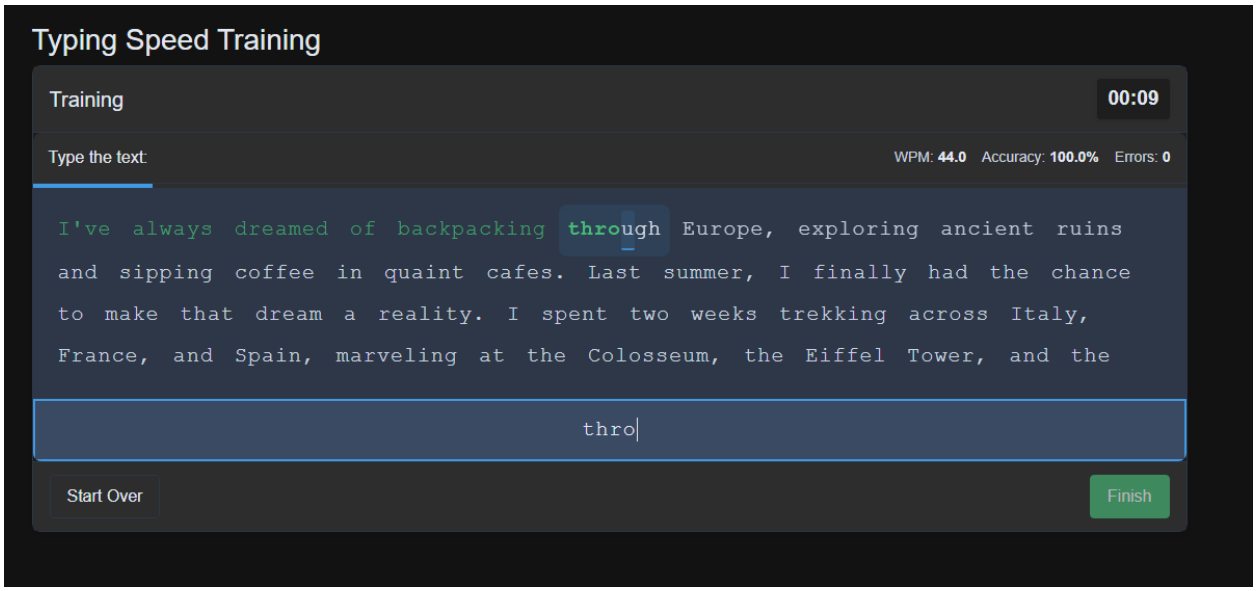

Рисунок 4.11 – Активне тренування

В активному режимі тренування користувачеві відображається текст, який необхідно набрати. Нижче розташоване поле вводу, куди користувач вводить символи. JavaScript-модуль «trainInterop.js» відіграє ключову роль на клієнтській стороні: він обробляє введення користувача, порівнює його з оригінальним текстом, підсвічує правильно та неправильно набрані символи або слова, а також оновлює в реальному часі показники: швидкість друку (WPM – слів за хвилину), точність (у відсотках) та кількість помилок. Також відображається таймер, що показує час, витрачений на тренування, та прогрес-бар, що візуалізує обсяг набраного тексту.

Коли користувач завершує набір усього тексту або натискає кнопку «Завершити», тренувальна сесія вважається завершеною. Компонент «Train.razor» викликає метод «Complete». На сервер надсилається запит (модель «SessionCompleteRequestDto») на ендпоінт «/api/Session/complete» контролера «SessionController.cs». Цей запит містить ідентифікатор сесії, розраховані показники WPM, точності, кількість помилок та тривалість сесії. Сервіс «ITypingSessionService» через метод «FinalizeAndGetResultAsync» оновлює запис «TypingSession», встановлюючи час завершення, створює запис «TypingStatistic» з детальними результатами та зберігає «TypingSessionResult». Також на цьому етапі може спрацьовувати логіка перевірки та нагородження досягнень через «IAchievementCheckerService».

Після успішного завершення сесії на сервері, користувачеві відображається модальне вікно або спеціальний блок з підсумковими результатами тренування: фінальна швидкість друку, точність, кількість помилок та загальний час, витрачений на набір. Система також може вказати, чи є поточний результат особистим рекордом користувача.

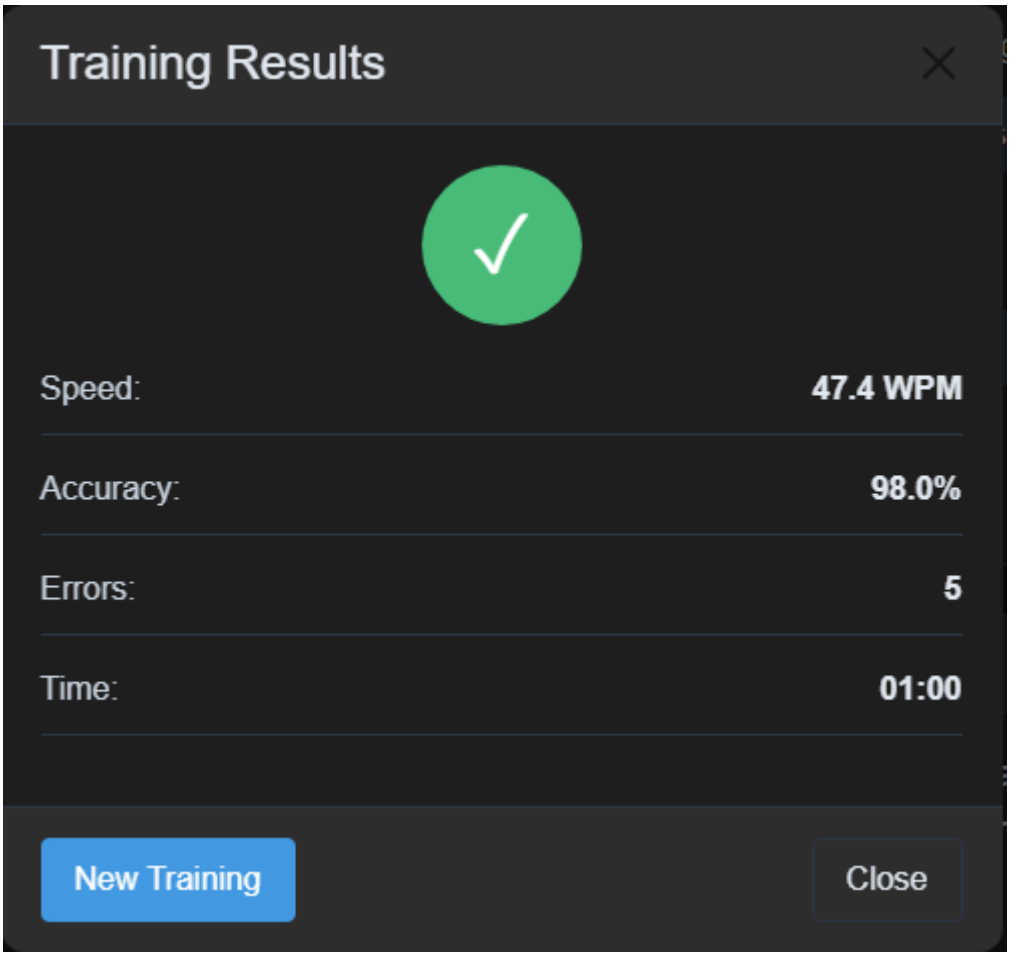

Рисунок 4.12 – Модальне вікно з результатами тренування

Користувач має можливість закрити вікно результатів та повернутися до вибору нового тексту для наступного тренування або перейти до інших розділів сервісу. Таким чином, функціонал індивідуального тренування забезпечує повний цикл від підготовки до аналізу результатів, дозволяючи користувачам ефективно вдосконалювати свої навички швидкого друку.

4.4 Режим змагань у реальному часі

Доступ до режиму змагань здійснюється через пункт «Змагання» в головному навігаційному меню, що веде на сторінку «Competitions.razor». На цій сторінці відображається список активних ігрових лобі, доступних для приєднання. Кожен елемент списку містить назву лобі, ім'я хоста, кількість гравців (поточна/максимальна), статус лобі (наприклад, «Очікування», «В грі») та кнопку «Приєднатися». Також на цій сторінці присутня кнопка «Створити лобі».

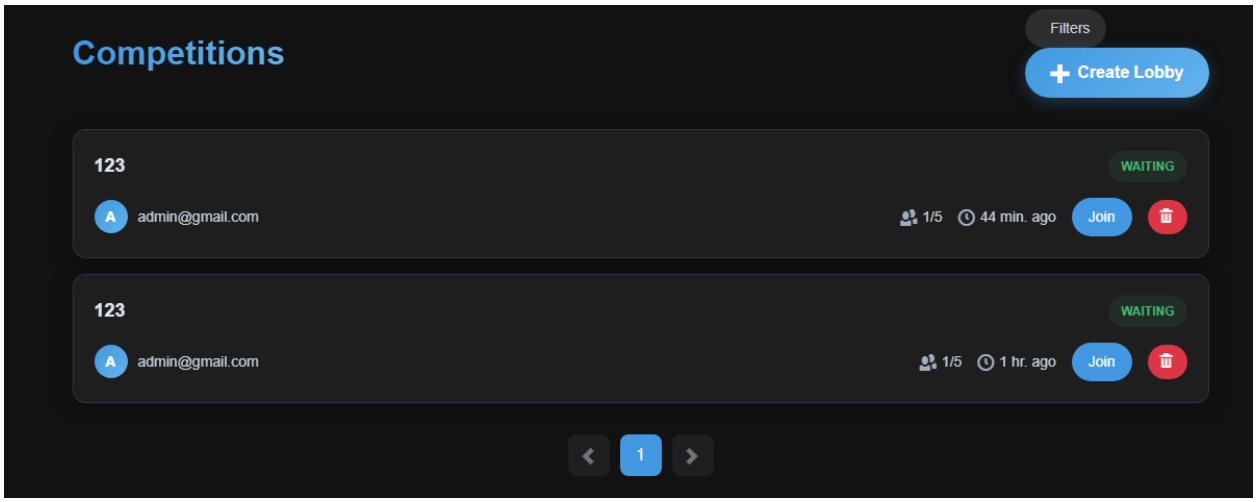

Рисунок 4.13 – Сторінка змагань

Для створення нового змагання користувач натискає кнопку «Створити лобі». Це відкриває модальне вікно, де необхідно вказати назву лобі, максимальну кількість гравців та, за бажанням, встановити пароль для приватного доступу. Дані для створення лобі представлені моделлю «CreateLobbyDto.cs». Після заповнення форми та натискання кнопки «Створити», запит надсилається на ендпоінт «/api/lobby/create» контролера «LobbyController.cs». Сервіс «RedisCompetitionLobbyService.cs» через метод «CreateLobbyAsync» створює новий запис про лобі в кеші Redis, призначаючи користувача-творця хостом. Після успішного створення, користувач автоматично перенаправляється на сторінку створеного лобі, реалізовану компонентом «CompetitionLobby.razor».

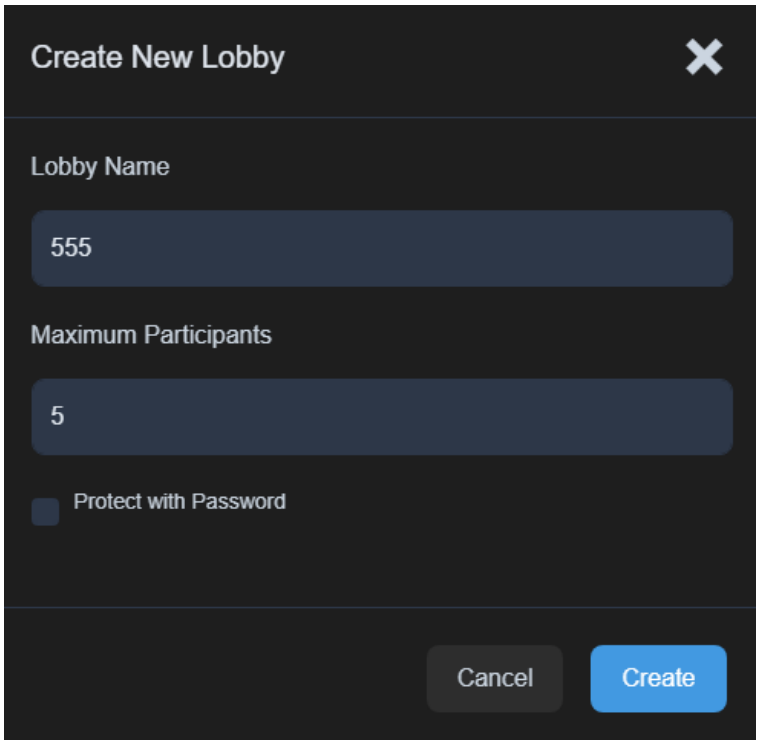

Рисунок 4.14 – Модальне вікно створення лоббі

Інші користувачі можуть приєднатися до існуючого лобі зі списку на сторінці «Competitions.razor». При натисканні кнопки «Приєднатися» для обраного лобі (і введенні пароля, якщо він встановлений), надсилається запит на ендпоінт «/api/lobby/join» контролера «LobbyController.cs» (використовується «JoinLobbyDto.cs»). Сервіс «RedisCompetitionLobbyService.cs» через метод «JoinLobbyAsync» додає користувача до списку учасників лобі. Усі учасники лобі отримують оновлення стану через SignalR хаб «TypingHub.cs» (події «PlayerJoined», «LobbyUpdated»), і новий учасник також переходить на сторінку «CompetitionLobby.razor».

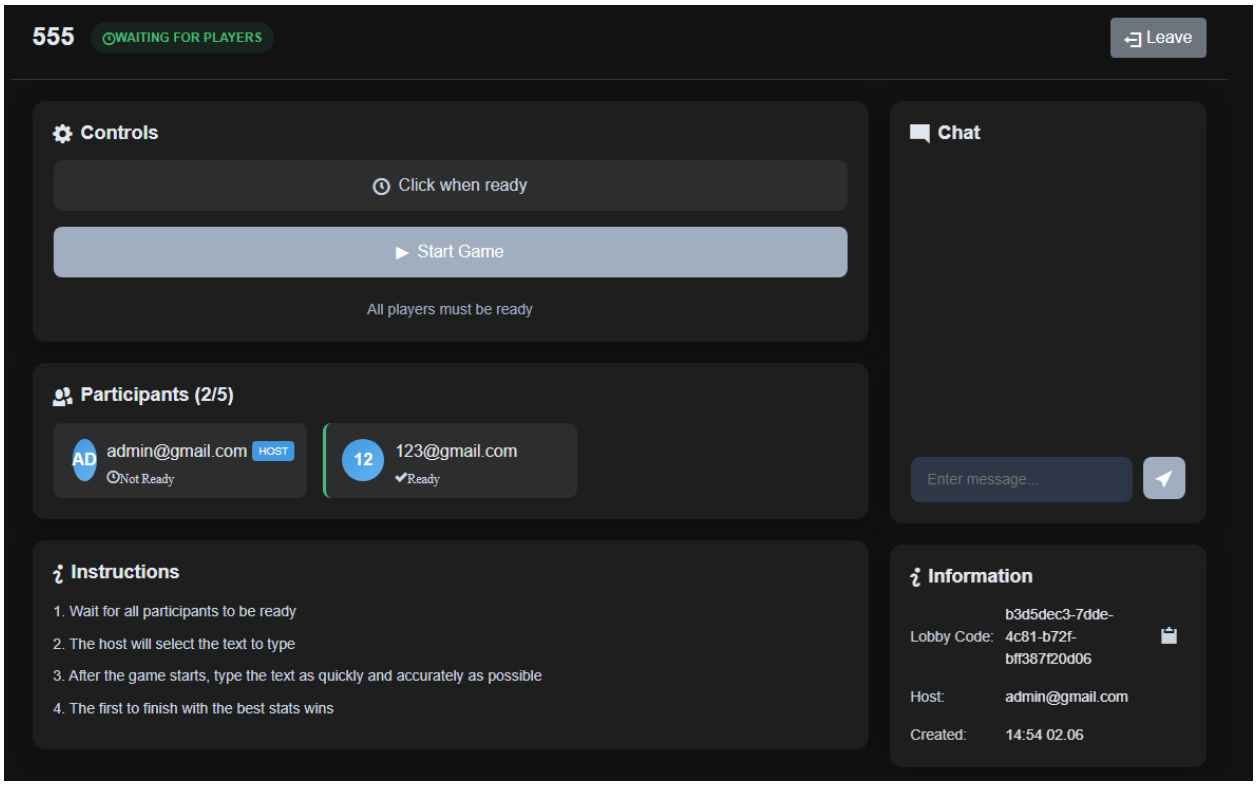

Рисунок 4.16 – Створене лоббі з двома користувачами

На сторінці лобі («CompetitionLobby.razor») у фазі очікування гравці бачать список учасників, їхній статус готовності, а також чат для спілкування. Кожен гравець може змінити свій статус готовності, натиснувши відповідну кнопку. Ця дія через «LobbyController.cs» та «RedisCompetitionLobbyService.cs» («UpdateReadyStatusAsync») оновлює статус гравця, і ця зміна транслюється іншим учасникам через «TypingHub.cs» (подія «ReadyStatusChanged»). Хост лобі бачить кнопку «Почати гру», яка стає активною, коли мінімально необхідна кількість гравців приєдналася та всі вони позначили себе як готові.

Коли хост натискає «Почати гру», він може обрати текстовий фрагмент для змагання (аналогічно до режиму індивідуального тренування, використовуючи «ITextSnippetService»). Після вибору тексту, запит на старт

гри (модель «StartGameDto.cs») надсилається на ендпоінт «/api/lobby/start» контролера «LobbyController.cs». Сервіс «RedisCompetitionLobbyService.cs» («StartGameAsync») змінює статус лобі на «InGame», зберігає ідентифікатор обраного тексту та скидає прогрес гравців. «TypingHub.cs» сповіщає всіх учасників про початок гри (подія «GameStarted»). Інтерфейс «CompetitionLobby.razor» оновлюється, відображаючи текст для набору, поле вводу та панель прогресу всіх учасників.

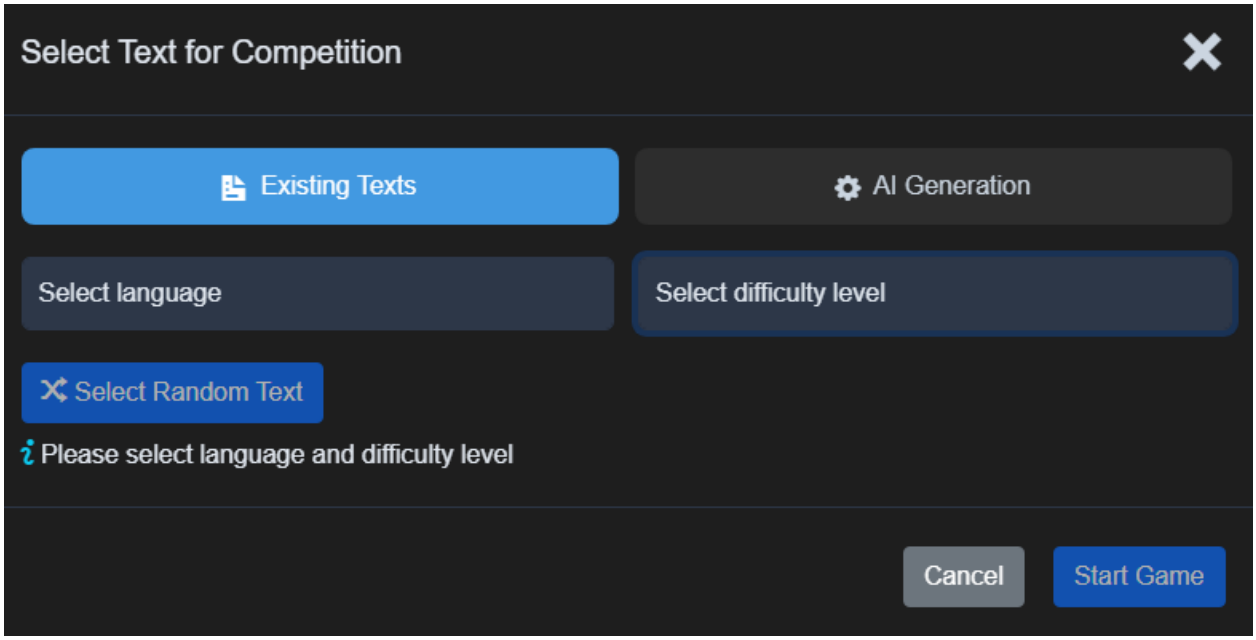

Рисунок 4.17 – Модальне вікно вибору тексту для гри

Під час змагання гравці набирають запропонований текст. Клієнтська логіка, реалізована в «lobbyInterop.js» та «CompetitionLobby.razor», відстежує прогрес кожного гравця (швидкість, точність, відсоток набраного тексту). Ці дані періодично надсилаються через «TypingHub.cs» (метод «UpdateProgress») на сервер, де «RedisCompetitionLobbyService.cs» («UpdatePlayerProgressAsync») оновлює стан гравця в лобі. Хаб транслює ці оновлення всім учасникам (подія «PlayerProgressUpdated»), що дозволяє бачити прогрес суперників у реальному часі на відповідних індикаторах.

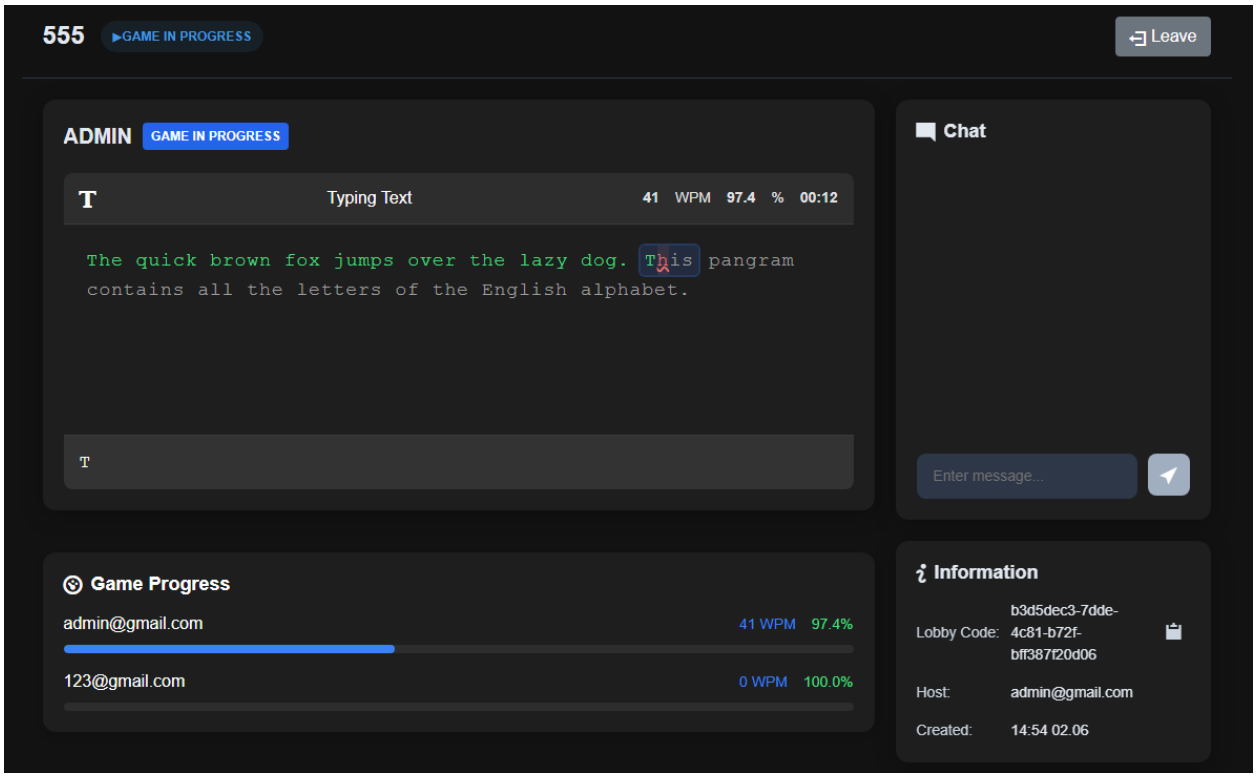

Рисунок 4.18 – Активна фаза змагання

Коли гравець завершує набір тексту, його клієнт надсилає фінальні результати (WPM, точність) через «TypingHub.cs» (метод «FinishGame») на сервер. «RedisCompetitionLobbyService.cs» (метод «CompleteGameAsync» або «PlayerFinishedAsync») фіксує завершення гри для даного гравця, його результати та позицію. Інші учасники отримують сповіщення (події «PlayerFinished», «LobbyUpdated»). Коли всі гравці завершили гру, або вийшов час (якщо реалізовано), сервіс змінює статус лобі на «Finished». «TypingHub.cs» може надіслати подію «AllPlayersFinished» та фінальні результати «GameResults».

Після завершення змагання для всіх учасників, на сторінці «CompetitionLobby.razor» відображається підсумкова таблиця або подіум з результатами. Тут показується місце кожного гравця, його ім'я, фінальна швидкість друку, точність та час. Ця інформація береться з оновленої моделі «CompetitionLobby» з сервісу «RedisCompetitionLobbyService.cs».

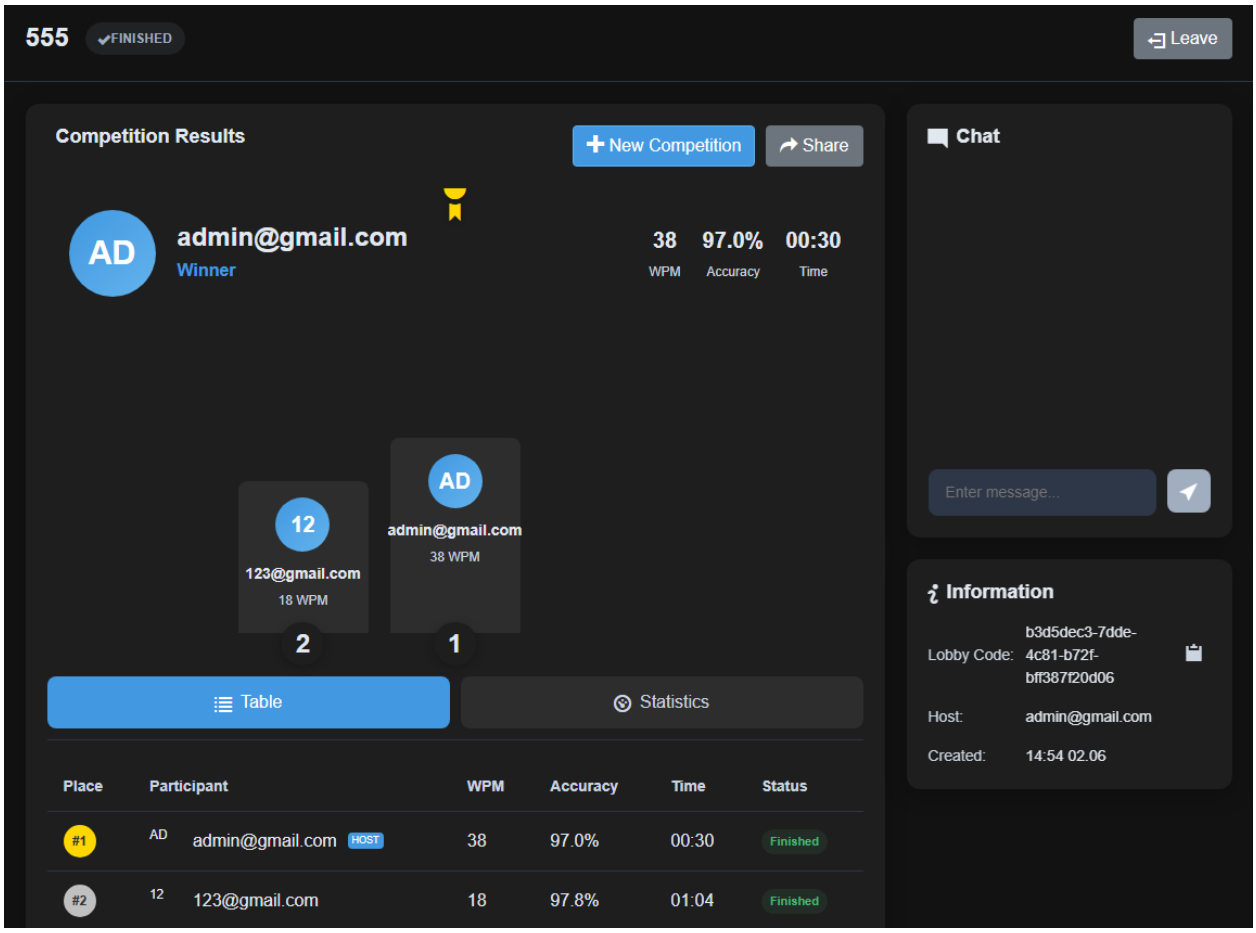

Рисунок 4.19 – Результати змагання

4.5 Функціонал сторінки практики

Сторінка «/practice», реалізована компонентом «Practice.razor», надає користувачам спеціалізоване середовище для цілеспрямованого тренування навичок друку, зосереджуючись на окремих клавішах або наборах символів для обраної мови та розкладки клавіатури. Цей режим відрізняється від загального тренування на довільних текстах («Train.razor») своєю структурою, орієнтованою на поступове освоєння клавіатури.

При першому завантаженні сторінки «/practice» може відображатися індикатор завантаження, представлений класом «loading-overlay», поки завантажуються необхідні словники слів для різних мов, наприклад, файли «english-words.json», «russian-words.json», «ukrainian-words.json» з папки «wwwroot/data/».

Основний інтерфейс сторінки «/practice» складається з кількох ключових блоків.

Перший блок – це панель метрик та керування, що має клас «metrics-bar». Розташована у верхній частині, ця панель відображає поточні показники тренування: швидкість друку («Speed») у словах за хвилину (WPM), включаючи «чистий» WPM та «сирий» WPM (Raw WPM), точність («Accuracy») у відсотках, та загальний рахунок («Score»). Праворуч на цій

панелі знаходяться кнопки керування. Кнопка «?» відкриває модальне вікно допомоги («Help»). Кнопка «↺» перезапускає поточне тренування для обраної клавіші («RestartPractice»). Кнопка «←» переходить до тренування попередньої клавіші в послідовності («PreviousPractice»). Кнопка «→» переходить до тренування наступної клавіші («NextPractice»). Кнопка «⛶» вмикає або вимикає повноекранний режим. Кнопка «⚙ SETTINGS» відкриває модальне вікно налаштувань («Settings»).

Другий блок – це інформаційна панель практики, що має клас «practice-info». Цей блок надає контекст поточного тренування. Рядок «All keys» відображає список усіх клавіш, що вивчаються для обраної мови. Кожна клавіша представлена як елемент з класом «key-badge». Поточна клавіша для тренування підсвічується класом «current», а вже освоєні клавіші, для яких завершено тренування, позначаються як «completed». Рядок «Current key» показує поточну клавішу, на якій зосереджено тренування, та короткий опис або статус для неї, наприклад, «Most common letter in English.». Рядок «Accuracy» відображає текстовий статус поточної точності, наприклад, «Excellent! Keep up the good work.».

Третій блок – це область друку, що має клас «typing-area». Це центральний елемент, де відбувається безпосередній процес набору тексту. Внутрішній елемент з класом «target-text» відображає текст, який необхідно набрати. Символи в цьому тексті динамічно змінюють свій стиль: правильно набрані стають «correct», неправильно набрані – «incorrect» і підкреслюються, а поточний символ, який очікується для вводу, підсвічується як «current» з анімацією миготіння. Пробіли можуть візуалізуватися спеціальним символом «space-dot». Приховане поле з класом «typed-text» зберігає фактично набраний користувачем текст. Підказка «Click here and start typing...» з'являється, якщо область друку не в фокусі.

Четвертий блок – це візуалізація клавіатури, що має клас «keyboard-container». Внизу сторінки відображається інтерактивна клавіатура. Її вигляд залежить від обраної мови та розкладки, що зберігається у змінній «keyboardLayout». Для англійської мови доступні розкладки QWERTY, Dvorak, Colemak. Для російської та української мов використовується стандартна розкладка. Клавіші на віртуальній клавіатурі динамічно змінюють свій вигляд. Клас «pressed» застосовується, коли клавіша підсвічується при натисканні фізичної клавіші. Класи «heat-low», «heat-medium», «heat-high» можуть змінювати колір клавіші залежно від частоти її правильного натискання, створюючи ефект теплової карти. Клас «highlighted» може застосовувати спеціальне підсвічування до клавіші, що є поточною для тренування. Клас «completed» може позначати клавіші, для яких тренування успішно завершено.

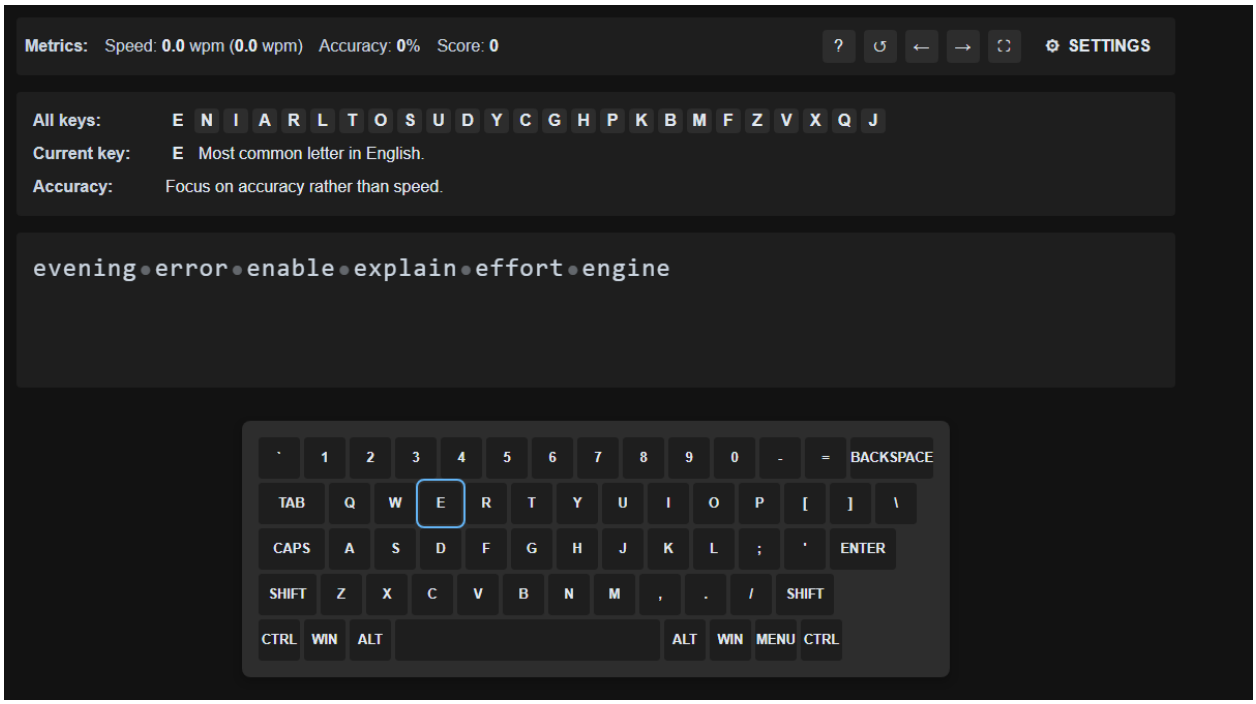

Рисунок 4.20 – Сторінка практики

Сторінка також використовує декілька модальних вікон.

Модальне вікно налаштувань, що має клас «settings-modal», відкривається кнопкою «⚙ SETTINGS». Воно дозволяє користувачеві обрати мову тренування: «English», «Русский», «Українська», та розкладку клавіатури для англійської мови: «QWERTY», «Dvorak», «Colemak». Для інших мов розкладка фіксована. Зміни зберігаються кнопкою «Save» або скасовуються кнопкою «Cancel». Зміна мови призводить до скидання прогресу та початку тренування з першої клавіші нової мови.

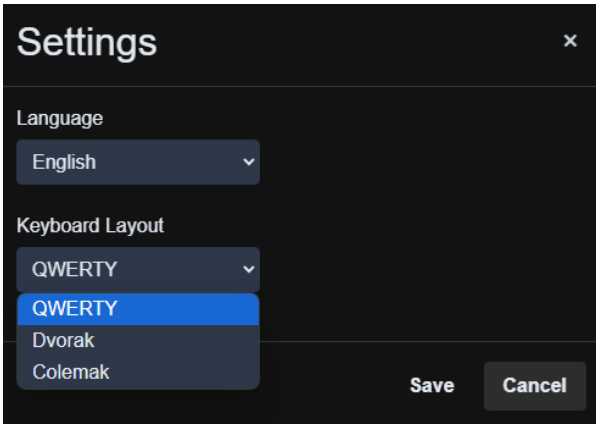

Рисунок 4.21 – Модальне вікно налаштувань

Модальне вікно допомоги, що має клас «help-modal», відкривається кнопкою «?». Воно містить інформацію про те, як користуватися тренажером, список гарячих клавіш, наприклад, Esc для перезапуску,

Ctrl+Enter для нового тренування, та пояснення метрик WPM, Accuracy, Score.

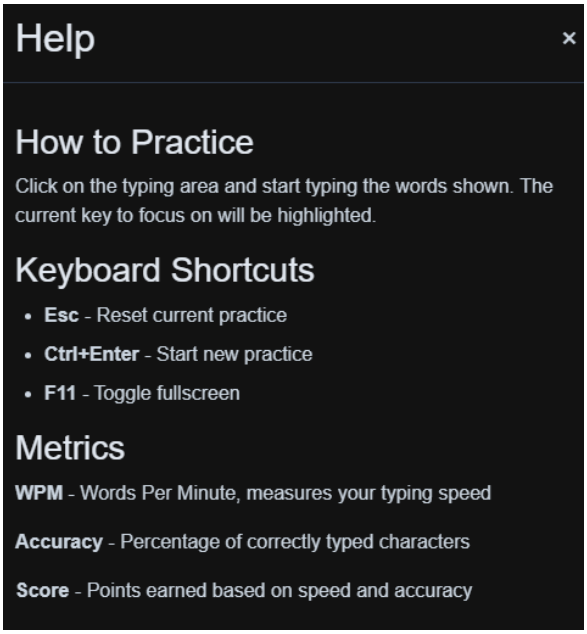

Рисунок 4.22 – Модальне вікно допомоги

Модальне вікно зворотного зв'язку по точності, що має клас «feedback-modal», з'являється автоматично після завершення тренування для поточної клавіші. Воно відображає результати для цієї клавіші: точність, швидкість, рахунок, текстове повідомлення, наприклад, «Excellent! Your accuracy with this key is outstanding.», та таймер зворотного відліку до автоматичного переходу до наступної клавіші.

Модальне вікно загальної статистики, що має клас «statistics-modal», з'являється після завершення тренування всіх клавіш для обраної мови. Воно показує загальні результати: середню точність, середню швидкість, загальний рахунок. Також може відображати список клавіш, які потребують додаткової практики («Keys to Practice»), та рекомендації. Кнопка «Practice Again» дозволяє скинути весь прогрес та почати тренування заново.

Логіка роботи компонента «Practice.razor» використовує словники слів, що зберігаються у змінній «languagePracticeWords», які є специфічними для кожної мови та часто для кожної окремої клавіші. При виборі клавіші для тренування, що зберігається у змінній «currentKey», генерується набір слів, представлений змінною «practiceWords», що містять цю клавішу. Користувач набирає цей текст. Система в реальному часі розраховує WPM, точність та рахунок. Після успішного завершення тексту для однієї клавіші, користувач переходить до наступної. Прогрес по кожній клавіші, включаючи статистику та статус завершення, зберігається у словнику «keyStatistics».

ВИСНОВКИ

У ході виконання даної роботи було успішно розроблено та описано веб-сервіс KeyRaces, призначений для тренування навичок швидкого друку та проведення змагань між користувачами. Проєкт охопив усі етапи створення програмного продукту, від аналізу вимог та проєктування архітектури до реалізації функціоналу, тестування та підготовки до розгортання.

Основними досягненнями роботи є:

1. Створення повнофункціонального веб-додатку на платформі Blazor Server з використанням .NET Core, що забезпечує високу продуктивність та інтерактивність користувацького інтерфейсу.
2. Реалізація багатошарової архітектури (Core, Infrastructure, Server), яка сприяє гнучкості, масштабованості та легкості підтримки коду.
3. Розробка комплексного функціоналу, що включає реєстрацію та автентифікацію користувачів, різні режими тренувань, створення та участь у змаганнях, систему досягнень, таблиці лідерів, а також адміністративну панель для управління системою.
4. Впровадження технології SignalR для забезпечення взаємодії в реальному часі, зокрема для відображення прогресу учасників у змаганнях та функціонування чату в лобі.
5. Інтеграція з локальною великою мовною моделлю Ollama, що дозволило реалізувати унікальну функцію динамічної генерації текстів для тренувань, значно збагативши контентну базу додатку.
6. Використання Entity Framework Core для взаємодії з базою даних PostgreSQL, включаючи розробку схеми даних та механізмів міграції.
7. Забезпечення базового рівня якості програмного продукту шляхом написання модульних тестів для ключових компонентів бізнес-логіки.
8. Підготовка додатку до розгортання за допомогою технології контейнеризації Docker, що спрощує процес деплою та забезпечує консистентність середовища виконання.

Проєкт KeyRaces продемонстрував можливість ефективного застосування сучасних веб-технологій для створення складних інтерактивних додатків. Розроблений сервіс надає користувачам зручну та мотиваційну платформу для вдосконалення навичок швидкого друку, а також для змагального духу. Обрані технологічні рішення та архітектурні підходи забезпечують надійну основу для подальшого розвитку та розширення функціоналу проєкту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Офіційний сайт Typing. URL: https://www.typing.com/ (дата звернення 25.04.2025).
2) Офіційний сайт 10FastFingers. URL: https://10fastfingers.com/ (дата звернення 25.04.2025).
3) Офіційний сайт Keybr. URL: https://www.keybr.com/ (дата звернення 25.04.2025).
4) Офіційний сайт TypeRacer. URL: https://play.typeracer.com/ (дата звернення 25.04.2025).
5) Офіційний сайт TypingMaster. URL: https://www.typingmaster.com/ (дата звернення 25.04.2025).
6) Офіційний сайт Klavaro. URL: https://klavaro.sourceforge.io/ (дата звернення 25.04.2025).
7) Офіційний сайт Elo. URL: https://elo.international/uni/what-is-elo/ (дата звернення 26.04.2025).
8) Feit A. M., Logan J. Cognitive and motor aspects of text entry: A large-scale study of typing performance across skill levels. Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems. New York : ACM, 2018. С. 1-12.
9) Salthouse T. A. Electromyographic analysis of typing: Effects of skill level and keyboard design. Ergonomics. 2020. Т. 63, № 5. С. 612-627.
10) Yamanashi K., Takahashi R., Nakamura S. Error patterns in keyboard typing: A comprehensive analysis across languages and skill levels. International Journal of Human-Computer Studies. 2022. Т. 158. С. 102745.
11) Документація Microsoft, SignalR. URL: https://learn.microsoft.com/en-us/aspnet/signalr/ (дата звернення 01.05.2025).
12) Документація Microsoft, .NET8. URL: https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-8/overview (дата звернення 01.05.2025).
13) Офіційний сайт Redis. URL: https://redis.io/ (дата звернення 02.05.2025).
14) Документація PostgreSQL. URL: https://www.postgresql.org/docs/ (дата звернення 02.05.2025).
15) Документація Microsoft, EntityFramework. URL: https://learn.microsoft.com/en-us/ef/ (дата звернення 02.05.2025).
16) Документація Docker. URL: https://docs.docker.com/ (дата звернення 03.05.2025).
17) Бібліотека моделей Ollama. URL: https://ollama.com/library (дата звернення 04.05.2025).

18) Документація Microsoft, WebSockets. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/websockets?view=aspnetcore-9.0 (дата звернення 05.05.2025).
19) REST API як спосіб спілкування компонент додатків. URL: https://foxminded.ua/shcho-take-rest-api/ (дата звернення 06.05.2025).

ДОДАТКИ

Додаток А

Код проєкту на github: https://github.com/etherfoun/keyraces

Додаток Б

Зміст файлу "Dockerfile"

```
# Stage 1: build and publish
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src

# Copy project files for all three projects
COPY ["keyraces.Core/keyraces.Core.csproj", "keyraces.Core/"]
COPY ["keyraces.Infrastructure/keyraces.Infrastructure.csproj",
"keyraces.Infrastructure/"]
COPY ["keyraces.Server/keyraces.Server.csproj", "keyraces.Server/"]

# Restore only the server project (it will pull in the others by
project-reference)
RUN dotnet restore "keyraces.Server/keyraces.Server.csproj"

# Copy all source code
COPY . .

# Publish the server project
WORKDIR "/src/keyraces.Server"
RUN dotnet publish "keyraces.Server.csproj" -c Release -o /app/publish

# Stage 2: runtime image
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime
WORKDIR /app

# Copy the published output from the build stage
COPY --from=build /app/publish .

# Expose the port the app runs on
EXPOSE 80
EXPOSE 8080

# Start the application
ENTRYPOINT ["dotnet", "keyraces.Server.dll"]
```

Зміст файлу "docker-compose.yml":

```
services:
  keyraces:
    image: mcr.microsoft.com/dotnet/sdk:8.0
    container_name: keyraces_app
    ports:
      - "7077:8080"
    depends_on:
      db:
        condition: service_healthy
      redis:
        condition: service_healthy
    environment:
      -
ConnectionStrings__Default=Host=db;Port=5432;Database=keyracesdb;Usern
ame=keyraces_user;Password=1
      - Redis__Connection=redis:6379
      - ASPNETCORE_ENVIRONMENT=Docker
```

```
      - DOTNET_WATCH_RESTART_ON_RUDE_EDIT=true
      - ASPNETCORE_URLS=http://0.0.0.0:8080
      - ASPNETCORE_HTTP_PORTS=8080
      - Ollama__ApiUrl=http://host.docker.internal:11434/api/generate
      - Ollama__ModelName=llama2
    volumes:
      - ./:/src
      - nuget-packages:/root/.nuget/packages
      - /src/keyraces.Core/bin
      - /src/keyraces.Core/obj
      - /src/keyraces.Infrastructure/bin
      - /src/keyraces.Infrastructure/obj
      - /src/keyraces.Server/bin
      - /src/keyraces.Server/obj
    working_dir: /src
    restart: unless-stopped
    networks:
      - keyraces-network
    command: >
      bash -c "cd keyraces.Server &&
      dotnet restore &&
      dotnet build &&
      dotnet watch run --no-build --urls http://0.0.0.0:8080"

  db:
    image: postgres:15
    container_name: keyraces_db
    environment:
      POSTGRES_USER: keyraces_user
      POSTGRES_PASSWORD: 1
      POSTGRES_DB: keyracesdb
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    restart: unless-stopped
    networks:
      - keyraces-network
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U keyraces_user -d keyracesdb"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 10s

  redis:
    image: redis:7-alpine
    container_name: keyraces_redis
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    command: redis-server --appendonly yes
    restart: unless-stopped
    networks:
      - keyraces-network
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
```

```
      interval: 10s
      timeout: 5s
      retries: 5

networks:
  keyraces-network:
    driver: bridge

volumes:
  postgres_data:
  redis_data:
  nuget-packages:
```