НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В.о. завідувача кафедри
_____ Артем ВОЛОКИТА
(підпис)

“__” _____2026 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Система для координації та взаємодії спільнот любителів настільних ігор

Виконав : студент __4__ курсу, групи ІМ-21
(шифр групи)

_____Русак Я. А._____ _____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____асистент, к.т.н. Ковальчук О. М._____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) ас., д-р. філос. Коренко Д. В. _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _ас., д-р. філос., ас. кафедри ІСТ Нікітін В. А.___ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2026 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Артем ВОЛОКИТА
(підпис)
“__” _____2026 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Русака Я. А.

1. Тема проєкту *Система для координації та взаємодії спільнот любителів настільних ігор*
керівник проєкту Ковальчук О. М., асистент, к.т.н.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 3 червня 2026 року №2056-с

2. Термін здачі студентом закінченого проєкту *4 червня 2026 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

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

Розділ 1. Аналіз предметної області та огляд існуючих рішень.

Розділ 2. Проєктування системи та вибір технологій.

Розділ 3. Реалізація програмного забезпечення.

Розділ 4. Тестування, розгортання та використання системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи (структурна схема), діаграма класів (функціональна схема), ЕР-діаграма (принципова схема).

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Коренко Д. В.		

7. Дата видачі завдання «24» вересня 2025 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту
1.	*Затвердження теми проєкту*	10.01.2026-21.01.2026
2.	*Вивчення та аналіз завдання*	21.01.2026-22.03.2026
3.	*Розробка архітектури та загальної структури системи*	22.03.2026-07.04.2026
4.	*Розробка структур окремих підсистем*	07.04.2026-26.04.2026
5.	*Програмна реалізація системи*	26.04.2026-15.05.2026
6.	*Оформлення пояснювальної записки*	15.05.2026-25.05.2026
7.	*Захист програмного продукту*	28.05.2026
8.	*Передзахист*	02.06.2026
9.	*Захист*	18.06.2026

Студент-дипломник _____ Ярослав РУСАК
(підпис)

Керівник проєкту _____ Олександр КОВАЛЬЧУК
(підпис)

АНОТАЦІЯ

У цій роботі розроблено веборієнтовану систему координації та взаємодії спільнот любителів настільних ігор. Метою роботи є створення програмного продукту, який спрощує організацію ігрових зустрічей, взаємодію між учасниками спільноти та керування запрошеннями до подій. У межах проєкту реалізовано клієнтську та серверну частини застосунку, механізми реєстрації й автентифікації користувачів, керування профілями, формування подій, підтримку соціальних зв'язків між учасниками (друзі/контакти), а також обмін даними в режимі, наближеному до реального часу. Серверну частину побудовано на базі NestJS, клієнтську — із використанням React і Vite, а доступ до даних організовано через Prisma ORM. Результатом виконання роботи є працездатний прототип інформаційної системи, придатний до подальшого розширення та використання як основа для цифрової підтримки діяльності ігрових спільнот.

Ключові слова: веборієнтована інформаційна система, настільні ігри, координація спільноти, організація подій, соціальна взаємодія, NestJS, React, TypeScript, Prisma, PostgreSQL.

ANNOTATION

This work presents the development of a web-based system for coordination and interaction within board game enthusiast communities. The purpose of the work is to create a software product that simplifies meetup planning, participant communication, and invitation management. Within the project scope, both client-side and server-side components were implemented, including user registration and authentication mechanisms, profile management, event creation and maintenance, support for social connections between users (friends/contacts), and near real-time data exchange. The backend was developed using NestJS, while the frontend was implemented with React and Vite; data access and persistence were organized through Prisma ORM. The result of the work is a functional prototype of an information

system suitable for further extension and practical use as a digital platform supporting community-driven board game activities.

Keywords: web-based information system, board games, community coordination, event management, social interaction, NestJS, React, TypeScript, Prisma, PostgreSQL.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземп-ляра	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Система для координації та	4		
			взаємодії спільнот			
			любителів настільних ігор			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Система для координації та	93		
			взаємодії спільнот			
			любителів настільних ігор			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Система для координації та	1		
			взаємодії спільнот			
			любителів настільних ігор			
			Структурна схема			
			Системи (структурна схема)			
	А4	***ІАЛЦ.467200.005 Д2***	Система для координації та	1		
			взаємодії спільнот			
			любителів настільних ігор			
			Діаграма класів			
			(функціональна схема)			
	А4	***ІАЛЦ.467200.006 Д3***	Система для координації та	1		
			взаємодії спільнот			
			любителів настільних ігор			
			ЕР-діаграма (принципова			
			схема)			
	А4	***ІАЛЦ.467200.007 Д4***	Система для координації та	68		
			взаємодії спільнот			
			любителів настільних ігор			
			Текст програмного коду			

Зм	*Лист*	*№ докум.*	*Підп.*	*Дата*	

ІАЛЦ.467200.001 ОА

Розробив		Русак Я.А.			*Система для координації та взаємодії спільнот любителів настільних ігор* Опис альбому	Літ. / Аркуш: 1 / Аркушів: 1
Перевірив		Ковальчук О.М.				***КПІ ім. Ігоря Сікорського ФІОТ ІМ-21***

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Система для координації та взаємодії спільнот любителів настільних ігор»

Київ – 2026

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Це технічне завдання полягає в установленні складу, структури, функціональних, нефункціональних, організаційних і технологічних вимог до створення програмного засобу, призначеного для інформаційної підтримки процесів взаємодії учасників спільнот любителів настільних ігор, зокрема планування та супроводу ігрових зустрічей, адміністрування користувацьких облікових записів, керування запрошеннями, а також підтримки сервісів міжкористувацької комунікації.

Галузь застосування — діяльність організованих та неформальних спільнот любителів настільних ігор (клубів, гуртків, громадських об’єднань, закладів дозвілля, студентських ініціатив тощо), у межах якої виникає потреба в автоматизації процесів координації учасників, планування спільних заходів, обліку взаємодій і забезпечення оперативного інформаційного обміну засобами веборієнтованої інформаційної системи.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «Бакалавр інженерії програмного забезпечення», який був затверджений факультетом «Інформатики та обчислювальної техніки» кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка системи для координації та взаємодії спільнот любителів настільних ігор, що дозволить ефективно вирішувати дану задачу.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розроблення даного дипломного проєкту є офіційна технічна документація до використаних програмних платформ, фреймворків, бібліотек та інструментальних засобів, науково-технічні публікації з питань проєктування веборієнтованих інформаційних систем, нормативні документи у сфері розроблення програмного забезпечення, а також актуальні методичні матеріали щодо забезпечення якості, безпеки та супровідності програмних продуктів.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- забезпечувати реєстрацію, автентифікацію та авторизацію користувачів із розмежуванням прав доступу відповідно до ролей;
- забезпечувати ведення та актуалізацію профілів користувачів;
- забезпечувати створення, редагування, перегляд і скасування ігрових зустрічей (подій);
- забезпечувати механізми запрошення користувачів до участі у зустрічах та оброблення статусів запрошень;
- забезпечувати засоби встановлення, перегляду та керування зв’язками між користувачами (друзі/контакти спільноти);
- забезпечувати пошук і фільтрацію даних за визначеними атрибутами (події, користувачі, запрошення);
- забезпечувати цілісність, несуперечливість і актуальність даних під час одночасної роботи користувачів;
- забезпечувати відмовостійку роботу серверної частини та коректне оброблення помилкових і виняткових ситуацій;
- забезпечувати масштабованість, супровідність та можливість подальшого функціонального розширення системи;
- забезпечувати сумісність клієнтської частини з актуальними версіями сучасних веббраузерів.

5.2. Вимоги до програмного забезпечення

- Операційна система Windows, Mac чи Linux.
- Node.JS версії 20 або новіше

5.3. Вимоги до апаратної частини

- Центральний процесор: AMD Ryzen 5 5500U (2.1 ГГц) або краще.
- Оперативна пам’ять: не менше ніж 8 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	10.01.2026-21.01.2026
Вивчення та аналіз завдання	21.01.2026-22.03.2026
Розробка архітектури та загальної структури системи	22.03.2026-07.04.2026
Розробка структур окремих частин системи	07.04.2026-26.04.2026
Програмна реалізація системи	26.04.2026-10.05.2026
Виправлення помилок	10.05.2026-15.05.2026
Оформлення пояснювальної записки	15.05.2026-25.05.2026

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Система для координації та взаємодії спільнот любителів настільних ігор»

Київ – 2026

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API	Інтерфейс програмування застосунків
AWS	Хмарна платформа Amazon Web Services
CI/CD	Безперервна інтеграція та доставка
CORS	Cross-Origin Resource Sharing
DNS	Система доменних імен
DTO	Об’єкт передачі даних
HTML	Мова розмітки гіпертекстових документів
IDE	Інтегроване середовище розробки
JSON	Текстовий формат обміну даними
JWT	Вебтокен на основі JSON
OAuth	Відкритий стандарт авторизації
ORM	Об’єктно-реляційне відображення
REST	Архітектурний стиль взаємодії через HTTP
S3	Amazon Simple Storage Service
SCSS	Sassy CSS
SPA	Односторінковий вебзастосунок
SQL	Мова структурованих запитів до БД
SSH	Secure Shell
TLS	Transport Layer Security
URL	Уніфікований локатор ресурсу
VPS	Віртуальний приватний сервер
WebSocket	Двосторонній протокол обміну в реальному часі
БД	База даних

ВСТУП

Настільні ігри вже давно перестали бути лише способом провести вільний вечір у вузькому колі знайомих. Для багатьох людей вони стали окремим видом дозвілля, що поєднує спілкування, стратегічне мислення, командну взаємодію та живий соціальний контакт. На відміну від багатьох цифрових розваг, настільні ігри потребують безпосередньої участі кількох людей, узгодження часу, місця, складу учасників і вибору гри. Саме тому навколо цього захоплення природно формуються локальні спільноти, клуби та неформальні групи гравців.

Водночас розвиток таких спільнот часто стикається з організаційними труднощами. Навіть за наявності зацікавлених учасників не завжди легко знайти людей із подібними ігровими вподобаннями, домовитися про зустріч, визначити зручний формат гри або підтримувати постійний зв’язок між учасниками. Частина комунікації зазвичай відбувається у месенджерах, частина інформації зберігається в окремих таблицях, списках або усних домовленостях. Через це дані про ігри, учасників, заплановані зустрічі та попередній досвід взаємодії залишаються розрізненими.

Актуальність створення спеціалізованої системи для любителів настільних ігор полягає в потребі об’єднати ці процеси в одному зручному цифровому середовищі. Така система має не лише надавати доступ до каталогу ігор, а й допомагати користувачам взаємодіяти між собою: створювати профілі, додавати ігри до власної колекції, оцінювати їх, залишати відгуки, знаходити інших гравців, організовувати ігрові зустрічі та вести комунікацію щодо них. Завдяки цьому цифровий інструмент може виступати не заміною живої взаємодії, а засобом, який робить її простішою та регулярнішою.

У межах цієї дипломної роботи розглядається розробка веборієнтованої системи для координації та взаємодії спільнот любителів настільних ігор. Основна увага приділяється побудові програмного продукту, який поєднує соціальні можливості та функції організації ігрових подій. Користувач

системи може переглядати каталог настільних ігор, формувати власну колекцію, оцінювати ігри, залишати рецензії, знаходити інших учасників спільноти, створювати зустрічі та обмінюватися повідомленнями.

Метою дипломної роботи є створення системи, що спрощує організацію ігрових зустрічей і підтримує взаємодію між учасниками спільнот настільних ігор. Для досягнення цієї мети необхідно проаналізувати предметну область, визначити основні потреби користувачів, спроєктувати архітектуру застосунку, реалізувати серверну та клієнтську частини, а також забезпечити роботу ключових функцій: автентифікації користувачів, каталогу ігор, колекцій, оцінок і відгуків, системи друзів, зустрічей та обміну повідомленнями.

Практична цінність розробленої системи полягає в тому, що вона може бути використана як основа для онлайн-платформи локальної або ширшої спільноти гравців. Її функціональність спрямована на зменшення організаційних бар’єрів, покращення комунікації між учасниками та створення єдиного простору, у якому зберігається інформація про ігри, користувачів і заплановані події. Це робить систему корисною як для окремих гравців, так і для невеликих клубів або груп, які регулярно проводять ігрові зустрічі.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1. Аналіз потреб спільноти любителів настільних ігор

1.1.1. Характеристика цільової авдиторії та специфіка хобі

Цільовою авдиторією системи є користувачі, які цікавляться настільними іграми та беруть участь в ігрових зустрічах у приватному або спільнотному форматі. До цієї групи належать як новачки, що лише знайомляться з настільними іграми, так і досвідчені гравці, які мають власні колекції, стежать за новинками, регулярно відвідують ігрові події або самостійно організовують партії. Окрему частину авдиторії становлять організатори локальних зустрічей, адміністратори клубів, власники невеликих ігрових просторів та активні учасники спільнот, які координують інших гравців.

Настільні ігри як хобі мають низку особливостей, що відрізняють їх від багатьох інших видів дозвілля. Насамперед більшість ігор передбачає участь кількох осіб, тому для проведення ігрової партії необхідно заздалегідь узгодити склад учасників, час, місце та тривалість зустрічі. Крім того, різні ігри мають різні вимоги до кількості гравців, рівня підготовки, тривалості партії та складності правил. Через це вибір гри часто залежить не лише від особистих уподобань, а й від можливостей конкретної групи учасників.

Користувачі, які займаються настільними іграми регулярно, зазвичай мають власні вподобання щодо жанрів, механік і формату гри. Одні надають перевагу коротким сімейним або компанійським іграм, інші цікавляться складними стратегічними проєктами, кооперативними іграми, рольовими системами або дуельними форматами. Також важливим чинником є наявність конкретної гри у когось із учасників, оскільки настільна гра є фізичним об’єктом, який потрібно принести на зустріч або мати в ігровому просторі.

Саме тому інформація про колекції користувачів може бути корисною для планування майбутніх партій.

Для новачків важливими є простота входження у спільноту, можливість знайти відкриті зустрічі та отримати базову інформацію про гру до участі в ній. Такі користувачі можуть не знати правил, не мати власної колекції та потребувати допомоги з вибором події відповідно до свого рівня досвіду. Натомість досвідчені гравці частіше зацікавлені в пошуку конкретних ігор, партнерів зі схожими вподобаннями, зручному плануванні партій та обговоренні ігрового досвіду після зустрічей.

Окрему роль у межах цільової авдиторії відіграють організатори. Їхні потреби пов’язані не лише з участю в іграх, а й з управлінням подіями: створенням опису зустрічі, визначенням місця проведення, обмеженням кількості учасників, вибором гри та комунікацією з запрошеними гравцями. Для таких користувачів важливо мати інструмент, який дозволяє швидко донести інформацію до учасників і зменшити кількість ручних домовленостей у різних каналах зв’язку.

Таким чином, спільнота любителів настільних ігор є неоднорідною за рівнем досвіду, інтересами та ролями користувачів. Водночас її учасників об’єднує потреба у зручному пошуку ігор, людей та подій. Специфіка цього хобі полягає в тому, що успішна ігрова зустріч залежить від поєднання кількох факторів: наявності відповідної гри, достатньої кількості учасників, спільного часу, місця проведення та попередньої комунікації. Це створює передумови для розробки спеціалізованої системи, орієнтованої саме на координацію таких взаємодій.

1.1.2. Основні сценарії взаємодії користувачів

У межах спільноти любителів настільних ігор взаємодія користувачів зазвичай не обмежується лише фактом участі в одній ігровій зустрічі. Перед проведенням партії учасники мають знайти одне одного, обрати гру, домовитися про час і місце, визначити кількість гравців та обмінятися

додатковою інформацією. Після завершення зустрічі взаємодія також може продовжуватися через обговорення вражень, оцінювання гри, формування нових контактів або планування наступних подій. Саме тому система для такої предметної області повинна враховувати кілька основних сценаріїв поведінки користувачів.

Першим базовим сценарієм є пошук та перегляд настільних ігор. Користувач може використовувати каталог для ознайомлення з доступними іграми, їхніми характеристиками, кількістю гравців, орієнтовною тривалістю партії, описом та зображенням. Для новачків цей сценарій допомагає швидше зорієнтуватися у великій кількості ігор, а для досвідчених гравців є способом знайти конкретну гру або перевірити основні параметри перед організацією зустрічі. Додатково корисними є оцінки та відгуки інших користувачів, оскільки вони дозволяють отримати не лише формальний опис, а й практичне уявлення про ігровий досвід.

Другим важливим сценарієм є формування власної колекції ігор. Оскільки настільна гра є фізичним об’єктом, наявність гри у конкретного користувача має практичне значення для організації партій. Користувач може позначати ігри, які має у власній колекції, хоче спробувати або вже грав раніше. Така інформація надалі може використовуватися під час планування зустрічей: наприклад, організатор може обрати гру, яка вже є у когось із потенційних учасників, або зрозуміти, які ігри викликають найбільший інтерес у групи.

Окремим сценарієм є пошук інших гравців і встановлення соціальних зв’язків. Для настільних ігор важливо не лише знайти гру, а й знайти людей, з якими буде комфортно грати. Користувачі можуть переглядати профілі інших учасників, звертати увагу на їхні ігрові вподобання, місто, активність, колекцію або попередні відгуки. Можливість додавання користувачів до списку друзів спрощує подальшу комунікацію та створює основу для стабільних ігрових груп. У майбутньому цей сценарій може бути розширений

за рахунок рекомендацій гравців на основі спільних інтересів, географічної близькості або схожих ігрових колекцій.

Найбільш важливим для предметної області є сценарій організації ігрової зустрічі. Організатор створює подію, вказує її назву, дату й час проведення, місце, бажану кількість учасників, опис та гру, навколо якої планується зустріч. Інші користувачі можуть переглядати доступні події та приймати рішення щодо участі. Для цього їм важливо бачити не лише загальну інформацію про зустріч, а й склад учасників, статус події та пов’язані з нею деталі. Такий сценарій дозволяє перевести неформальні домовленості у більш структурований формат і зменшити ризик втрати важливої інформації в чатах.

Не менш важливою є комунікація між учасниками. Навіть якщо основні параметри зустрічі вже визначені, у процесі підготовки можуть виникати уточнення: хто приносить гру, чи потрібно знати правила заздалегідь, чи можна запросити ще одного учасника, чи змінюється час або місце проведення. Для цього система має підтримувати обмін повідомленнями між користувачами або в межах окремої групової розмови, пов’язаної з конкретною зустріччю. Такий підхід є зручнішим, ніж розрізнені повідомлення у сторонніх месенджерах, оскільки контекст комунікації залишається пов’язаним із подією.

Після проведення ігрової партії можливим є сценарій обміну досвідом. Користувачі можуть оцінити гру, залишити короткий відгук, поділитися враженнями або використати отриманий досвід для планування наступних зустрічей. Для спільноти це має додаткову цінність, оскільки поступово формується база користувацьких оцінок і коментарів. Вона допомагає іншим учасникам краще обирати ігри, а також підтримує активність у системі між самими зустрічами.

Таким чином, типова взаємодія користувачів у системі може бути подана як послідовність взаємопов’язаних дій: реєстрація, заповнення профілю, перегляд каталогу ігор, формування колекції, пошук інших гравців,

створення або вибір зустрічі, комунікація з учасниками та подальший обмін враженнями.

1.1.3. Проблематика існуючих методів координації та комунікації

На практиці координація зустрічей любителів настільних ігор часто відбувається за допомогою загальних інструментів комунікації: месенджерів, групових чатів, соціальних мереж, електронних таблиць або особистих домовленостей між учасниками. Такі засоби є зручними для швидкого обміну повідомленнями, однак вони не враховують специфіку організації ігрових партій. У результаті значна частина інформації зберігається неструктуровано, швидко губиться в історії листування або залежить від пам’яті окремих учасників.

Однією з основних проблем є відсутність єдиного місця для зберігання даних про майбутню зустріч. У типовому груповому чаті інформація про дату, час, місце, обрану гру, кількість учасників і додаткові умови може бути розкидана між багатьма повідомленнями. Якщо хтось із учасників приєднується до обговорення пізніше, йому доводиться перечитувати значну частину листування або повторно уточнювати деталі. Це створює зайве навантаження на організатора та збільшує ймовірність непорозумінь.

Іншою суттєвою проблемою є складність контролю складу учасників. Для багатьох настільних ігор кількість гравців є важливим параметром: деякі ігри розраховані на конкретний діапазон учасників, а частина з них найкраще працює лише за певного складу. У звичайному чаті організатору доводиться вручну відстежувати, хто точно прийде, хто ще не визначився, а хто відмовився від участі. Якщо кількість місць обмежена, виникає додаткова потреба фіксувати чергу або повідомляти користувачів про зміни.

Ще однією проблемою є те, що звичайні месенджери не пов’язують домовленість про зустріч із повною інформацією про конкретну настільну гру. Наприклад, у чаті можна написати назву гри та домовитися про час, однак учасникам усе одно доведеться окремо шукати її опис, правила, тривалість партії, рекомендовану кількість гравців або відгуки. Для нових учасників це

створює додатковий бар’єр, оскільки з повідомлень у чаті не завжди зрозуміло, наскільки гра складна та чи підходить вона для їхнього досвіду. У спеціалізованій системі зустріч може бути одразу пов’язана з карткою гри, що робить підготовку до партії зрозумілішою та зручнішою.

Окремо слід зазначити проблему пошуку нових гравців. Груповий чат добре працює для вже сформованої компанії, але є менш зручним для людини, яка тільки хоче долучитися до спільноти. Новому користувачу складно зрозуміти, які зустрічі є відкритими, який рівень досвіду очікується, хто організатор, чи є вільні місця та які ігри плануються. У таких умовах вхід до спільноти залежить не стільки від зацікавленості користувача, скільки від того, наскільки активно хтось із учасників допоможе йому зорієнтуватися.

Недоліком неформальних способів координації є і відсутність історії взаємодії в зручному вигляді. Після завершення зустрічі її результати зазвичай не фіксуються: не зберігається інформація про проведену партію, склад учасників, враження від гри або оцінки. Унаслідок цього спільнота не накопичує корисні дані, які могли б допомогти в майбутньому: які ігри користуються попитом, хто часто бере участь у зустрічах, які формати подій є найзручнішими для конкретної групи.

Таким чином, використання звичайних чатів і неспеціалізованих сервісів частково вирішує проблему комунікації, але не забезпечує повноцінної координації ігрових зустрічей. Для настільно-ігрової спільноти важливими є не лише повідомлення між користувачами, а й структуроване зберігання інформації про ігри, учасників, події, колекції та домовленості. Саме ця різниця між простим спілкуванням і організацією спільної діяльності створює потребу в окремій системі, орієнтованій на особливості настільних ігор.

1.2. Огляд та порівняльний аналіз існуючих платформ

1.2.1. Глобальні інформаційні бази та платформи

Одним із найвідоміших прикладів спеціалізованої платформи у сфері настільних ігор є BoardGameGeek [1]. Цей ресурс фактично виконує роль

великої інформаційної бази, у якій зібрано дані про настільні ігри, доповнення, авторів, видавців, роки випуску, жанри, механіки, тривалість партій, рекомендовану кількість гравців та інші характеристики. Для багатьох користувачів BoardGameGeek є основним джерелом інформації під час вибору гри, ознайомлення з відгуками або порівняння різних настільних проєктів.

Основною перевагою такої платформи є велика кількість структурованих даних. Користувач може знайти сторінку конкретної гри, переглянути її рейтинг, коментарі інших гравців, фотографії компонентів, варіанти правил, файли, обговорення та пов'язані матеріали. Це особливо корисно для досвідчених гравців, які хочуть детально оцінити гру перед купівлею або участю в партії. Також платформа дозволяє користувачам вести власні колекції, позначати ігри як зіграні, бажані або такі, що перебувають у власності.

BoardGameGeek добре вирішує завдання накопичення та пошуку інформації про настільні ігри. Завдяки великій кількості користувацьких оцінок і коментарів ресурс дає змогу отримати уявлення не лише про формальні характеристики гри, а й про її сприйняття спільнотою. Наприклад, користувач може оцінити, наскільки гра підходить для певної кількості учасників, чи є вона складною для новачків, як довго триває реальна партія та які враження вона залишає після кількох зіграних сесій.

Водночас BoardGameGeek передусім орієнтований на інформаційний аспект, а не на координацію локальних ігрових зустрічей. Його основна цінність полягає у великій базі знань про ігри, однак сценарії пошуку компанії, швидкого створення події, запрошення знайомих учасників або прив'язки зустрічі до конкретної локальної спільноти реалізовані не як головна функція платформи. Для користувача, який хоче не просто прочитати про гру, а знайти людей для партії у своєму місті або домовитися про конкретну зустріч, цього може бути недостатньо.

Ще одним обмеженням є складність інтерфейсу для нових користувачів. Через велику кількість розділів, налаштувань, статистики та додаткових

матеріалів платформа може виглядати перевантаженою для людини, яка тільки починає цікавитися настільними іграми. Такий підхід є зручним для активних учасників міжнародної спільноти, але не завжди відповідає потребам невеликої локальної групи, якій потрібен простий інструмент для організації зустрічей і підтримки комунікації.

Таким чином, BoardGameGeek

можна розглядати як сильний приклад глобальної інформаційної платформи для настільних ігор. Вона ефективно виконує функції каталогу, бази відгуків, рейтингової системи та інструменту для ведення особистої колекції. Проте для задачі координації локальних спільнот її можливості є обмеженими, оскільки основний акцент зроблено на збереженні інформації про ігри, а не на організації взаємодії між гравцями в межах конкретних подій. Саме тому при розробці власної системи доцільно врахувати переваги такого підходу, зокрема структурований каталог і користувацькі оцінки, але доповнити їх механізмами соціальної взаємодії та планування ігрових зустрічей.

1.2.2. Універсальні сервіси для організації подій

Окрім спеціалізованих інформаційних баз, для організації зустрічей часто використовуються універсальні сервіси подій. До таких рішень можна віднести Meetup [2] та Facebook Events [3]. Їхня основна ідея полягає в тому, щоб надати користувачам інструменти для створення подій, запрошення учасників, поширення інформації та пошуку заходів за інтересами або місцем проведення. Такі платформи не обмежуються настільними іграми й можуть використовуватися для лекцій, спортивних занять, професійних зустрічей, культурних подій та інших форматів дозвілля.

Перевагою універсальних сервісів є те, що вони вже мають звичну для багатьох користувачів модель взаємодії. Організатор може створити подію, додати назву, опис, дату, місце проведення, зображення та запросити інших учасників. Користувачі, у свою чергу, можуть переглядати події, підтверджувати участь, отримувати сповіщення про зміни та поширювати

інформацію серед знайомих. Для публічних заходів це є важливим, оскільки такі платформи допомагають залучати нову аудиторію та роблять подію помітнішою.

Meetup більше орієнтований на регулярні спільноти за інтересами. Користувачі можуть приєднуватися до груп, стежити за майбутніми подіями, шукати активності у своєму місті та взаємодіяти з організаторами. Для настільно-ігрових клубів це може бути корисним, якщо йдеться про відкриті зустрічі, турніри або періодичні заходи для широкої аудиторії. Натомість Facebook Events часто використовується як частина ширшої соціальної мережі, де події поширюються через сторінки, групи, особисті профілі та рекомендації друзів.

Водночас універсальний характер таких сервісів є і їхнім обмеженням. Вони надають загальні поля для опису події, але не враховують специфіку настільних ігор. Наприклад, під час створення ігрової зустрічі важливо вказати не лише дату та місце, а й конкретну гру, рекомендовану кількість учасників, орієнтовну тривалість партії, складність правил, потребу в попередній підготовці та наявність вільних місць. Універсальні сервіси не мають окремих механізмів для роботи з такими параметрами, тому організатору доводиться описувати їх вручну.

Також ці платформи зазвичай не пов’язують подію з особистими колекціями користувачів або їхнім ігровим досвідом. Для настільних ігор це важливо, оскільки вибір гри часто залежить від того, хто має фізичну копію, хто вже знайомий із правилами та хто хоче спробувати певну гру. У Meetup або Facebook Events така інформація не є частиною стандартного сценарію. Через це організатор змушений додатково уточнювати деталі в коментарях, приватних повідомленнях або сторонніх чатах.

Ще одним недоліком є недостатня зручність для невеликих приватних або напівзакритих груп. Універсальні сервіси добре підходять для анонсування подій широкій аудиторії, але не завжди є оптимальними для регулярної координації кількох знайомих гравців. Якщо група збирається

часто, їй потрібен не лише окремий опис кожної події, а й постійний контекст: попередні зустрічі, список учасників, їхні вподобання, колекції ігор, історія взаємодії та можливість швидко домовитися про наступну партію.

Отже, Meetup та Facebook Events можна розглядати як корисні інструменти для базової організації подій і залучення учасників. Вони добре вирішують загальні задачі публікації заходу, запрошення людей і поширення інформації. Проте для спільнот любителів настільних ігор їхніх можливостей недостатньо, оскільки вони не враховують ігрову специфіку, не інтегрують події з каталогом ігор, колекціями користувачів та історією ігрової взаємодії. Це свідчить про доцільність створення спеціалізованої системи, у якій організація події буде пов’язана не лише з часом і місцем, а й з конкретною грою, учасниками та потребами настільно-ігрової спільноти.

1.2.3. Локальні ініціативи та використання неспеціалізованих месенджерів

Окрім великих платформ, значна частина настільно-ігрових спільнот організовується на локальному рівні. Це можуть бути невеликі клуби, групи друзів, студентські ініціативи, тематичні Telegram [4]-канали, Discord [5]-сервери, Viber-групи або сторінки у соціальних мережах. Такі формати часто виникають природно: кілька активних учасників створюють канал або чат, запрошують знайомих, публікують оголошення про зустрічі та підтримують спілкування всередині спільноти.

Перевагою локальних ініціатив є їхня простота та доступність. Для створення групового чату не потрібно розробляти окрему систему або налаштовувати складну інфраструктуру. Більшість користувачів уже мають встановлені месенджери, тому можуть швидко приєднатися до обговорення, поставити запитання, підтвердити участь або отримати повідомлення від організатора. Для невеликих груп, де всі учасники добре знайомі між собою, такий спосіб може бути достатнім на початковому етапі.

Discord-сервери та Telegram-спільноти мають дещо ширші можливості порівняно зі звичайними чатами. У них можна створювати окремі канали для

оголошень, обговорень, пошуку гравців, обміну фотографіями або правил гри. Discord також підтримує ролі користувачів, голосові канали та структурування спільноти за темами. Завдяки цьому такі інструменти можуть бути зручними для підтримки активного спілкування, особливо якщо спільнота має постійних учасників і власних модераторів.

Разом з тим, локальні чати та сервери залишаються переважно засобами комунікації, а не повноцінними системами управління подіями. Інформація про зустрічі часто подається у вигляді повідомлень або закріплених постів, які потрібно оновлювати вручну. Якщо одночасно планується кілька ігрових партій, користувачам може бути складно відстежити, до якої саме зустрічі вони приєдналися, скільки місць залишилося, хто вже підтвердив участь і які ігри будуть доступні.

Ще однією проблемою є залежність від активності окремих організаторів. У локальних групах саме адміністратори або найбільш активні учасники зазвичай відповідають за створення оголошень, збір заявок, уточнення деталей і нагадування про події. Якщо організатор пропускає повідомлення або не оновлює інформацію вчасно, інші учасники можуть отримати застарілі або неповні дані. Для стабільної роботи спільноти потрібні інструменти, які частину цих процесів роблять більш структурованими.

Також неспеціалізовані месенджери не створюють окремої бази знань про ігри, учасників і минулі зустрічі. Користувач може написати в чаті, що має певну гру або хоче зіграти в неї, але з часом це повідомлення губиться серед інших. Так само складно швидко знайти інформацію про те, хто вже грав у конкретну гру, які були відгуки, хто готовий пояснити правила або які партії проводилися раніше. У результаті спільнота має активну комунікацію, але не накопичує структуровані дані, які могли б покращити організацію наступних подій.

Отже, локальні ініціативи та месенджери є важливими для живого спілкування всередині спільнот, однак їхні можливості обмежені, коли йдеться про системну координацію настільно-ігрової діяльності. Вони добре

підходять для швидких повідомлень, неформальних обговорень і підтримки контакту між учасниками, але не забезпечують повноцінної роботи з каталогом ігор, колекціями, статусами участі, історією подій та структурованим плануванням зустрічей. Саме тому такі інструменти доцільно розглядати не як повну альтернативу спеціалізованій платформі, а як приклад поширеної практики, недоліки якої можуть бути враховані під час проєктування власної системи.

1.3. Визначення недоліків існуючих систем та шляхів їх вирішення

1.3.1. Обмеження існуючих рішень щодо специфіки настільних ігор

Проведений огляд показує, що наявні рішення частково закривають потреби спільнот любителів настільних ігор, але зазвичай роблять це лише в межах окремих сценаріїв. Одні платформи добре працюють як джерело інформації про ігри, інші надають базові інструменти для створення подій, а месенджери забезпечують швидку комунікацію між учасниками. Проте жоден із таких підходів не поєднує в одному середовищі всі основні складові, необхідні для регулярної організації настільно-ігрових зустрічей.

Першим суттєвим обмеженням є слабкий зв’язок між каталогом ігор і плануванням подій. У спеціалізованих інформаційних базах користувач може знайти детальний опис гри, переглянути її рейтинг або додати до власної колекції. Водночас такі платформи не завжди зручно використовувати для створення локальної зустрічі навколо цієї гри. У сервісах подій, навпаки, можна вказати час і місце проведення, але інформацію про гру доводиться додавати вручну в опис. Через це користувачі змушені поєднувати кілька інструментів: один для вибору гри, інший для домовленостей, третій для подальшого спілкування.

Другим обмеженням є недостатня увага до колекцій користувачів. Для настільних ігор це має важливе значення, оскільки гра зазвичай існує як фізичний набір компонентів. Якщо певна гра є лише в одного учасника,

організація партії залежить від його присутності та готовності принести її на зустріч. Більшість універсальних платформ не враховує цей фактор. У результаті організатору доводиться окремо з’ясовувати, хто має потрібну гру, хто може її взяти із собою та чи є альтернативні варіанти.

Третім недоліком є обмежені можливості для підбору учасників за ігровими інтересами. У настільних іграх важливо враховувати не лише факт наявності вільного часу, а й рівень досвіду, бажані жанри, ставлення до складних правил, готовність грати у довгі партії або, навпаки, перевагу коротким іграм. Неспеціалізовані сервіси зазвичай не зберігають таку інформацію у структурованому вигляді. Через це пошук відповідної компанії часто відбувається вручну, через повідомлення або особисті знайомства.

Також існуючим рішенням бракує єдиного простору для комунікації, пов’язаної саме з конкретною подією. У месенджерах спілкування може бути активним, але воно не завжди відокремлене від інших тем. У сервісах подій коментарі часто виконують допоміжну роль і не замінюють повноцінного обговорення між учасниками. Для ігрової зустрічі важливо мати можливість уточнити правила, домовитися про склад учасників, обговорити зміни та зберегти цей контекст поряд із самою подією.

Ще одним обмеженням є відсутність цілісної історії взаємодії користувачів зі спільнотою. Окремі платформи можуть зберігати оцінки ігор або список відвіданих подій, але ці дані рідко поєднуються між собою. Для розвитку спільноти корисно бачити, які ігри користуються попитом, які користувачі часто беруть участь у зустрічах, які події були успішними, а які формати потребують змін. Без такої історії організація наступних зустрічей значною мірою спирається на суб’єктивні враження та ручні домовленості.

Отже, основною проблемою існуючих рішень є не повна відсутність потрібних функцій, а їхня розпорошеність між різними сервісами. Для настільно-ігрової спільноти важливо, щоб каталог ігор, колекції користувачів, соціальні зв’язки, події та комунікація працювали як частини однієї системи. Саме ця вимога відрізняє спеціалізовану платформу від набору загальних

інструментів і визначає напрям подальшого проєктування власного програмного продукту.

1.3.2. Обґрунтування доцільності розробки власної спеціалізованої вебплатформи

З огляду на виявлені обмеження існуючих рішень, розробка власної спеціалізованої вебплатформи є доцільною, оскільки вона дозволяє зосередитися саме на потребах спільнот любителів настільних ігор. На відміну від універсальних сервісів, така система може бути спроєктована не навколо абстрактної події або загального спілкування, а навколо конкретних сутностей предметної області: настільної гри, користувача, колекції, ігрової зустрічі та комунікації між учасниками.

Однією з головних переваг власної платформи є можливість об’єднати кілька важливих сценаріїв в одному середовищі. Користувач може спочатку переглянути каталог ігор, ознайомитися з їхніми характеристиками, оцінками та відгуками, після цього додати цікаві ігри до власної колекції або списку вподобань, а згодом створити зустріч, пов’язану з конкретною грою. Такий підхід зменшує потребу переходити між різними сервісами та дозволяє зберігати контекст взаємодії всередині однієї системи.

Вебформат є зручним для реалізації такої системи, оскільки не потребує встановлення окремого застосунку на пристрій користувача. Доступ до платформи може здійснюватися через браузер, що спрощує використання як на персональних комп’ютерах, так і на мобільних пристроях. Для спільнот, де учасники мають різний технічний досвід і користуються різними пристроями, це є важливою перевагою. Крім того, вебплатформа дозволяє централізовано оновлювати функціональність без необхідності окремого оновлення клієнтських застосунків.

Розробка власної системи також дає змогу врахувати логіку локальних ігрових зустрічей. У межах платформи можна передбачити створення подій із зазначенням дати, місця, кількості учасників, опису, видимості та прив’язки до конкретної гри. Це дозволяє формалізувати ті домовленості, які у звичайних

чатах часто залишаються у вигляді розрізнених повідомлень. Для організатора це спрощує управління подією, а для учасників робить інформацію про зустріч зрозумілою та доступною.

Важливою перевагою спеціалізованої платформи є можливість розвитку соціальної складової. Користувачі можуть створювати профілі, знаходити інших гравців, додавати їх до друзів, обмінюватися повідомленнями та підтримувати зв'язок після завершення окремої партії. У перспективі така функціональність може бути доповнена рекомендаціями гравців, ігор або подій на основі спільних інтересів, колекцій, міста чи попередньої активності. Це робить систему не лише інструментом планування, а й середовищем для поступового формування спільноти.

Доцільність власної розробки також пов'язана з можливістю гнучкого подальшого розширення. На початковому етапі система може містити базові функції: реєстрацію користувачів, каталог ігор, оцінки, відгуки, колекції, друзів, створення зустрічей і обмін повідомленнями. Надалі її можна доповнювати календарем подій, сповіщеннями, фільтрацією за містом, розширеними ролями організаторів, рекомендаційними механізмами, статистикою активності та іншими функціями. Такий підхід відповідає практиці поетапної розробки програмного забезпечення, коли спочатку реалізується основна цінність продукту, а додаткові можливості додаються поступово.

Отже, створення власної вебплатформи є обґрунтованим рішенням, оскільки воно дозволяє усунути розрізненість існуючих інструментів і побудувати систему, орієнтовану на реальні сценарії взаємодії любителів настільних ігор. Така платформа може поєднати інформаційні, соціальні та організаційні функції, забезпечуючи користувачам єдиний простір для пошуку ігор, планування зустрічей, комунікації та розвитку локальних спільнот.

Підсумок порівняльного аналізу з підрозділів 1.2 і 1.3 подано в таблиця 1.1. У ній зіставлено типові підходи, які вже використовують спільноти, із сценаріями з п. 1.1.2. Останній стовпець відображає не готовий продукт, а

цільовий набір можливостей розроблюваної вебплатформи згідно з обмеженнями, сформульованими в п. 1.3.1. Приклади в заголовках стовпців ілюструють кожен тип рішення; оцінювання стосується класу засобів загалом, а не окремого інтерфейсу чи редакції сервісу.

Таблиця 1.1 — Порівняння типів існуючих рішень за ключовими сценаріями предметної області

Критерій оцінювання	Інформаційні бази (BGG)	Події (Meetup, Facebook Events)	Месенджери (Telegram, Discord, Viber)	Розроблюване рішення
Перегляд каталогу ігор і їхніх характеристик	так	ні	ні	так
Ведення особистої колекції ігор	так	ні	ні	так
Оцінювання ігор і користувацькі відгуки	так	ні	ні	Так
Пошук гравців і підтримка соціальних зв’язків	частково	частково	частково	так
Організація ігрових зустрічей	частково	так	частково	так
Контроль складу учасників і статусів участі	ні	частково	ні	так
Зв’язок події з конкретною настільною грою	частково	ні	ні	так
Комунікація учасників у контексті події	ні	частково	частково	так
Орієнтація на локальну спільноту	ні	так	так	так
Зручність для новачків	частково	частково	частково	так

ВИСНОВКИ ДО РОЗДІЛУ

У розглянутих підрозділах було проаналізовано предметну область, пов’язану з організацією взаємодії спільнот любителів настільних ігор. Визначено, що цільова аудиторія такої системи є досить різноманітною: до неї належать новачки, досвідчені гравці, власники колекцій, організатори зустрічей та учасники локальних ігрових спільнот. Специфіка цього хобі полягає в необхідності узгодження не лише часу й місця зустрічі, а й вибору гри, кількості учасників, рівня підготовки та подальшої комунікації між гравцями.

Було розглянуто основні сценарії взаємодії користувачів: перегляд каталогу ігор, формування власної колекції, пошук інших гравців, організація ігрових зустрічей, обмін повідомленнями та подальше оцінювання ігор. Також встановлено, що звичайні месенджери й неформальні способи координації не забезпечують достатньої структурованості даних. Інформація про події, учасників, ігри та домовленості часто розміщується в різних каналах або губиться в історії листування.

Під час огляду існуючих рішень було визначено, що глобальні інформаційні платформи, зокрема BoardGameGeek, добре виконують роль бази знань про настільні ігри, але не орієнтовані на організацію локальних зустрічей. Універсальні сервіси подій, такі як Meetup і Facebook Events, дозволяють створювати заходи та запрошувати учасників, однак не враховують специфічні параметри настільних ігор. Локальні Telegram-, Viber- або Discord-спільноти зручні для швидкого спілкування, проте не забезпечують повноцінної роботи з каталогом ігор, колекціями, статусами участі та історією подій.

Отже, основною проблемою існуючих підходів є розрізненість функцій між різними сервісами. Для ефективної координації настільно-ігрової спільноти потрібна система, у якій каталог ігор, профілі користувачів, колекції, зустрічі, соціальні зв’язки та комунікація працюють як частини єдиного середовища. Саме це обґрунтовує доцільність розробки власної

спеціалізованої вебплатформи та створює основу для подальшого формування вимог до системи.

РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ ТА ВИБІР ТЕХНОЛОГІЙ

2.1. Проєктування архітектури вебплатформи

2.1.1. Обґрунтування клієнт-серверної архітектури

Для розробки системи було обрано клієнт-серверну архітектуру, оскільки вона дозволяє чітко розділити відповідальність між інтерфейсом користувача, серверною логікою та зберіганням даних. Клієнтська частина відповідає за відображення вебінтерфейсу, навігацію між сторінками та відправлення запитів до сервера. Серверна частина обробляє бізнес-логіку, виконує автентифікацію користувачів, перевіряє коректність даних і взаємодіє з базою даних.

Основна взаємодія між клієнтом і сервером здійснюється через REST API [6]. Такий підхід є зручним для реалізації типових операцій системи: реєстрації та входу користувача, перегляду каталогу ігор, роботи з колекціями, створення зустрічей, додавання друзів, оцінювання ігор та залишення відгуків. Дані передаються у форматі JSON, що добре підходить для сучасних вебзастосунків і спрощує інтеграцію між клієнтською та серверною частинами.

Для функцій, які потребують миттєвого обміну даними, зокрема для чату, планується окремий канал реального часу на основі WebSocket [7]. На відміну від звичайних HTTP-запитів, такий механізм дозволяє підтримувати постійне з’єднання між клієнтом і сервером та передавати нові повідомлення без перезавантаження сторінки або регулярного опитування сервера. Саме ця вимога випливає з аналізу першого розділу: координація зустрічі часто відбувається у форматі швидкого діалогу, де затримка у кілька секунд уже сприймається як незручність.

Важливою перевагою клієнт-серверної архітектури є централізований контроль доступу до даних. Користувач не взаємодіє з базою даних напряму,

а всі операції проходять через сервер, де перевіряються права доступу, автентичність користувача та коректність введеної інформації. Це особливо важливо для системи, у якій зберігаються профілі користувачів, повідомлення, списки друзів і дані про заплановані зустрічі.

Загальну структуру системи можна подати у вигляді трьох основних рівнів: клієнтського вебзастосунку, серверної частини та бази даних. Додатково система може взаємодіяти із зовнішнім сховищем зображень, яке використовується для аватарів користувачів і зображень настільних ігор.

Отже, клієнт-серверна архітектура відповідає потребам розроблюваної платформи, оскільки забезпечує структурований поділ системи, безпечну роботу з даними та можливість поєднання звичайних API-запитів із механізмами реального часу.

2.1.2. Проєктування модульної структури системи

Для спрощення розробки та подальшої підтримки систему доцільно поділити на окремі функціональні модулі. Такий підхід дозволяє ізолювати логіку різних частин платформи та зменшити зв’язність між компонентами. Кожен модуль відповідає за окрему групу задач і взаємодіє з іншими модулями через визначені сервіси та інтерфейси.

У межах розроблюваної системи можна виділити такі основні модулі:

- модуль автентифікації та користувачів, що відповідає за реєстрацію, вхід у систему, роботу з профілем і перевірку прав доступу;
- модуль каталогу настільних ігор, який забезпечує перегляд, пошук, створення та оновлення інформації про ігри;
- модуль колекцій, що дозволяє користувачам зберігати інформацію про власні ігри, бажані позиції або ігри, якими вони володіли раніше;
- модуль оцінок і відгуків, призначений для фіксації користувацького досвіду щодо конкретних ігор;
- модуль друзів, який реалізує соціальні зв’язки між користувачами;
- модуль ігрових зустрічей, що відповідає за створення подій, керування учасниками, запрошеннями та статусами зустрічей;

- модуль повідомлень, який забезпечує особисті, групові та пов’язані із зустрічами розмови.

На серверній частині такий поділ відповідає модульному підходу фреймворку NestJS [8]. У кожному модулі можуть бути визначені контролери для приймання HTTP-запитів, сервіси для реалізації бізнес-логіки, DTO для опису вхідних даних і репозиторії для роботи з базою даних. Це дозволяє не змішувати, наприклад, логіку створення зустрічі з логікою оцінювання гри або обробки повідомлень.

Модульна структура також є корисною для поступового розвитку системи. Якщо в майбутньому виникне потреба додати сповіщення, рекомендації ігор або інтеграцію із зовнішнім сховищем зображень, такі можливості можна реалізувати як окремі модулі, не змінюючи суттєво вже наявну логіку. Це робить архітектуру більш гнучкою та зручною для підтримки.

Отже, поділ системи на функціональні модулі дозволяє краще організувати кодову базу, спростити тестування окремих частин застосунку та забезпечити можливість подальшого розширення платформи.

2.1.3. Проєктування бази даних

Для зберігання даних системи обрано реляційну базу даних PostgreSQL [9]. Такий вибір є доцільним, оскільки предметна область містить багато взаємопов’язаних сутностей: користувачів, настільні ігри, колекції, оцінки, відгуки, дружні зв’язки, ігрові зустрічі, учасників подій і повідомлення. Реляційна модель дозволяє явно описати ці зв’язки та забезпечити цілісність даних.

Основною сутністю системи є користувач. З ним пов’язані профільні дані, колекція ігор, залишені оцінки та відгуки, дружні зв’язки, створені зустрічі й повідомлення. Сутність настільної гри містить назву, опис, рік видання, кількість гравців, орієнтовну тривалість партії та зображення. Між користувачами та іграми існує зв’язок через колекцію, де додатково зберігається статус гри: у власності, у списку бажаного або раніше у власності.

Для організації подій використовується сутність ігрової зустрічі. Вона містить дату, місце, назву, опис, максимальну кількість учасників, статус, видимість, організатора та, за потреби, пов’язану настільну гру. Окремо зберігається список учасників зустрічі та запрошення користувачів. Для комунікації передбачено сутності розмови, учасника розмови та повідомлення. Це дозволяє реалізувати як особисті чати, так і групові розмови або обговорення, пов’язані з конкретною ігровою зустріччю.

Для взаємодії серверної частини з базою даних використовується Prisma [10] ORM. Вона дозволяє описувати структуру даних у вигляді схеми, виконувати міграції та працювати з типізованим клієнтом у коді застосунку. Це зменшує кількість ручних SQL-запитів і спрощує підтримку моделі даних під час розвитку системи.

Отже, спроєктована модель даних відображає основні процеси предметної області: роботу з користувачами, каталогом ігор, колекціями, соціальними зв’язками, зустрічами та повідомленнями. Використання PostgreSQL і Prisma ORM забезпечує структуроване зберігання даних, підтримку зв’язків між сутностями та зручну інтеграцію з серверною частиною системи.

2.2. Технології серверної частини

2.2.1. Вибір Node.js та TypeScript

Для реалізації серверної частини системи було обрано середовище виконання Node.js [11]. Воно дозволяє створювати серверні застосунки мовою JavaScript і добре підходить для вебплатформ, у яких значна частина роботи пов’язана з обробкою HTTP-запитів, зверненнями до бази даних та обміном повідомленнями між користувачами. Завдяки асинхронній моделі виконання Node.js може ефективно працювати з великою кількістю операцій введення-виведення без блокування основного потоку виконання.

Для розроблюваної системи це є важливим, оскільки сервер має одночасно обробляти різні типи запитів: авторизацію користувачів, перегляд

каталогу ігор, створення зустрічей, роботу з колекціями, надсилання повідомлень і отримання оновлень у реальному часі. Більшість таких операцій не потребує складних обчислень, але активно взаємодіє з базою даних або мережею, тому асинхронна модель Node.js є доречною.

Разом із Node.js планується використовувати мову TypeScript [12]. Вона розширює JavaScript статичною типізацією, що дозволяє виявляти частину помилок ще на етапі розробки. Для системи з багатьма сутностями, такими як користувачі, ігри, зустрічі, повідомлення та колекції, типізація допомагає точніше описувати структуру даних і зменшує ризик некоректної передачі параметрів між модулями.

Використання TypeScript також покращує підтримуваність коду. Розробник може швидше орієнтуватися у структурах об’єктів, вхідних даних і результатах роботи сервісів. Це особливо важливо в модульній архітектурі, де окремі частини системи взаємодіють між собою через визначені типи, DTO та інтерфейси.

Отже, поєднання Node.js і TypeScript є доцільним для серверної частини розроблюваної платформи. Node.js забезпечує ефективну обробку мережевих запитів і асинхронних операцій, а TypeScript підвищує надійність, зрозумілість і контрольованість програмного коду.

2.2.2. Використання NestJS для побудови API

Для побудови серверної частини системи планується фреймворк NestJS. Він працює на платформі Node.js і надає готову структуру для створення масштабованих серверних застосунків. Основною перевагою NestJS у контексті розроблюваної платформи є модульний підхід, який безпосередньо відповідає функціональному поділу, описаному в підрозділі 2.1.2: окремі частини застосунку можуть відповідати за автентифікацію, каталог ігор, колекції, зустрічі, друзів і повідомлення без змішування їхньої логіки в одному файлі.

У NestJS логіка застосунку зазвичай поділяється на контролери, сервіси та модулі. Контролери приймають HTTP-запити від клієнтської частини та

повертають відповіді. Сервіси містять основну бізнес-логіку, наприклад створення зустрічі, додавання гри до колекції або надсилання повідомлення. Модулі об’єднують пов’язані контролери, сервіси та залежності в окремі функціональні блоки. Такий поділ робить код зрозумілішим і спрощує його подальшу підтримку.

Для опису вхідних даних у системі планується застосовувати DTO. Вони визначають структуру даних, які клієнт може передати на сервер, наприклад під час реєстрації, створення гри або оновлення профілю. Разом із бібліотекою class-validator це дозволяє перевіряти коректність даних до виконання бізнес-логіки. Для платформи, де користувачі самостійно створюють ігри, зустрічі та відгуки, така перевірка є не формальністю, а засобом захисту від некоректних або надмірно довгих записів, які могли б порушити роботу інтерфейсу або бази даних.

Ще однією перевагою NestJS є підтримка документації API через Swagger [13]. Це дозволяє автоматично формувати опис доступних маршрутів, параметрів запитів і форматів відповідей. Така документація корисна як під час розробки клієнтської частини, так і для подальшої підтримки системи, оскільки полегшує розуміння контрактів між клієнтом і сервером.

Таким чином, NestJS є доцільним вибором для серверної частини платформи, оскільки забезпечує структуровану організацію коду, підтримує модульну архітектуру, спрощує валідацію вхідних даних і дозволяє будувати зрозуміле REST API для взаємодії з клієнтським вебзастосунком.

2.2.3. Автентифікація, робота з даними та безпека

Для доступу до персоналізованих функцій системи планується автентифікація користувачів. Після реєстрації та входу користувач отримуватиме токен доступу, який надалі передаватиметься разом із запитами до захищених маршрутів API. Для цього доцільно застосувати підхід на основі JWT [14], оскільки він дозволяє серверу перевіряти особу користувача без зберігання стану сесії для кожного клієнта. Для вебплатформи з окремим клієнтом і сервером це спрощує масштабування: сервер не змушений

утримувати в пам’яті активні сесії всіх відвідувачів, а клієнт може зберігати токен локально й додавати його до кожного запиту.

Паролі користувачів не повинні зберігатися у відкритому вигляді. Перед записом до бази даних вони хешуються за допомогою алгоритму bcrypt [15]. Це зменшує ризики у випадку несанкціонованого доступу до бази даних, оскільки отримані значення не можуть бути безпосередньо використані як паролі. Під час входу в систему сервер порівнює введений пароль із збереженим хешем.

Робота з базою даних планується через Prisma ORM. Вона забезпечує типізований доступ до сутностей бази даних і дозволяє виконувати міграції під час зміни структури таблиць. Завдяки цьому модель даних системи залишається узгодженою з програмним кодом, а ризик помилок під час роботи із запитами зменшується — це особливо важливо там, де одна операція зачіпає кілька пов’язаних таблиць, наприклад додавання учасника до зустрічі або створення повідомлення в груповому чаті.

Безпека системи також залежить від перевірки прав доступу. Користувач повинен мати змогу редагувати лише власний профіль, керувати власною колекцією, створювати зустрічі від свого імені та виконувати дії лише в тих чатах або подіях, до яких він має доступ. Такі перевірки мають виконуватися на серверному рівні, оскільки клієнтська частина не може вважатися надійним джерелом контролю.

Додатково важливо перевіряти вхідні дані, які надходять від користувача. Для цього застосовуються DTO та механізми валідації, що дозволяють відхиляти некоректні або неповні запити ще до виконання основної логіки. Це допомагає зменшити кількість помилок у роботі системи та підвищує її стійкість до неправильного використання.

Отже, для забезпечення автентифікації, роботи з даними та базового рівня безпеки в системі планується поєднати JWT, bcrypt, Prisma ORM і серверну валідацію вхідних даних. Такий підхід відповідає вимогам вебплатформи, у якій обробляються облікові записи користувачів, соціальні

зв’язки, повідомлення та інформація про зустрічі, а також узгоджується з принципом централізованого контролю доступу, сформульованим у підрозділі 2.1.1.

2.3. Технології клієнтської частини

2.3.1. Вибір React та Vite

Клієнтська частина платформи має охоплювати одночасно кілька різних сценаріїв: перегляд великого каталогу ігор із фільтрами, роботу з особистою колекцією, профілем і списком друзів, календар і форми ігрових зустрічей, а також вікно повідомлень. Для такої предметної області доцільно обрати компонентний підхід, коли кожен екран збирається з окремих блоків інтерфейсу, а не як єдиний статичний документ. Саме тому для реалізації клієнта планується бібліотека React [16]: вона дозволяє повторно використовувати картки ігор, елементи навігації, форми та модальні вікна в різних розділах і змінювати лише ту частину сторінки, яка залежить від нових даних із сервера.

Додатковим аргументом є те, що інтерфейс платформи суттєво залежить від стану: після додавання гри до колекції, зміни статусу зустрічі або появи нового повідомлення користувач має бачити оновлення без повного перезавантаження сторінки. Декларативна модель React, за якої зовнішній вигляд визначається поточними даними, краще відповідає цій вимозі, ніж ручне керування вмістом HTML-документа. Для дипломного проєкту це також означає можливість поступово нарощувати функціональність: спочатку реалізувати каталог і профіль, а згодом додати календар зустрічей або чат, не перебудовуючи всю клієнтську частину з нуля.

Окремо варто обґрунтувати вибір TypeScript на клієнті. У системі багато сутностей із власними полями — гра, зустріч, учасник, повідомлення, елемент колекції — і всі вони передаються між сервером і браузером у вигляді структурованих об’єктів. Використання тієї самої мови на клієнті й сервері разом із спільним пакетом контрактів у монорепозиторії зменшує ризик того,

що інтерфейс очікуватиме одні поля, а API повертатиме інші. Для автора проєкту це важливо ще й тому, що помилки такого типу важко помітити під час ручного тестування всіх екранів, а статична перевірка типів допомагає виявити їх на етапі розробки.

Інструмент Vite [17] обрано як засіб локальної розробки та збирання клієнта з практичних міркувань. Під час проєктування інтерфейсу каталогу, форм створення зустрічі та екранів авторизації потрібен короткий цикл «змінив код — одразу побачив результат». Vite забезпечує швидке оновлення під час роботи над макетами та формує оптимізований статичний комплект файлів для подальшого розміщення поряд із серверною частиною. Для вебплатформи, до якої користувачі звертатимуться через браузер без встановлення окремого застосунку, такий підхід є достатнім і не вимагає ускладнювати проєкт мобільною чи десктопною оболонкою.

2.3.2. Маршрутизація та робота із серверним стеком

Користувач платформи переміщується між багатьма логічно відокремленими розділами: каталог, колекція, друзі, зустрічі, повідомлення, налаштування профілю. Якщо для кожного переходу браузер повністю перезавантажував сторінку, втрачалася б плавність роботи, а стан фільтрів або відкритої вкладки доводилося б відновлювати вручну. Тому для клієнта планується односторінкова модель із маршрутизацією TanStack Router [18]: кожен розділ відповідає власному шляху в URL, але загальний каркас інтерфейсу залишається незмінним.

Такий вибір пов’язаний і з предметною областю. Наприклад, у каталозі ігор користувач може застосувати фільтри за жанром, кількістю гравців або складністю; у списку зустрічей — перемикати календар і режим перегляду, обирати тиждень або фільтрувати події за видимістю. Зберігання цих параметрів у рядку запиту дозволяє повернутися до того самого вигляду списку після оновлення сторінки або надіслати посилання іншому учаснику спільноти. Для закритих розділів, як-от колекція чи повідомлення, доцільно

перевіряти авторизацію ще до відкриття маршруту, щоб не показувати порожні екрани неавторизованому відвідувачу.

Звернення до серверної частини планується централізувати в типізованому API-клієнті. У розроблюваній системі багато схожих за структурою операцій — отримати список, завантажити деталі, створити запис, оновити статус — але кожна з них має власні правила авторизації та формат відповіді. Окремі функції для ігор, зустрічей, колекції, друзів і чату, побудовані поверх спільної обгортки HTTP-запитів, дозволять не дублювати обробку помилок, додавання токена доступу й оновлення простроченої сесії в кожній сторінці. Адреса API задаватиметься змінною середовища під час збирання, що спростить подальшу публікацію системи поза локальним комп’ютером розробника.

Для керування даними, які надходять із сервера, доцільно застосувати TanStack Query [19]. Каталог ігор, список зустрічей, профіль користувача та історія повідомлень не є статичними: вони змінюються після дій користувача або інших учасників спільноти. Бібліотека кешування запитів дозволяє не завантажувати ті самі дані при кожному поверненні на сторінку, але водночас оновлювати інтерфейс після створення зустрічі, додавання гри до колекції чи зміни профілю. Для платформи, де одночасно використовуються і звичайні REST-запити, і події реального часу в чаті, такий поділ виглядає доречним: історія діалогів завантажується через API, а нові повідомлення можуть з’являтися в інтерфейсі без повторного опитування всього списку.

2.3.3. Побудова користувацького інтерфейсу

Інтерфейс розроблюваної платформи має бути не лише функціональним, а й зрозумілим для різних груп користувачів, про які йшлося в першому розділі: новачків, досвідчених гравців і організаторів зустрічей. Новачку важливо швидко знайти гру та зрозуміти, чи підходить вона за кількістю гравців і тривалістю; організатору — одним поглядом побачити склад учасників і статус події; активному учаснику — переміщатися між каталогом, чатом і календарем без втрати контексту. Тому інтерфейс планується будувати

як набір сторінок із спільним каркасом: бічна навігація, область профілю та єдиний стиль карток, форм і кнопок.

Компонентний поділ дозволяє відокремити завантаження даних від відображення. Сторінка каталогу відповідатиме за отримання списку ігор і роботу з параметрами URL, а картка гри, панель фільтрів або блок пагінації матимуть власні невеликі модулі. Такий підхід спрощує підтримку: зміни в оформленні картки гри автоматично поширюватимуться і на каталог, і на сторінку колекції, якщо там використовується той самий компонент. Для дипломного проєкту це також зменшує ризик розростання окремих файлів до важкочитаних «монолітних» сторінок.

Стилізацію планується виконувати модулями SCSS із спільними змінними теми. Платформа містить багато повторюваних елементів — картки, вкладки, поля форм, бейджі статусів зустрічей — і без єдиної системи відступів, кольорів і шрифтів інтерфейс швидко стане візуально розрізненим. Модульні стилі дозволяють прив’язати оформлення до конкретного компонента й не допустити випадкового перетину класів між різними розділами. Окремо доцільно передбачити стани завантаження, помилки та порожні списки: якщо каталог ігор ще підвантажується або користувач ще не додав жодної гри до колекції, інтерфейс має пояснити це явно, а не показувати «порожній екран».

Нарешті, форми створення зустрічі, входу, реєстрації та редагування профілю потребують зворотного зв’язку після дій користувача. Організатор повинен одразу бачити, чи прийнято подію сервером; користувач — чи вдалося увійти або чи є помилка в полі електронної пошти. Для системи координації настільно-ігрових зустрічей це безпосередньо впливає на довіру до платформи: якщо після натискання кнопки «створити зустріч» немає зрозумілої реакції, учасники знову повертатимуться до неформальних чатів. Тому клієнтська частина проєктується не лише як набір екранів, а як інтерфейс, який підтримує основні сценарії спільноти з першого розділу — від пошуку гри до домовленості про партію.

2.4. Реалізація взаємодії в реальному часі

2.4.1. Порівняння підходів до оновлення чату

У першому розділі було показано, що комунікація між учасниками ігрової зустрічі є невід'ємною частиною координації: потрібно уточнити правила, склад гравців, час, місце та наявність гри. Якщо для цього використовувати лише звичайні HTTP-запити з періодичним опитуванням сервера, браузер змушений регулярно питати «чи з'явилися нові повідомлення», навіть коли нічого не змінилося. Для невеликої групи це ще терпимо, але для платформи, де одночасно можуть бути відкриті кілька діалогів і активна переписка щодо різних зустрічей, такий підхід створює зайве навантаження і затримку між надсиланням повідомлення та його появою на екрані.

Саме тому для модуля повідомлень доцільно передбачити окремий канал реального часу на основі WebSocket. Він дозволяє серверу надіслати оновлення одразу після збереження повідомлення, а клієнту — показати його без очікування наступного циклу опитування. Для решти сценаріїв платформи, де миттєвість не є критичною — перегляд каталогу, редагування профілю, створення зустрічі, завантаження історії чату — достатньо звичайного REST API [20], ключові положення якого сформульовано у дисертаційній роботі Роя Філдінга [21]. Такий поділ відповідає логіці системи: не всі операції потребують однакового механізму доставки даних.

Варто також зважити економічну сторону рішення. Постійне опитування кожні кілька секунд означає, що навіть неактивний користувач, який просто залишив відкритою вкладку чату, генерує потік запитів до сервера. Для дипломного проєкту, який планується розміщувати на VPS із обмеженими ресурсами, це не є нейтральним фактором. WebSocket-з'єднання споживає ресурси інакше — переважно під час фактичної активності — тому для сценарію «діалог у реальному часі» воно виглядає більш збалансованим, ніж імітація миттєвості через часті HTTP-запити.

2.4.2. Використання Socket.IO для чатів

Після визначення потреби в каналі реального часу постає питання, на якому рівні його реалізовувати. Низькорівневий WebSocket дає максимальну гнучкість, але змушує самостійно проєктувати підключення користувачів, підписку на кімнати розмов, повторні спроби з'єднання та обробку розривів мережі. Для дипломного проєкту, де основна увага має бути зосереджена на предметній області — зустрічах, колекціях, профілях — доцільно обрати бібліотеку Socket.IO [22] поверх WebSocket.

Це рішення зумовлене кількома практичними міркуваннями. По-перше, у чаті платформи повідомлення мають потрапляти лише до учасників конкретної розмови, а не до всіх підключених користувачів; Socket.IO спрощує організацію таких «кімнат». По-друге, у реальних умовах не завжди гарантовано стабільну роботу WebSocket через мережеві обмеження, тоді як бібліотека може перемкнутися на резервний транспорт. По-третє, серверна частина планується на NestJS, який уже має засоби інтеграції з Socket.IO, тож не доведеться окремо поєднувати два різні стеки для HTTP і для чату. Для автора проєкту це означає менший обсяг інфраструктурного коду й більше часу на реалізацію сценаріїв, важливих для спільноти настільних ігор.

Порівняно з використанням сторонніх хмарних сервісів push-повідомлень Socket.IO залишає контроль над логікою доступу на власному сервері. Користувач може писати в чаті лише тоді, коли він є учасником розмови або зустрічі, і саме серверна частина перевіряє це правило перед приєднанням до кімнати. Для платформи, яка поєднує соціальні зв'язки та координацію подій, така централізація є важливішою, ніж економія кількох рядків коду за рахунок зовнішнього SaaS-рішення, яке вимагало б окремої інтеграції з моделлю користувачів і зустрічей.

2.4.3. Поєднання збереження повідомлень і доставки оновлень

Навіть за наявності каналу реального часу повідомлення не варто сприймати лише як тимчасову подію в браузері. Користувач може закрити вкладку, повернутися до діалогу через кілька днів або прочитати переписку з

іншого пристрою. Тому доцільно зберігати текст у базі даних через REST API, а Socket.IO використовувати для сповіщення вже підключених учасників про новий запис. Такий гібридний підхід поєднує надійність постійного сховища зі швидкою реакцією інтерфейсу для тих, хто зараз онлайн.

Для розроблюваної платформи це має пряме значення. Організатор зустрічі може надіслати уточнення щодо місця проведення, а учасники, які вже відкрили чат, побачать його майже миттєво; той, хто зайде пізніше, отримає повну історію через звичайний запит до API. Крім того, розділення ролей між збереженням і доставкою спрощує подальшу підтримку: зміни в логіці перевірки прав доступу чи формату повідомлення залишаються в REST-шарі, тоді як канал реального часу відповідає лише за оперативне оновлення екрана.

Окремо варто обґрунтувати, чому для чату не доцільно будувати окремий мікросервіс або повністю відмовлятися від HTTP. У межах дипломного проєкту система залишається єдиним застосунком із спільною базою користувачів, зустрічей і розмов; якщо збереження повідомлень винести в інший сервіс, доведеться дублювати перевірки доступу, узгоджувати транзакції та ускладнювати локальне розгортання. Натомість поєднання «REST для запису й читання історії + Socket.IO для миттєвої доставки» відповідає масштабу платформи й не перевантажує архітектуру зайвими компонентами.

Такий самий принцип можна поширити на інші сценарії, де користувач очікує швидкої реакції, але не потребує окремого протоколу для всієї системи. Наприклад, після прийняття запрошення на зустріч або зміни її статусу достатньо оновити дані через API й інвалідувати кеш на клієнті за допомогою React Query; для цього не обов’язково тримати постійне з’єднання з сервером на кожній сторінці. Таким чином, канал реального часу залишається вузькоспеціалізованим інструментом саме для чату, а не універсальним транспортом усієї платформи, що знижує складність супроводу та ризик помилок у синхронізації стану між клієнтом і сервером.

Нарешті, для модуля повідомлень важливо передбачити поведінку під час тимчасової втрати з’єднання. Якщо WebSocket розірвано, користувач не повинен втратити доступ до вже надісланих повідомлень: після відновлення підключення клієнт може повторно завантажити останню частину історії через REST і лише потім знову підписатися на оновлення. Це рішення узгоджується з тим, що в першому розділі месенджери розглядалися як зручний, але не завжди надійний канал координації; розроблювана платформа має поєднати оперативність чату зі стійкістю даних, характерною для повноцінної вебсистеми. Детальний опис реалізації цього підходу в модулі чату наведено в третьому розділі.

2.5. Обґрунтування інфраструктури, розгортання та супроводу розробки

2.5.1. Середовище розробки на основі контейнеризації

Під час створення платформи розробнику доводиться одночасно працювати з кількома взаємопов’язаними компонентами: базою даних, серверним API, вебклієнтом і змінними середовища для адрес і секретів. Якщо кожен із них налаштовувати окремо на локальному комп’ютері, легко отримати ситуацію, коли проєкт «працює в одного автора, але не відтворюється в іншому середовищі». Саме тому для розробки планується Docker [23] Compose [24] із сервісами PostgreSQL, API та вебклієнта в спільній мережі.

Такий вибір обґрунтовується не лише зручністю, а й наближенням до майбутньої публікації. Локально компоненти вже взаємодіють так, як це відбуватиметься на сервері: клієнт звертається до API, API — до бази даних, а конфігурація задається змінними середовища, а не жорстко прошитими в коді. Для дипломного проєкту це зменшує ризик того, що реалізація буде перевірена лише на машині розробника без узгодженого способу запуску всієї системи разом. Окремо контейнеризація ізолює базу даних від інших локальних

сервісів і спрощує повторне розгортання чистого середовища під час експериментів із міграціями або тестовими даними.

2.5.2. Публікація на VPS і зберігання зображень у хмарі

Після завершення розробки платформу потрібно зробити доступною для користувачів поза локальною мережею. Для дипломного проєкту доцільним виглядає розміщення на віртуальному сервері (VPS): він дає повний контроль над середовищем, дозволяє за помірну вартість обслуговувати API, статичний фронтенд і довготривалі з'єднання чату, а також не вимагає одразу переходити до складніших хмарних оркестраторів. Наразі система ще не опублікована в мережі; етапи налаштування сервера, зворотного проксі та HTTPS планується описати в четвертому розділі, коли буде наведено фактичну реалізацію розгортання.

Окремо варто обґрунтувати вибір зовнішнього сховища для зображень. У платформі передбачені аватари користувачів, ілюстрації настільних ігор і, можливо, фото до відгуків. Зберігати двійкові файли безпосередньо в реляційній базі або на диску VPS не є доцільним: це збільшує розмір резервних копій, ускладнює масштабування та змішує транзакційні дані з важким медіавмістом. Тому для зображень планується використовувати об'єктне сховище AWS S3 [25], а в базі лишати лише посилання на них. Для автора проєкту такий підхід також спрощує подальше розширення: якщо з'явиться більше ігор або активніших користувачів, навантаження на медіа не доведеться вирішувати перебудовою серверної файлової системи.

2.5.3. Організація коду, документування та контроль якості

Оскільки система складається з кількох пов'язаних частин — API, вебклієнта та спільних контрактів — доцільно тримати їх в одному монорепозиторії з менеджером пакетів pnpm. Це дозволяє узгоджено змінювати формат даних на сервері й у клієнті, не розриваючи проєкт на кілька незалежних репозиторіїв, де легко втратити синхронність. Для дипломного проєкту такий підхід має і практичний сенс: зміна поля в описі зустрічі або статусі учасника одразу вимагає оновлення і DTO на сервері, і типів у

спільному пакеті, і форми на клієнті; монорепозиторій зменшує ризик, що одна з цих частин залишиться неузгодженою під час захисту роботи.

Для супроводу історії змін планується Git. Він дає змогу фіксувати етапи розробки, повертатися до стабільних станів і відокремлювати експериментальні гілки від основної лінії проєкту. Це особливо корисно під час поетапної реалізації модулів: спочатку каталог і профіль, потім зустрічі, далі чат. Можливість відновити робочу версію після невдалого експерименту з маршрутизацією або схемою бази даних знижує вартість помилок і дозволяє сміливіше перевіряти архітектурні рішення, описані в цьому розділі.

У пояснювальній записці потрібно наочно показати архітектуру, потоки даних і взаємодію модулів. Замість окремих растрових схем, які важко підтримувати, планується використовувати Mermaid [26] — текстовий формат діаграм, що зберігається разом із проєктом і легко оновлюється при зміні структури системи. Це рішення зручне саме для дипломної роботи, де схеми мають відповідати фактичній реалізації на момент захисту: якщо в третьому розділі з’явиться новий модуль або зміниться потік автентифікації, діаграму можна оновити без перемальовування зображення в графічному редакторі.

Нарешті, для зменшення кількості типових помилок у коді планується поєднати вже обрану типізацію TypeScript із перевірками ESLint [27], єдиним форматуванням Prettier [28] і модульними тестами Jest [29] для серверної частини. Такий набір не замінює ручне тестування сценаріїв користувача, але допомагає раніше виявляти проблеми в логіці API, валідації даних і обробці помилок. Для платформи, де одночасно працюють автентифікація, соціальні зв’язки, зустрічі та повідомлення, це знижує ризик регресій під час подальшого розширення функціональності.

Окремо варто підкреслити, що інструменти контролю якості обрано не заради формального «набору технологій», а як підтримку обґрунтованої раніше архітектури. Модульні тести доцільні насамперед для серверної бізнес-логіки — створення зустрічі, перевірки прав учасника, обмеження доступу до чату — тобто там, де помилка безпосередньо впливає на коректність

предметної області. ESLint і Prettier, у свою чергу, допомагають утримувати єдиний стиль у монорепозиторії, де код писатиметься протягом тривалого періоду і може містити як серверні модулі NestJS, так і клієнтські компоненти React. Таким чином, супровід розробки розглядається не як окрема «технічна додаткова глава», а як продовження проєктних рішень, спрямованих на те, щоб платформа залишалася зрозумілою, відтворюваною та придатною до захисту як цілісна система.

ВИСНОВКИ ДО РОЗДІЛУ

У другому розділі здійснено проєктування соціальної вебплатформи для настільних ігор і обґрунтовано вибір технологій для її реалізації. Показано, що предметна область поєднує каталог ігор, особисті колекції, профілі, дружні зв’язки, ігрові зустрічі та повідомлення, тож система потребує чіткого розмежування ролей між інтерфейсом, серверною логікою та зберіганням даних, а також можливості розвивати окремі функції без надмірного ускладнення всієї кодової бази.

За архітектурною основу обрано клієнт-серверну модель. Більшість операцій планується реалізувати через REST API та обмін у форматі JSON; це охоплює реєстрацію й вхід, роботу з каталогом, колекціями, зустрічами, друзями, оцінками й відгуками. Для чату та інших сценаріїв миттєвого оновлення доцільним є окремий канал реального часу на основі WebSocket, тоді як перевірка прав доступу й цілісність даних залишаються на сервері. Модульна структура відображає основні групи задач — автентифікацію, каталог, колекції, оцінки й відгуки, друзів, зустрічі, повідомлення та медіа — і узгоджується з організацією серверного застосунку в NestJS.

Спроєктована модель даних на основі PostgreSQL враховує взаємозв’язані сутності предметної області: користувачів, ігри, колекції зі статусами володіння, зустрічі з учасниками й запрошеннями, розмови та повідомлення. Реляційний підхід дає змогу явно задати зв’язки між об’єктами й підтримувати цілісність інформації; для доступу з коду серверної частини обрано Prisma ORM як засіб опису схеми, міграцій і типізованих запитів.

Для серверної частини обґрунтовано поєднання Node.js і TypeScript: асинхронна модель виконання відповідає переважно мережевим і базовим операціям платформи, а статична типізація полегшує роботу з численними сутностями та контрактами між модулями. NestJS обрано як каркас для REST API з контролерами, сервісами, DTO, валідацією та документуванням інтерфейсу. Для захисту облікових записів передбачено JWT, хешування

паролів, серверні перевірки прав і валідацію вхідних даних до виконання бізнес-логіки.

Клієнтську частину планується побудувати на React і Vite з TypeScript і спільними контрактами монорепозиторію. Компонентна організація інтерфейсу, маршрутизація TanStack Router, централізований API-клієнт і React Query мають забезпечити навігацію між розділами, узгоджений обмін із сервером і керування серверним станом без зайвого дублювання логіки на сторінках. Оформлення передбачається через SCSS-модулі, спільні дизайн-токени та повторно використовувані елементи інтерфейсу з урахуванням станів завантаження, помилок і порожніх даних.

Для модуля повідомлень обґрунтовано відмову від постійного HTTP-опитування на користь Socket.IO поверх WebSocket із можливістю резервного транспорту, а також поєднання збереження повідомлень через REST із доставкою оновлень підключеним клієнтам. У сфері інфраструктури визначено Docker Compose для локальної розробки, публікацію на VPS, винесення зображень у AWS S3 і супровід проєкту засобами Git, pnpm, Mermaid, ESLint, Prettier і Jest. Отже, у другому розділі сформовано цілісне технічне рішення для платформи; у третьому розділі буде наведено опис реалізації обраних підходів у програмному коді та середовищі виконання.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Організація програмного проєкту та середовище розробки

У третьому розділі наведено опис фактичної реалізації соціальної вебплатформи для настільних ігор, спроєктованої у другому розділі. Програмний комплекс реалізовано як монорепозиторій із менеджером пакетів pnpm і складається з трьох основних частин: серверного застосунку (apps/api), клієнтського вебзастосунку (apps/web) та пакета спільних контрактів (packages/shared).

Структура репозиторію відображає розподіл відповідальності між компонентами. Каталог apps/api містить NestJS-застосунок із модулями предметної області, Prisma-схемою бази даних і міграціями. Каталог apps/web містить односторінковий інтерфейс на React і Vite. Пакет packages/shared експортує типи даних, узгоджені між клієнтом і сервером, що зменшує ризик розбіжності форматів JSON-відповідей.

Для локальної розробки та демонстрації системи використовується Docker Compose (файл docker-compose.yml у корені репозиторію). У спільній мережі app-network піднімаються три сервіси: PostgreSQL 16 (порт 5432), API (порт 3000) і вебклієнт (порт 5173). Після старту контейнера api автоматично виконуються команди pnpm install, prisma generate, prisma migrate deploy і запуск сервера в режимі start:dev. Таким чином відтворюється повний ланцюг «браузер — клієнт — API — база даних», описаний у підрозділі 2.1.1.

Окремо налаштовано наповнення бази тестовими даними (скрипт apps/api/prisma/seed.ts): користувачі, ігри, жанри, зустрічі, дружні зв’язки та повідомлення. Це спрощує перевірку інтерфейсу та підготовку матеріалів для розділу 4.

Зовнішнє сховище зображень (AWS S3) підключається через змінні середовища API; у локальному середовищі без налаштованих ключів AWS завантаження медіа може бути обмежене, проте архітектура передбачає зберігання в базі лише URL-адрес файлів.

На рисунку 3.1 зображено компоненти системи в режимі розробки.

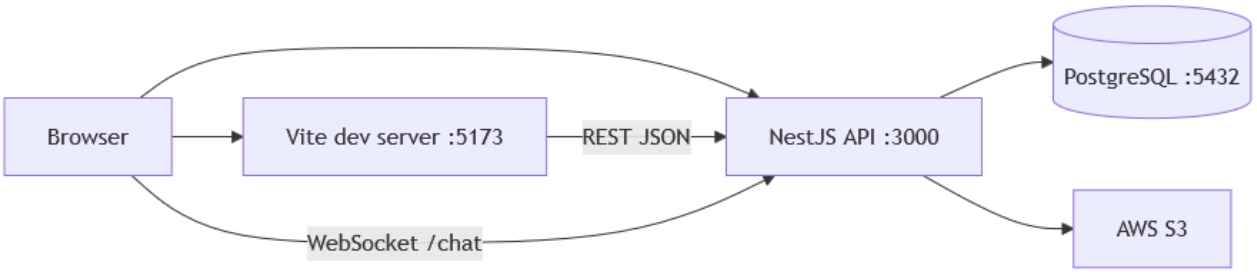


Рисунок 3.1 — Схема системи в режимі розробки

Розподіл каталогів програмного комплексу в монорепозиторії узагальнено в таблиці 3.1; у корені репозиторію також розміщені pnpm-workspace.yaml, package.json і docker-compose.yml.

Таблиця 3.1 — Склад програмного комплексу

Компонент	Технологія	Призначення
API	NestJS, Prisma, Socket.IO	REST API, WebSocket /chat, Swagger
Web	React 18, Vite, TanStack Router/Query	Користувацький інтерфейс
Shared	TypeScript	Спільні типи контрактів API
DB	PostgreSQL 16	Зберігання даних

Для супроводу коду застосовано ESLint і Prettier; збирання виконується командою pnpm build у корені монорепозиторію.

3.2. Реалізація бази даних

Логічна модель даних, описана в підрозділі 2.1.3, була трансформована у фізичну модель та реалізована в СУБД PostgreSQL за допомогою Prisma

ORM. Такий підхід дозволив описати схему бази даних декларативно у файлі apps/api/prisma/schema.prisma та генерувати типізований клієнт для доступу до даних із серверної частини . Усі зміни структури фіксуються за допомогою системи міграцій Prisma, що забезпечує узгодженість бази даних у різних середовищах розгортання .

Загальну ER-діаграму основних сутностей та зв'язків між ними наведено в Додатку В. Фізична реалізація ключових модулів системи вимагала денормалізації певних даних для оптимізації запитів та створення проміжних таблиць для зв'язків типу "багато-до-багатьох". Детальний опис структур ключових таблиць наведено нижче.

3.2.1. Користувачі та безпека

Фундаментом системи є підсистема керування користувачами та їхньою автентифікацією. Головна сутність User (таблиця 3.2) містить базову інформацію профілю та облікові дані. З міркувань безпеки оригінальні паролі не зберігаються у базі; замість цього поле passwordHash містить криптографічний хеш, згенерований алгоритмом bcrypt.

Для підтримки безперервної та безпечної роботи клієнтського застосунку реалізовано механізм ротації refresh-токенів. Логіка сесій винесена в окрему сутність AuthSession (таблиця 3.3), що дозволяє користувачу мати кілька активних сесій (наприклад, з мобільного пристрою та ПК) і забезпечує можливість їх відкликання (поле revokedAt). Процес відновлення доступу ізольовано в таблиці PasswordResetToken (таблиця 3.4) з жорстким контролем часу дії токенів (поле expiresAt) .

Таблиця 3.2 — User (користувач)

Поле	Тип	Обмеження	Опис
id	String	PK	Унікальний ідентифікатор
email	String	UNIQUE, NOT NULL	Електронна пошта
googleId	String?	UNIQUE	Ідентифікатор Google OAuth
passwordHash	String	NOT NULL	Хеш пароля (bcrypt)
displayName	String	NOT NULL	Відображуване ім’я
avatarUrl	String?		URL аватара

Продовження таблиці 3.2

Поле	Тип	Обмеження	Опис
bio	String?		Біографія
city	String?		Місто
createdAt	DateTime	DEFAULT now()	Дата створення
updatedAt	DateTime		Дата оновлення

Таблиця 3.3 — AuthSession (сесія оновлення токена)

Поле	Тип	Обмеження	Опис
id	String	PK	Ідентифікатор сесії
userId	String	FK → User	Користувач
refreshTokenHash	String	NOT NULL	Хеш refresh-токена
tokenFamily	String	NOT NULL	Сім’я токенів (ротація)
expiresAt	DateTime	NOT NULL	Термін дії
revokedAt	DateTime?		Час відкликання
replacedBySessionId	String?		Ідентифікатор сесії-наступника
createdAt	DateTime	DEFAULT now()	Дата створення
updatedAt	DateTime	NOT NULL	Дата оновлення
ip	String?		IP-адреса запиту
userAgent	String?		Заголовок User-Agent

Таблиця 3.4 — PasswordResetToken (токен скидання пароля)

Поле	Тип	Обмеження	Опис
id	String	PK	Ідентифікатор
userId	String	FK → User	Користувач
tokenHash	String	NOT NULL	Хеш токена скидання пароля
expiresAt	DateTime	NOT NULL	Час відкликання
usedAt	DateTime?		Час використання
createdAt	DateTime	DEFAULT now()	Дата створення
requestedIp	String?		IP-адреса запиту
userAgent	String?		Заголовок User-Agent

3.2.2. Каталог ігор та колекції

Ядро платформи складає каталог настільних ігор та механізм управління користувацькими колекціями. Сутність BoardGame (таблиця 3.5) зберігає як загальнодоступні метадані гри, так і технічні параметри, необхідні для фільтрації: мінімальну та максимальну кількість гравців, час партії та рівень складності. Оскільки одна гра може належати до кількох жанрів, а один жанр включати безліч ігор, цей зв'язок реалізовано через класичну проміжну таблицю BoardGameGenre (таблиця 3.6).

Інтеграція ігор із профілями реалізована через таблицю UserGame (таблиця 3.7). Вона не просто фіксує зв'язок між користувачем і грою, а й

зберігає стан володіння (статуси OWNED, WISHLIST, PREVIOUSLY_OWNED) за допомогою поля status, що є важливим для організації локальних ігрових зустрічей.

Таблиця 3.5 — BoardGame (настільна гра)

Поле	Тип	Обмеження	Опис
id	String	PK	Ідентифікатор
slug	String	UNIQUE , NOT NULL	URL-ідентифікатор
title	String	NOT NULL	Назва
description	String?		Опис
yearPublished	Int?		Рік видання
minPlayers, maxPlayers	Int?		Кількість гравців
playTimeMin, playTimeMax	Int?		Тривалість (хв)
complexity	Float?		Складність 1–5
imageUrl	String?		Зображення
externalId	String?	UNIQUE	Зовнішній ID
createdAt	DateTime	DEFAULT now()	Дата створення
updatedAt	DateTime	NOT NULL	Дата оновлення

Таблиця 3.6 — GameGenre, BoardGameGenre (жанри, зв’язок M:N)

Таблиця	Ключ	Поля
GameGenre	id PK	slug UNIQUE, name
BoardGameGenre	(gameId, genreId) PK	FK → BoardGame, GameGenre

Таблиця 3.7 — UserGame (колекція)

Поле	Тип	Обмеження	Опис
id	String	PK	Ідентифікатор
userId	String	FK → User	Ідентифікатор користувача
gameId	String	FK → Game	Ідентифікатор гри
status	CollectionStatus	NOT NULL	Статус володіння
notes	String?		Примітки
acquiredAt	DateTime?		Дата отримання

3.2.3. Соціальна взаємодія та відгуки

Соціальна складова системи базується на механізмах відгуків та формування списку контактів. Таблиці Review та Rating (таблиця 3.8) розділяють текстові рецензії та числові оцінки. Для забезпечення достовірності рейтингів на рівні бази даних накладено обмеження унікальності на пару (userId, gameId), що унеможливлює накрутку оцінок одним користувачем.

Соціальні зв'язки фіксуються в таблиці Friendship (таблиця 3.9). Вона має поля для ініціатора запиту (requesterId) та адресата (addresseeId), а поле status дозволяє відстежувати життєвий цикл запиту від очікування до підтвердження чи блокування.

Таблиця 3.8 — Review, Rating (відгуки)

Таблиця	Унікальність	Основні поля
Review	(userId, gameId)	body, imageUrls[], timestamps
Rating	(userId, gameId)	score (Int)

Таблиця 3.9 — Friendship (дружба)

Поле	Тип	Обмеження	Опис
id	String	PK	Ідентифікатор
requesterId, addresseeId	String	FK → User, UNIQUE пара	Ідентифікатори адресата запиту та його адресанта
status	FriendshipStatus	NOT NULL	Статус запиту
createdAt	DateTime	NOT NULL	Відправка запиту

3.2.4. Організація зустрічей та відгуки

Найбільш комплексною частиною бази даних є підсистема ігрових зустрічей та комунікацій. Зустріч описується сутністю PlaySession (таблиця 3.10), яка містить організаційні деталі. Учасники події та запрошення розділені на дві таблиці — PlaySessionParticipant (таблиця 3.11) та PlaySessionInvitation (таблиця 3.12). Таке розділення дозволяє гнучко керувати статусами (ParticipantStatus: підтвердив, відхилив, під питанням) без втрати історії надісланих запрошень.

Модуль повідомлень спроєктовано з урахуванням різних типів комунікації (особисті чати, групові, чати зустрічей), що відображено у полі type таблиці Conversation (таблиця 3.13). Для прив'язки чату до конкретної ігрової сесії використовується зовнішній ключ playSessionId. Зберігання повідомлень у таблиці Message (таблиця 3.15) використовує індексацію за conversationId та createdAt, що є критично важливим для забезпечення швидкого завантаження історії чатів через REST API.

Таблиця 3.10 — PlaySession (ігрова зустріч)

Поле	Тип	Обмеження	Опис
id	String	PK	Ідентифікатор
hostId	String	FK → User	Організатор
title	String	NOT NULL	Назва
description	String?		Опис сесії
gameId	String?	FK → BoardGame	Пов’язана гра
scheduledAt	DateTime	NOT NULL	Дата й час
durationMin	Int?		Тривалість сесії
locationName	String?		Назва локації
locationAddress	String?		Адреса локації
maxPlayers	Int?		Максимальна кількість гравців
status	PlaySessionStatus	NOT NULL DEFAULT SCHEDULED	Статус проведення
visibility	PlaySessionVisibility	NOT NULL DEFAULT PUBLIC	Публічне відображення
createdAt	DateTime	DEFAULT now()	Дата створення
updatedAt	DateTime		Дата оновлення

Таблиця 3.11 — PlaySessionParticipant (учасник зустрічі)

Поле	Тип	Обмеження	Опис
id	String	PK	Ідентифікатор
sessionId	String	FK → PlaySession, UNIQUE (sessionId, userId)	Ігрова зустріч
userId	String	FK → User, UNIQUE (sessionId, userId)	Користувач
Status	ParticipantStatus	NOT NULL, DEFAULT JOINED	Статус запрошення
createdAt	DateTime	DEFAULT now()	Дата створення

Таблиця 3.12 — PlaySessionInvitation (запрошення на зустріч)

Поле	Тип	Обмеження	Опис
id	String	PK	Ідентифікатор
sessionId	String	FK → PlaySession, UNIQUE (sessionId, invitedUserId)	Ігрова зустріч
invitedUserId	String	FK → User, UNIQUE (sessionId, invitedUserId)	Запрошений користувач
createdAt	DateTime	DEFAULT now()	Дата створення

Таблиця 3.13 — Conversation (розмова)

Поле	Тип	Обмеження	Опис
id	String	PK	Ідентифікатор
type	ConversationType	NOT NULL, DEFAULT DIRECT	Тип чату (особисті, група, зустріч)
title	String?		Назва

Поле	Тип	Обмеження	Опис
playSessionId	String?	FK → PlaySession, UNIQUE	Пов’язана ігрова зустріч
createdAt	DateTime	DEFAULT now()	Дата створення
updatedAt	DateTime		Дата оновлення

Таблиця 3.14 — ConversationMember (учасник розмови)

Поле	Тип	Обмеження	Опис
id	String	PK	Ідентифікатор
conversationId	String	FK → Conversation, UNIQUE (conversationId, userId)	Розмова
userId	String	FK → User, UNIQUE (conversationId, userId)	Користувач
joinedAt	DateTime	DEFAULT now()	Дата приєднання
lastReadAt	DateTime?		Час останнього прочитання

Таблиця 3.15 — Message (повідомлення)

Поле	Тип	Обмеження	Опис
id	String	PK	Ідентифікатор
conversationId	String	FK → Conversation, NOT NULL	Розмова
senderId	String	FK → User, NOT NULL	Відправник
body	String	NOT NULL	Текст повідомлення
createdAt	DateTime?	DEFAULT now()	Дата створення

3.3. Реалізація серверної частини

Серверна частина побудована на NestJS і організована за шаровою схемою, узгодженою з модульною структурою з підрозділу 2.1.2. У кожному функціональному модулі виділено чотири шари: presentation (контролери, DTO, WebSocket-шлюз), application (сервіси сценаріїв), domain (типи та правила предметної області) та infrastructure (репозиторії Prisma, обгортки зовнішніх сервісів). Контролери звертаються лише до сервісів застосунку, а сервіси — до репозиторіїв; доменний шар не залежить від NestJS і Prisma, що дозволяє замінювати окремі компоненти інфраструктури без впливу на бізнес-логіку.

Кореневий AppModule (apps/api/src/app.module.ts) підключає глобальний ConfigModule, PrismaModule і сім доменних модулів: AuthModule, GamesModule, CollectionsModule, FriendsModule, MeetupsModule, ChatModule, MediaModule. Глобально зареєстровано JwtAuthGuard (захист маршрутів за

замовчуванням, обхід через декоратор @Public()) та ThrottlerGuard з обмеженням 120 запитів за 60 секунд. Валідація DTO виконується глобальним ValidationPipe з пакета class-validator з опцією whitelist; документація API формується автоматично з декораторів @nestjs/swagger і доступна за маршрутом /docs. Розподіл відповідальності між класами наведено в таблиці 3.16.

Таблиця 3.16 — Модулі серверної частини

Модуль	Сервіс застосунку	Контролери	Інфраструктура
auth	AuthApplicationService, WsSocketAuthService	AuthController, UsersController	PrismaAuthUsersRepository, PrismaAuthSessionsRepository, BcryptPasswordHasher, JwtAccessTokenIssuer, RefreshTokenService, AuthCookieService
games	GamesApplicationService	GamesController, GameReviewsController, GameRatingsController	PrismaBoardGamesRepository, PrismaGameReviewsRepository, PrismaGameRatingsRepository
collections	CollectionsApplicationService	CollectionsController	PrismaUserGamesRepository
friends	FriendsApplicationService	FriendsController	PrismaFriendshipsRepository
meetups	MeetupsApplicationService	MeetupsController	PrismaPlaySessionsRepository
chat	ChatApplicationService	ChatController, ChatGateway	PrismaChatRepository, ChatBroadcastService
media	MediaApplicationService	MediaController	S3MediaStorageService

3.3.1. Модуль автентифікації

Центральний клас AuthApplicationService реалізує реєстрацію, вхід за паролем, оновлення та відкликання сесій, вхід через Google OAuth і редагування власного профілю. Паролі обробляються класом BcryptPasswordHasher з кількістю раундів 12 і зберігаються лише у полі passwordHash; повідомлення про помилки на маршрутах /auth/login та

/auth/register умисно не розрізняють «неправильний пароль» і «немає такого користувача», аби не давати підказку при підборі облікового запису.

Видача токенів реалізована за схемою «короткий JWT + опаковий refresh-токен». Access-токен зі строком життя 15 хвилин формує JwtAccessTokenIssuer і містить поля sub, email, sid (ідентифікатор сесії). Refresh-токен генерується класом RefreshTokenService як 48 випадкових байтів у форматі base64url; у таблицю AuthSession записується лише його SHA-256-хеш, що захищає від використання у разі витоку дампа бази. Сесії об’єднано в «сім’ї» через поле tokenFamily: під час оновлення попередня позначається revokedAt, а у разі повторного використання вже застарілого токена сервер відкликає всю сім’ю. Refresh-токен передається у HttpOnly-кукі [30] через AuthCookieService, що не дає до нього доступу клієнтському JavaScript.

Інтеграцію з Google реалізовано через @nestjs/passport і стратегію passport-google-oauth20: метод loginWithGoogle знаходить або створює користувача за googleId/email і повертає таку саму пару токенів. Відновлення пароля використовує окрему сутність PasswordResetToken (зберігається лише SHA-256-хеш, поле usedAt блокує повторне застосування). Перевірка прав на HTTP-маршрутах виконується JwtStrategy через Passport.js [31]; для WebSocket-з’єднань паралельно використовується WsSocketAuthService, що верифікує той самий JWT із поля auth.token у handshake-події Socket.IO. Контролери модуля розділено за зоною відповідальності: AuthController обслуговує власну автентифікацію та профіль (/auth/*), а UsersController — публічні картки інших користувачів (GET /users/:id).

3.3.2. Модуль каталогу ігор, відгуків і оцінок

GamesApplicationService поєднує операції з каталогом, відгуками та оцінками. Метод listGames реалізує комбінований пошук за назвою, фільтрацію за жанрами (через таблицю BoardGameGenre), діапазоном часу гри, інтервалами складності та сортування за назвою або роком; параметри page і limit транслюються у skip/take Prisma, а у відповіді разом із даними

повертається мета-об’єкт {total, page, limit} для побудови пейджера на клієнті. Метод getGameBySlug повертає детальну картку гри з агрегатами оцінок.

Відгуки винесено в окремий контролер GameReviewsController. Унікальність @@unique([userId, gameId]) у схемі бази гарантує правило «один відгук від одного користувача»: у разі дубля репозиторій кидає DuplicateGameReviewError, яку сервіс перетворює у ConflictException. Оцінки в окремому контролері GameRatingsController зберігаються через метод upsertRating, що уникає потреби явно розрізняти «створити» і «оновити» на стороні клієнта.

3.3.3. Модуль особистих колекцій

CollectionsApplicationService керує сутністю UserGame. Метод addToCollection виконує upsert за парою (userId, gameId) — повторний виклик не створює дублікат, а оновлює статус (OWNED, WISHLIST, PREVIOUSLY_OWNED). Метод updateCollectionItem використовує підхід «часткового патчу»: у Prisma передаються лише ті поля, які не undefined, що дозволяє клієнту надсилати PATCH-запити з мінімально необхідним набором полів. Допоміжний myEntryForGame повертає поточний запис для одної гри та використовується на сторінці деталей, щоб одразу показати кнопку відповідної до поточного статусу дії.

3.3.4. Модуль соціальних зв’язків

FriendsApplicationService реалізує дружні запити на основі сутності Friendship зі статусами PENDING, ACCEPTED, BLOCKED і направленістю requesterId / addresseeId. Метод discover, окрім списку профілів, обчислює стан стосунків функцією relationshipFromRow і повертає значення none, outgoing_pending, incoming_pending, friend або blocked — клієнт на основі цього одразу показує доречну кнопку без додаткових запитів.

Логіку життєвого циклу запиту реалізовано так: повторне надсилання «у відповідь» на чужий вхідний запит автоматично переводить його в ACCEPTED (взаємний інтерес), що відповідає типовій поведінці соціальних мереж і прибирає зайвий крок підтвердження. Метод areAcceptedFriends

використовується модулями чату та зустрічей для перевірки доступу до direct-розмови чи приватної події з видимістю FRIENDS; завдяки централізації цієї перевірки в одному класі правила лишаються консистентними.

3.3.5. Модуль ігрових зустрічей

MeetupsApplicationService охоплює створення, редагування, скасування зустрічей, приєднання та запрошення. Метод updateMeetup застосовує частковий патч і працює лише для зустрічей зі статусом SCHEDULED; редагування скасованих або завершених подій блокується, аби уникнути неузгодженого стану у вже надісланих запрошеннях і прив’язаних чатах.

Метод joinMeetup концентрує усі правила доступу. Перевіряється послідовність умов: статус зустрічі, факт організаторства, видимість (для FRIENDS — наявність дружнього зв’язку, для INVITE_ONLY — індивідуального запрошення в PlaySessionInvitation), відсутність попереднього приєднання та порогове значення maxPlayers. Лише після проходження всіх перевірок створюється запис PlaySessionParticipant. Метод leaveMeetup забороняє організатору залишити власну зустріч, направляючи його до скасування. Перегляд приватних зустрічей реалізовано через Prisma PlaySessionsRepository.findManyVisibleToUser, який враховує дружбу та запрошення; неавторизований доступ повертає 404 (а не «forbidden»), щоб не розкривати факт існування приватної події стороннім користувачам.

3.3.6. Модуль повідомлень

Модуль chat реалізує комбінований сценарій «REST для історії — WebSocket для оновлень», обґрунтований у підрозділі 2.4.3. ChatApplicationService підтримує три типи розмов (DIRECT, GROUP, SESSION). Створення direct-розмови обумовлено наявністю дружнього зв’язку, груповий чат вимагає, аби всі учасники були друзями ініціатора. Перевірки доступу до сторонніх розмов завжди повертають 404 за відсутності членства, тож сам факт існування чату не розкривається стороннім користувачам.

Надсилання повідомлення (sendMessage) перевіряє членство відправника, створює запис Message і викликає ChatBroadcastService.emitNewMessage. Останній надсилає подію message:new у кімнату виду conv:<conversationId> сервера Socket.IO. ChatGateway з namespace /chat обслуговує два повідомлення — join і leave (приєднати / залишити кімнату відповідної розмови); автентифікація сокета виконується через WsSocketAuthService у хуці handleConnection. Завдяки тому, що кімнати названі за ідентифікатором розмови, broadcast потрапляє лише до релевантних слухачів і не вимагає обходу списку учасників.

3.3.7. Модуль роботи з медіа

MediaApplicationService інкапсулює завантаження зображень: аватарів користувачів, фото настільних ігор і зображень до відгуків. Доступ до S3-сумісного сховища ізольовано в єдиному класі S3MediaStorageService, що дозволяє при необхідності замінити провайдера без зміни інших модулів. Файли розміщуються в трьох логічних каталогах (avatars, game-photos, review-images); у базі зберігаються лише URL-адреси, тому об’єкти Prisma залишаються легкими і не несуть бінарних даних.

3.4. Реалізація клієнтської частини

Клієнтський застосунок реалізовано як односторінкову програму (SPA) на React 18 із збиранням за допомогою Vite. Точка входу apps/web/src/main.tsx ініціалізує глобальні провайдери (QueryClientProvider з TanStack React Query, AuthProvider, ToastProvider, RouterProvider) і синхронізує тему інтерфейсу зі сховищем браузера ще до першого рендеру, щоб уникнути «миготіння» між світлою і темною схемами.

3.4.1. Маршрутизація та доступ

Маршрутизація винесена у файл apps/web/src/router.tsx і будується на TanStack Router з типізованою валідацією пошукового рядка та хуком beforeLoad для асинхронної підготовки. Усі сторінки є дочірніми до спільного rootRoute з компонентом RootLayout (бічна навігація, основна область,

модальне вікно автентифікації), а сама сторінка рендериться через Outlet — переходи не призводять до перерендерингу каркасу. Захищені маршрути використовують функцію requireAuthOrRedirect: вона дочікується завершення стартового завантаження автентифікації, перевіряє access-токен і за його відсутності виконує редирект на /games із намір-прапором показати форму входу. Перелік маршрутів подано в таблиці 3.17.

Таблиця 3.17 — Маршрути клієнтської частини

URL	Компонент сторінки	Призначення	Авторизація
/	redirect → /games	Стартова точка	ні
/home	HomePage	Головна сторінка	ні
/games	GamesListPage	Каталог настільних ігор із фільтрами	ні
/games/$slug	GameDetailPage	Деталі гри, відгуки, оцінки	ні
/collection	CollectionPage	Власна колекція з фільтром статусу	так
/profile, /settings	ProfilePage, SettingsPage	Профіль та налаштування	так
/u/$userId	PublicUserPage	Публічна сторінка користувача	ні
/friends	FriendsPage	Пошук друзів, запити, список	Так
/meetups	MeetupsListPage	Список зустрічей (календар і список)	так
/meetups/new, /meetups/$meetupId/edit	MeetupNewPage, MeetupEditPage	Створення та редагування зустрічі	так
/meetups/$meetupId	MeetupDetailPage	Деталі зустрічі	так
/meetups/$meetupId/invite	MeetupInvitePage	Надсилання запрошень	так
/messages, /messages/$conversationId	MessagesChatPage	Список розмов і конкретна розмова	так

Маршрут /messages у beforeLoad виконує запит fetchConversations і за наявності розмов виконує редирект на найсвіжішу через pickLatestConversation — користувач не бачить порожнього екрана зі списком одразу після переходу.

3.4.2. Шар API та керування серверним станом

Звернення до API централізовано в каталозі apps/web/src/api (auth, games, collection, friends, meetups, chat, media, users) поверх спільної обгортки apiFetch (apps/web/src/lib/api.ts). Вона інкапсулює формування базової адреси з VITE_API_URL, заголовок Authorization, серіалізацію JSON та обробку помилок через ToastProvider. На статус 401 викликається tryRefreshAccessToken, який надсилає POST /auth/refresh з credentials: 'include' (refresh-токен передається у HttpOnly-кукі); у разі успіху вихідний запит автоматично повторюється. Паттерн «одинокого» refreshPromise не дає декільком одночасним запитам ініціювати кілька оновлень — лише перший викликає мережевий запит, інші чекають його результату.

Серверний стан керується TanStack React Query; усі ключі кешу зібрані в централізованому файлі apps/web/src/lib/query-keys.ts, що робить інвалідацію кешу передбачуваною. Базовий staleTime запитів — 30 секунд, що зменшує кількість зайвих звернень при перемиканні між сторінками. Access-токен з міркувань безпеки не зберігається у localStorage — він живе лише у JavaScript-змінній auth-session.ts і відновлюється при кожному перезавантаженні через POST /auth/refresh (механізм startAuthBootstrap в AuthProvider).

3.4.3. Декомпозиція сторінок і компонентів

Усі сторінки розміщені в каталозі apps/web/src/pages/<feature>; кожна тека містить файл сторінки <feature>-page.tsx і теку components/ з підрозбиттям на самостійні компоненти. Такий підхід запобігає появі великих монолітних файлів: GamesListPage делегує більшість UI-роботи компонентам CatalogSearchShell (пошук), CatalogFiltersPanel (жанри, кількість гравців, тривалість, складність) і CatalogGameCard (картка гри з рейтингом і меню швидких дій). Стан фільтрів повністю зберігається в URL завдяки validateSearch, тому користувач може поділитися посиланням на конкретну вибірку, а кнопки браузера «вперед/назад» працюють у природний спосіб.

Сторінка зустрічей MeetupsListPage підтримує два режими перегляду — календарний і списковий, які перемикаються параметром URL view.

Створення та редагування зустрічей використовують одну форму MeetupForm з прапором режиму, що зменшує дублювання коду й гарантує консистентність інтерфейсу. Сторінка чату MessagesChatPage побудована як тришпальтовий макет: ChatConversationsRail (список розмов), ChatThread (стрічка повідомлень з полем введення та запрошенням друзів) і ChatContextPanel (учасники або деталі зустрічі для розмов типу SESSION). Адаптивна верстка перемикає шпальти через CSS-класи залежно від поточного маршруту.

3.4.4. Реальний час у клієнті

Інтеграцію з WebSocket-шлюзом інкапсульовано у файлі apps/web/src/lib/chat-socket.ts. Функція getSharedChatSocket повертає єдиний на весь застосунок socket.io-client до простору /chat з поточним JWT; при зміні токена попередня сесія коректно закривається, що попереджає накопичення дублікатів WebSocket. Компонент ChatThread підписується на подію message:new з фільтром за conversationId, оновлює кеш React Query через queryClient.setQueryData й автоматично надсилає join після кожного перепідключення; при розмонтуванні відписується через leave і off. Завдяки цьому браузер отримує оновлення лише для активної розмови, без «зомбі-підписок».

Архітектуру компонентів клієнта наведено на рисунку 3.2.

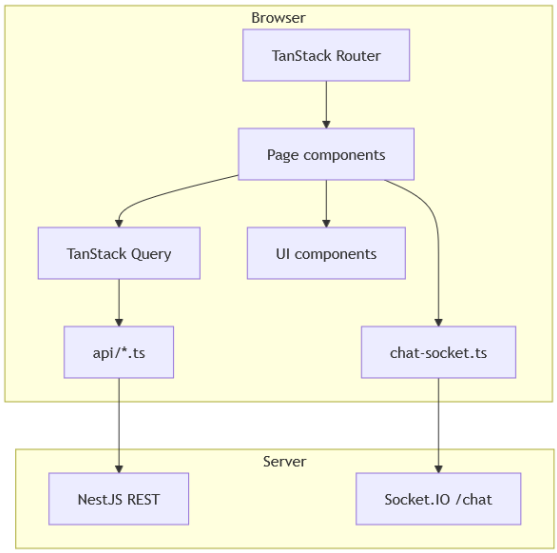


Рисунок 3.2 — Діаграма компонентів клієнтської частини

3.5. Реалізація пакета спільних контрактів

Пакет @boardgame/shared (packages/shared/src/index.ts) реалізує проміжний шар між серверним і клієнтським застосунками — типобезпечне визначення контрактів JSON-обміну. Він не залежить від NestJS, Prisma чи React; дати подаються як ISO-рядки, перелічення — як `as const` масиви рядкових літералів з виведенням типу через `(typeof X)[number]`. Це дає компілятору TypeScript роль єдиного «контракту верхнього рівня» між сервером і клієнтом, а на рівні виконання дозволяє повторно використовувати масиви значень як джерело для випадаючих списків і фільтрів.

Типи згруповано за модулями: пагінація (PaginatedMeta, PaginatedResponse), автентифікація та профіль (AuthUser, PublicUserCard, PublicProfileSummary, AuthSuccessResponse), каталог ігор (GameListItem, GameDetail, GameReview, відповідні Payload-типи), особиста колекція (CollectionStatus, CollectionEntry), соціальні зв’язки (FriendshipRelationship, DiscoverUserRow, FriendConnection), повідомлення (ConversationType, MessageView, ConversationListItem, ConversationMessages) та ігрові зустрічі (PlaySessionVisibility, PlaySessionStatus, MeetupListItem, MeetupDetail, CreateMeetupPayload, PatchMeetupPayload). На сервері ці типи використовуються у сигнатурах сервісів застосунку, а перетворення з форми Prisma в контрактну форму виконується в маперах інфраструктурного шару (наприклад, board-game-record.mapper.ts); на клієнті — у явних типах повернення функцій з apps/web/src/api/*.ts.

Цикл узгодження змін такий: тип оновлюється в packages/shared → серверні мапери приводяться до нової форми → клієнтські api-функції та компоненти, що споживають поле, оновлюються відповідно. Невідповідність на будь-якому етапі виявляється компілятором ще до запуску, що мінімізує ризик розбіжностей. Оскільки пакет не містить виконуваного коду, його підключення в монорепозиторії через pnpm-workspaces не вимагає окремого етапу публікації, а збирання відбувається миттєво.

ВИСНОВКИ ДО РОЗДІЛУ

У третьому розділі наведено реалізацію соціальної вебплатформи для настільних ігор, спроєктованої у попередньому розділі. Програмний комплекс оформлено як монорепозиторій pnpm із трьох частин: серверного застосунку на NestJS, клієнтського застосунку на React і Vite та проміжного пакета спільних контрактів. Спільне середовище розробки відтворюється через Docker Compose: у єдиній мережі підіймаються контейнери PostgreSQL, API та вебклієнта, що дозволяє відтворити повний ланцюг «браузер — клієнт — API — база даних» на будь-якому комп'ютері без ручного налаштування залежностей.

Модель даних, описана у другому розділі, реалізована у PostgreSQL за допомогою Prisma ORM. Декларативна схема у файлі schema.prisma описує сутності User, BoardGame, UserGame, Friendship, PlaySession, Conversation, Message та допоміжні таблиці; зміни структури версіонуються міграціями prisma migrate. Обмеження унікальності, відносини між таблицями та поведінка під час видалення (onDelete) задані безпосередньо у схемі, тому правила цілісності забезпечуються базою, а не лише кодом застосунку. ER-діаграму основних сутностей розміщено в додатку В, а опис полів — у відповідних таблицях підрозділу 3.2.

Серверну частину організовано за шаровою схемою з поділом на presentation, application, domain та infrastructure. Кореневий AppModule об'єднує сім доменних модулів — auth, games, collections, friends, meetups, chat, media — і реєструє глобальні фільтри, ValidationPipe для DTO та обмеження частоти запитів. Окремої уваги в реалізації потребували перевірки прав доступу: схема «короткий JWT плюс опаковий refresh-токен» із HttpOnly-кукі та хешем токена в базі, авто-прийняття зустрічних запитів дружби, послідовні перевірки доступу до зустрічей з урахуванням видимості та кількості учасників, а також єдина точка перевірки членства для приватних розмов. Документація REST API формується автоматично з декораторів

Swagger, що утримує опис інтерфейсу синхронним зі справжньою поведінкою сервера.

Клієнтський застосунок реалізовано як односторінкову програму з типізованою маршрутизацією на TanStack Router і керуванням серверним станом через TanStack React Query. Звернення до API централізовано в шарі apps/web/src/api поверх спільної обгортки apiFetch, яка прозоро виконує оновлення сесії за статусом 401 і повідомляє про помилки через єдиний механізм сповіщень. Декомпозиція за принципом «сторінка — складові компоненти» (наприклад, GamesListPage із CatalogSearchShell, CatalogFiltersPanel і CatalogGameCard) утримує файли невеликими, а збереження стану фільтрів у параметрах URL робить навігацію передбачуваною та зручною для надсилання посилань між користувачами.

Узгодженість обміну даними між клієнтом і сервером забезпечується пакетом @boardgame/shared із незалежними від фреймворків типами TypeScript. Зміна формату відповіді або запиту фіксується спочатку в пакеті, далі — у маперах серверної інфраструктури та функціях клієнтського API; неузгодженість виявляється компілятором ще до запуску, що мінімізує ризик розбіжностей. Для модуля повідомлень реалізовано комбіновану модель: історія завантажується через REST, а оновлення надсилаються підключеним учасникам кімнати через Socket.IO у просторі імен /chat, що дозволяє обійтись без періодичного опитування сервера й одночасно зберігати повідомлення в базі для подальшого перегляду.

Таким чином, у третьому розділі програмні рішення, обґрунтовані під час проєктування, доведено до робочого прототипу: реалізовано базу даних, серверну та клієнтську частини, спільні контракти й канал реального часу. У наступному розділі буде наведено результати модульного й функціонального тестування, порядок розгортання системи на VPS, а також демонстрацію роботи інтерфейсу користувача.

РОЗДІЛ 4. ТЕСТУВАННЯ, РОЗГОРТАННЯ ТА ВИКОРИСТАННЯ СИСТЕМИ

У четвертому розділі наведено результати перевірки розробленої соціальної вебплатформи для настільних ігор, порядок її публікації на віртуальному сервері (VPS) та інструкцію з використання основних функцій інтерфейсу. Матеріал розділу спирається на реалізацію з розділу 3 і на вимоги, сформульовані в розділі 1.

4.1. Тестування програмного забезпечення

4.1.1. Стратегія тестування

Перевірка якості розробленої системи поєднує автоматизоване модульне тестування серверної бізнес-логіки (Jest у apps/api), ручне функціональне тестування інтерфейсу в браузері, перевірку REST-контракту через Swagger UI та контрольний перелік безпекових властивостей. Основна увага зосереджена на критичних правилах предметної області (автентифікація, обмеження учасників зустрічей, доступ до приватних розмов), оскільки клієнтська частина значною мірою делегує валідацію серверу. Інтеграційні та навантажувальні тести в межах дипломного прототипу не виконуються.

4.1.2. Модульне тестування серверної логіки

Модульні тести розміщено поруч із тестованим кодом у файлах із суфіксом .spec.ts і запускаються командою pnpm --filter api test з кореня монорепозиторію. На момент підготовки розділу виконано 8 тестів у 4 test suites; усі завершилися успішно. Звіт у терміналі наведено на рисунку 4.1.

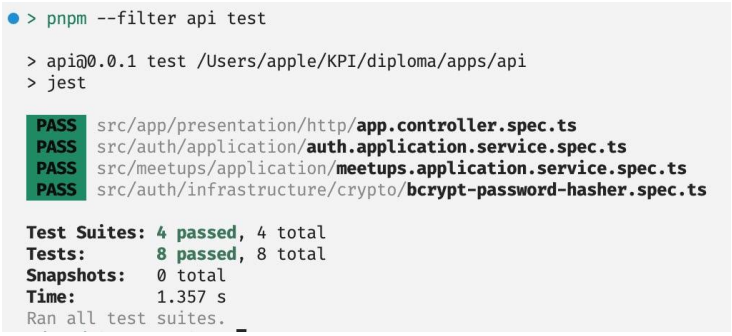


Рисунок 4.1 — Результат виконання модульних тестів

Тестами охоплено: клас BcryptPasswordHasher (коректність hash і verify пароля з cost factor c = 12); клас MeetupsApplicationService — метод joinMeetup із трьома сценаріями (відмова при заповненій зустрічі, відмова для INVITE_ONLY без запрошення, успішне приєднання при вільних місцях); клас AuthApplicationService (ротація refresh-сесії та відмова некоректного формату токена); контролер AppController (smoke-перевірка доступності API). Тести ізольовані від бази даних через jest.Mocked для методів репозиторіїв. Фрагмент тесту переповненої зустрічі наведено в рисунку 4.2.

```
it('rejects join when meetup is full', async () => {
  playSessionsRepository.getSessionMeta.mockResolvedValue({
    hostId: 'host-1',
    visibility: 'PUBLIC',
    maxPlayers: 2,
    status: 'SCHEDULED',
    scheduledAt,
  });
  playSessionsRepository.findParticipantStatus.mockResolvedValue(null);
  playSessionsRepository.countJoinedParticipants.mockResolvedValue(1);

  await expect(service.joinMeetup(userId, sessionId)).rejects.toThrow(
    ConflictException,
  );
  expect(playSessionsRepository.addParticipant).not.toHaveBeenCalled();
});
```

Рисунок 4.2 — Скріншот фрагмента unit-тесту joinMeetup при заповненій зустрічі (apps/api/src/meetups/application/meetups.application.service.spec.ts)

Перелік модульних тестів узагальнено в таблиці 4.1.

Таблиця 4.1 — Модульні тести серверної частини (Jest)

Test suite	Тест	Що перевіряє
BcryptPasswordHasher	hash and verify succeed	Коректність hash/verify
	verify fails for wrong password	Відмова при неправильному паролі
MeetupsApplicationService	rejects join when full	Ліміт учасників (maxPlayers)
	rejects INVITE_ONLY	Доступ за запрошенням
	allows join when capacity	Успішне приєднання
AuthApplicationService	refresh rotates session	Ротація refresh-сесії
	refresh rejects invalid	Відмова некоректного токена
AppController	should return Hello World	Доступність API

4.1.3. Функціональне тестування інтерфейсу

Ручне функціональне тестування виконується в браузері на локальному середовищі (адреса http://localhost:5173) або на опублікованому VPS. Перед сеансом застосовується скрипт apps/api/prisma/seed.ts, що створює тестових користувачів, ігри, зустрічі, дружні зв’язки та повідомлення.

Тест-кейси сформовані за принципом «один сценарій — один ідентифікатор ТС-XX» і прив’язані до модулів системи та сценаріїв розділу 1. Перевірено дванадцять кейсів ТС-01–ТС-12, що охоплюють реєстрацію, вхід, пошук у каталозі, роботу з колекцією, створення зустрічі, приєднання та переповнення (ТС-05), запит у друзі, обмін повідомленнями в чаті, контроль доступу до сторонніх розмов, редагування профілю та завантаження аватара. Покрокові сценарії та очікувані результати кожного кейсу демонструються у підрозділі 4.3 разом із відповідними скріншотами інтерфейсу.

Як приклад правил предметної області, що перевіряються інтерфейсом, на рисунку 4.3 наведено сторінку заповненої зустрічі: клієнт відображає кнопку приєднання у заблокованому стані з написом «Full», а серверна перевірка дублюється unit-тестом (рисунок 4.2) і повертає HTTP-код 409 Conflict у разі прямого виклику API.

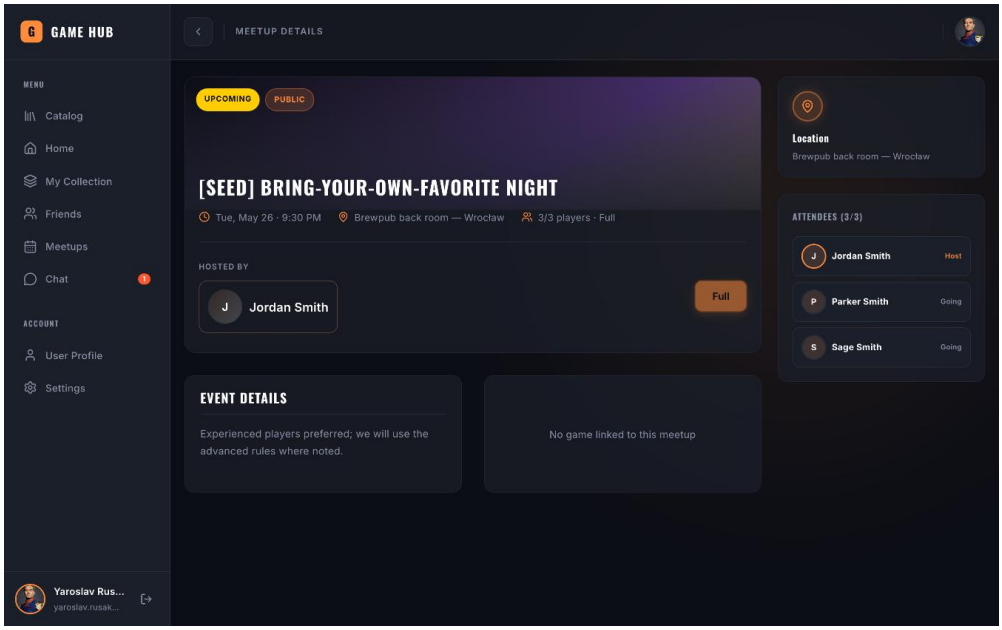


Рисунок 4.3 — Сторінка заповненої зустрічі із заблокованою кнопкою приєднання

4.1.4. Тестування REST-контракту через Swagger UI

Документація REST API формується автоматично з декораторів @nestjs/swagger і доступна за маршрутом /api. У межах перевірки контракту виконано авторизовані запити до груп маршрутів /auth/*, /games, /me/collection, /meetups, /friends, /chat: контролюється HTTP-код відповіді, структура JSON відповідно до типів пакета @boardgame/shared та повідомлення про помилки валідації (400) і доступу (401, 403, 404, 409). Приклад запиту GET /games наведено на рисунку 4.4.

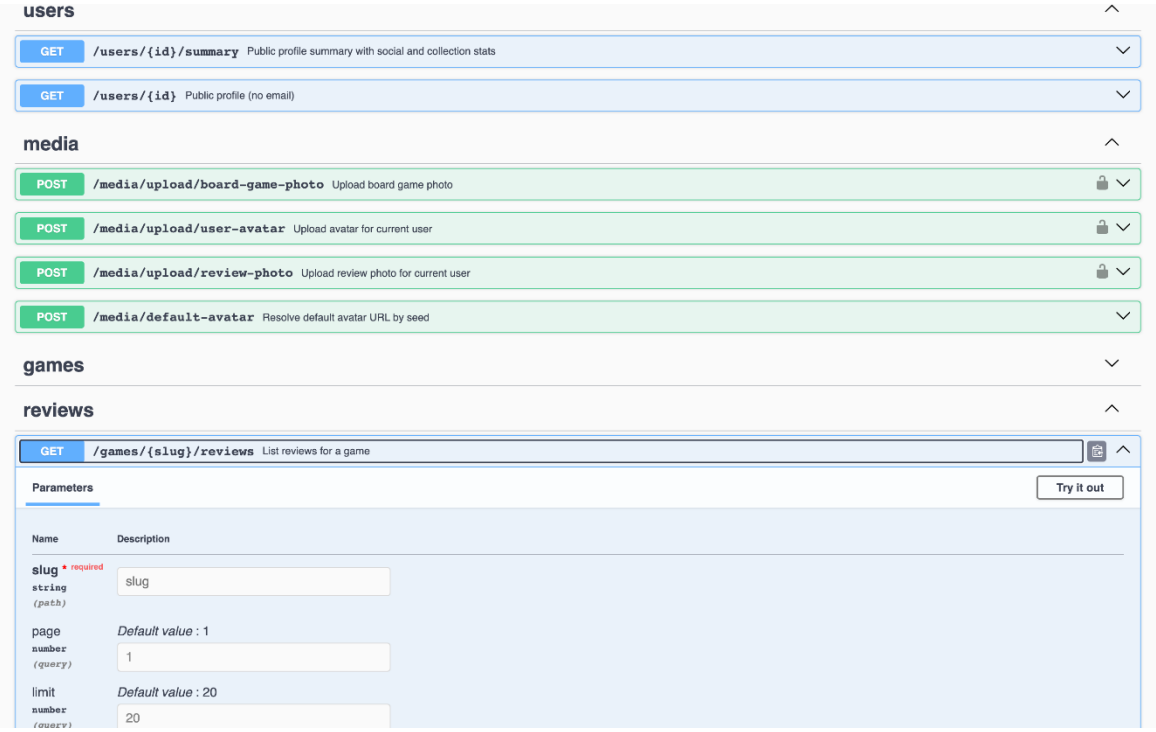


Рисунок 4.4 — Swagger UI: приклад запиту до API каталогу ігор

4.1.5. Перевірка модуля реального часу

Доставка повідомлень через Socket.IO перевіряється сценарієм із двома браузерними сесіями (два профілі-друзі з seed). Після надсилання повідомлення від першого користувача воно з’являється у стрічці другого без перезавантаження сторінки, що підтверджує роботу події message:new у namespace /chat і підписку компонента ChatThread на кімнату conv:<conversationId>. Додатково перевіряється повторне приєднання до

кімнати після розриву WebSocket (chat-socket.ts викликає join після reconnect). Логічна послідовність обміну наведена на рисунку 3.1 розділу 3.

4.1.6. Перевірка безпеки

Контрольний перелік безпекових перевірок узагальнено в таблиці 4.2. Перевірки виконуються вручну через Swagger та інструменти розробника в браузері.

Таблиця 4.2 — Контрольний перелік безпекових перевірок

Перевірка	Механізм у системі	Очікувана поведінка
Хешування паролів	BcryptPasswordHasher, c = 12	У БД лише passwordHash, відновити пароль неможливо
JWT access	JwtAccessTokenIssuer, exp у payload	Відхилення простроченого або підробленого токена
Refresh у HttpOnly-кукі	AuthCookieService	JavaScript не читає refresh-токен
Ротація та revoke family	RefreshTokenService, AuthSession	Повторне використання старого refresh відкликає сім'ю
Глобальний JWT guard	JwtAuthGuard, @Public()	401 без токена на захищених маршрутах
Rate limiting	ThrottlerGuard, 120 req / 60 s	429 при надмірній частоті запитів
Валідація DTO	ValidationPipe, whitelist	Відсікання зайвих полів у тілі запиту
CORS	WEB_ORIGIN у production	Запити лише з дозволеного походження
HTTPS	Nginx + Let's Encrypt	Кукі refresh лише через захищене з'єднання
Приватні ресурси	404 замість 403	Не розкривається існування приватної зустрічі/чату

4.1.7. Зведена матриця покриття

Таблиця 4.3 узагальнює зв'язок між функціональними модулями, типами тестування та сценаріями з розділу 1.

Таблиця 4.3 — Матриця модулів і типів тестування

Модуль	Unit (Jest)	Ручне UI	Swagger / API	Сценарій розділу 1
auth	bcrypt, refresh	TC-01, TC-02	login, register	Реєстрація, профіль
games	—	TC-03	GET /games	Каталог ігор
collections	—	TC-04, TC-10	/me/collection	Особиста колекція

Модуль	Unit (Jest)	Ручне UI	Swagger / API	Сценарій розділу 1
meetups	join/full, invite	TC-05, TC-06	POST /meetups	Координація зустрічей
friends	—	TC-07	/friends	Соціальні зв’язки
chat	—	TC-08, TC-09	REST + WebSocket	Комунікація
media	—	TC-12	upload	Аватари, фото ігор

4.2. Розгортання системи на VPS

Публікація програмного комплексу в продуктивному середовищі виконується на віртуальному сервері (VPS) під керуванням Ubuntu, з використанням Docker Compose для контейнерів застосунку та зовнішнього Nginx [32] на хості як зворотного проксі з сертифікатом HTTPS (Let’s Encrypt [33]). Підхід узгоджений із підрозділом 2.5.2 і дозволяє відтворити розгортання за документованою інструкцією в каталозі deploy/ репозиторію.

4.2.1. Архітектура продуктивного розгортання

Загальну схему взаємодії компонентів у продуктивному середовищі наведено на рисунку 4.5. Користувач звертається до системи через браузер по HTTPS; запити до статичного фронтенду проксуються на контейнер web (Nginx, 127.0.0.1:8080); запити з префіксом /api/ — на контейнер api (NestJS, 127.0.0.1:3000) із відсіканням префікса; з’єднання Socket.IO обслуговується окремим location /socket.io/ з заголовками Upgrade і Connection. Сервер API звертається до PostgreSQL у контейнері db і до зовнішнього сховища AWS S3.

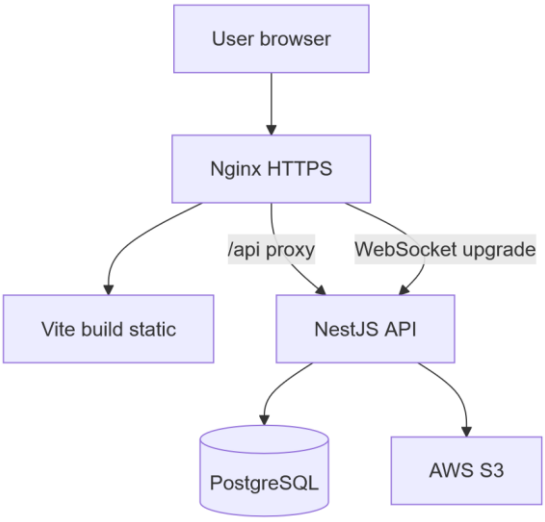


Рисунок 4.5 — Архітектура продуктивного розгортання (VPS, Docker, Nginx, HTTPS)

На відміну від середовища розробки (рисунок 3.1), у продуктивному середовищі фронтенд постачається як статичні файли, а єдина точка входу для користувача — доменне ім’я з валідним TLS-сертифікатом.

4.2.2. Вимоги до інфраструктури

Мінімальна конфігурація VPS для демонстраційного навантаження узагальнена в таблиці 4.4. Додатково на сервері має бути налаштований користувач deploy із SSH-ключем для автоматизованої доставки (підрозділ 4.2.6).

Таблиця 4.4 — Вимоги до серверної інфраструктури

Параметр	Рекомендоване значення	Призначення
ОС	Ubuntu 22.04 LTS або новіша	Сумісність з Docker і Certbot
vCPU	2	Збирання образів і одночасна робота API + БД
RAM	4 GB	PostgreSQL, Node.js, nginx у контейнерах
Диск	20 GB SSD	Образи Docker, том pgdata, логи
Мережа	Статична публічна IP	Стабільна адреса для DNS
Домен	A-запис на IP VPS	HTTPS і CORS (WEB_ORIGIN)
Програмне забезпечення	Docker Engine, Compose plugin, Nginx, Certbot	Контейнеризація та TLS

4.2.3. Контейнеризація для продуктивного середовища

У каталозі deploy/ підготовлено файли docker-compose.prod.yml, Dockerfile.api, Dockerfile.web і шаблон .env.prod.example. Docker Compose описує три сервіси в мережі app-network: db (postgres:16-alpine з томом pgdata і healthcheck), api (збирання з Dockerfile.api, env_file .env.prod, публікація 127.0.0.1:3000:3000), web (збирання з Dockerfile.web, аргумент VITE_API_URL на етапі build, публікація 127.0.0.1:8080:80). Залежність api від db реалізована через condition: service_healthy.

Dockerfile.api використовує багатостадійне збирання: стадія deps встановлює залежності pnpm; стадія build виконує prisma generate і nest build; стадія runner копіює dist, prisma та node_modules. Команда запуску контейнера показана на скріншоті з IDE (рисунок 4.6): вона спочатку застосовує міграції Prisma, потім стартує Node.js.

deploy › Dockerfile.api

```
1  FROM node:20-alpine AS base
2  RUN corepack enable
3  WORKDIR /workspace
4
5  FROM base AS deps
6  RUN apk add --no-cache python3 make g++
7  COPY package.json pnpm-lock.yaml pnpm-workspace.yaml ./
8  COPY apps/api/package.json apps/api/
9  COPY packages/shared/package.json packages/shared/
10 RUN pnpm install --frozen-lockfile --ignore-scripts --filter api --filter @boardgame/shared \
11   && pnpm --filter api rebuild bcrypt
12
13 FROM deps AS build
14 COPY packages/shared packages/shared
15 COPY apps/api apps/api
16 RUN pnpm --filter @boardgame/shared build \
17   && pnpm --filter api exec prisma generate \
18   && pnpm --filter api build
19
20 FROM node:20-alpine AS runner
21 RUN corepack enable
22 WORKDIR /workspace
23 ENV NODE_ENV=production
24 COPY --from=build /workspace/node_modules ./node_modules
25 COPY --from=build /workspace/apps/api/node_modules ./apps/api/node_modules
26 COPY --from=build /workspace/apps/api/dist ./apps/api/dist
27 COPY --from=build /workspace/apps/api/package.json ./apps/api/package.json
28 COPY --from=build /workspace/apps/api/prisma ./apps/api/prisma
29 COPY --from=build /workspace/packages/shared ./packages/shared
30 COPY package.json pnpm-workspace.yaml ./
31 EXPOSE 3000
32 CMD ["sh", "-c", "pnpm --filter api exec prisma migrate deploy && node apps/api/dist/main.js"]
```

Рисунок 4.6 — Скріншот команди запуску контейнера API у продуктивному середовищі

Dockerfile.web на стадії build виконує vite build із підстановкою VITE_API_URL; на стадії runner статика копіюється в nginx:alpine з конфігурацією deploy/nginx-web.conf (маршрутизація SPA, fallback на index.html). Запуск стеку з каталогу deploy/:

docker compose -f docker-compose.prod.yml --env-file .env.prod up -d --build

Після старту API слухає лише localhost:3000, веб — localhost:8080; зовнішній доступ забезпечує Nginx на хості.

4.2.4. Конфігурація через змінні середовища

Секрети та параметри середовища не зберігаються в репозиторії. Оператор копіює deploy/.env.prod.example у deploy/.env.prod і заповнює значення на VPS. Файл .env.prod залишається лише на сервері і не потрапляє в git; при автоматичному деплої git reset --hard не перезаписує нетрекані файли, тому секрети зберігаються між оновленнями. Перелік змінних — у таблиці 4.5.

Таблиця 4.5 — Змінні продуктивного середовища

Змінна	Компонент	Призначення
POSTGRES_USER, POSTGRES_PASSWORD, POSTGRES_DB	db, api	Облікові дані PostgreSQL
DATABASE_URL	api	Рядок підключення Prisma
JWT_ACCESS_SECRET, JWT_ACCESS_TTL	api	Підпис і термін access JWT
JWT_REFRESH_SECRET, JWT_REFRESH_TTL	api	Підпис і термін refresh
WEB_ORIGIN	api	CORS і дозволене походження кукі
VITE_API_URL	web (build-time)	Базова URL API для клієнта
GOOGLE_CLIENT_ID, GOOGLE_CLIENT_SECRET, GOOGLE_CALLBACK_URL	api	OAuth Google (за потреби)
AWS_REGION, AWS_S3_BUCKET, AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY	api	Завантаження зображень у S3

4.2.5. Reverse proxy на Nginx і HTTPS

Конфігурація зворотного проксі наведена у файлі deploy/nginx-site.conf. Сервер на порту 80 виконує перенаправлення 301 на HTTPS; сервер на порту 443 обслуговує TLS із сертифікатами Let’s Encrypt (certbot --nginx -d YOUR_DOMAIN). Фрагмент маршрутизації API та WebSocket наведено у вигляді скріншота з IDE на рисунку 4.7.

deploy > nginx-site.conf

```
upstream boardgame_api {
  server 127.0.0.1:3000;
}

upstream boardgame_web {
  server 127.0.0.1:8080;
}

server {
  listen 80;
  server_name YOUR_DOMAIN;
  return 301 https://$host$request_uri;
}

server {
  listen 443 ssl http2;
  server_name YOUR_DOMAIN;

  ssl_certificate     /etc/letsencrypt/live/YOUR_DOMAIN/fullchain.pem;
  ssl_certificate_key /etc/letsencrypt/live/YOUR_DOMAIN/privkey.pem;

  client_max_body_size 20M;

  location / {
    proxy_pass http://boardgame_web;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
  }

  location /api/ {
    rewrite ^/api/(.*)$ /$1 break;
    proxy_pass http://boardgame_api;
    proxy_http_version 1.1;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
  }

  location /socket.io/ {
    proxy_pass http://boardgame_api;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
  }
}
```

Рисунок 4.7 — Скріншот фрагмента nginx-site.conf

Директива client_max_body_size 20M дозволяє завантаження зображень. Після редагування YOUR_DOMAIN і встановлення сертифіката виконується nginx -t && systemctl reload nginx. Відкриття головної сторінки за публічним HTTPS-URL після розгортання наведено на рисунку 4.8 — це підтверджує роботу всього ланцюга DNS → Nginx → web → API.

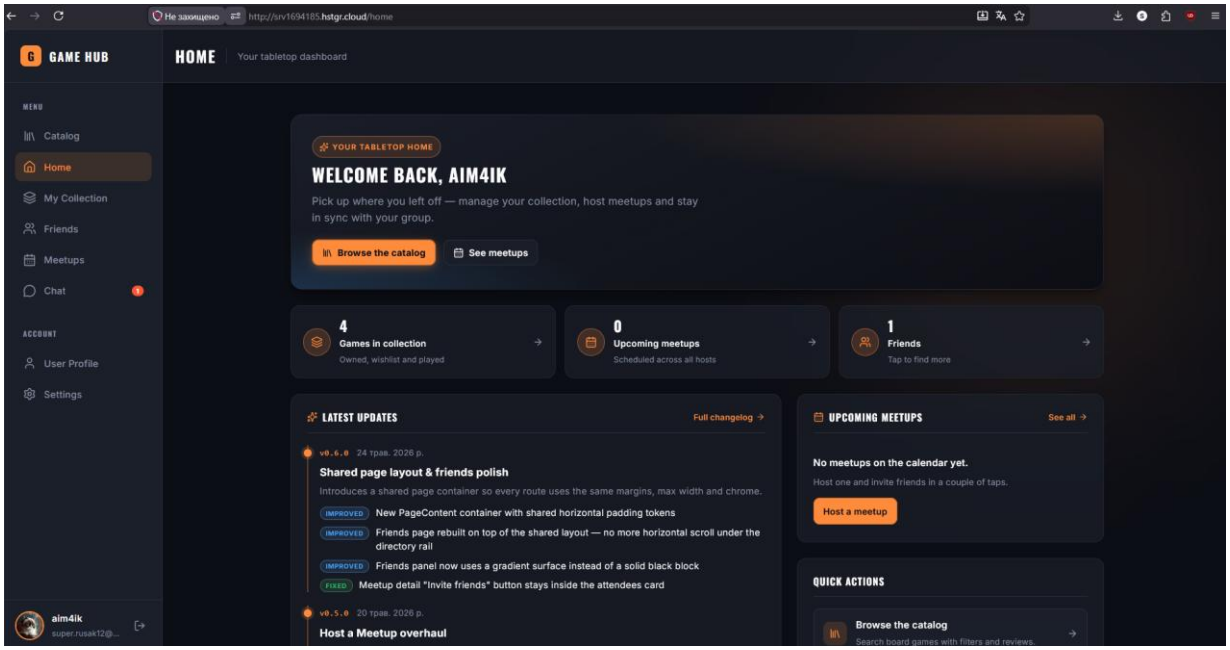


Рисунок 4.8 — Головна сторінка платформи після розгортання на VPS

4.2.6. Безперервна доставка через GitHub Actions

Для автоматизації оновлення продуктивного середовища у системі GitHub Actions [34] налаштовано workflow .github/workflows/deploy.yml. Подія push у гілку main або ручний запуск workflow_dispatch запускає job, що виконує checkout репозиторію, налаштування SSH (секрети VPS_HOST, VPS_USER, VPS_SSH_KEY) і виклик скрипта deploy/deploy.sh через stdin по SSH. Скрипт, фрагмент якого показано на скріншоті з IDE на рисунку 4.9, оновлює робочий каталог і перебудовує контейнери; concurrency group deploy-vps запобігає паралельним деплоям. Послідовність взаємодії учасників наведено на рисунку 4.10.

deploy > $ deploy.sh

```
#!/usr/bin/env bash
set -euo pipefail

REPO_DIR="${REPO_DIR:-/home/deploy/board-games-network}"
BRANCH="${BRANCH:-main}"
COMPOSE_FILE="docker-compose.prod.yml"
ENV_FILE=".env.prod"

echo "==> deploy started at $(date --iso-8601=seconds) on $(hostname)"
cd "$REPO_DIR"

echo "==> fetching origin/$BRANCH"
git fetch --depth 1 origin "$BRANCH"
git reset --hard "origin/$BRANCH"
echo "    HEAD is now $(git rev-parse --short HEAD) — $(git log -1 --pretty=%s)"

cd deploy

if [ ! -f "$ENV_FILE" ]; then
  echo "ERROR: $ENV_FILE not found in $(pwd); aborting" >&2
  exit 1
fi

echo "==> building images and recreating containers"
docker compose -f "$COMPOSE_FILE" --env-file "$ENV_FILE" up -d --build --remove-orphans

echo "==> pruning dangling images"
docker image prune -f >/dev/null

echo "==> deploy finished, container status:"
docker compose -f "$COMPOSE_FILE" --env-file "$ENV_FILE" ps
```

Рисунок 4.9 — Скріншот фрагмента deploy/deploy.sh (git fetch + docker compose)

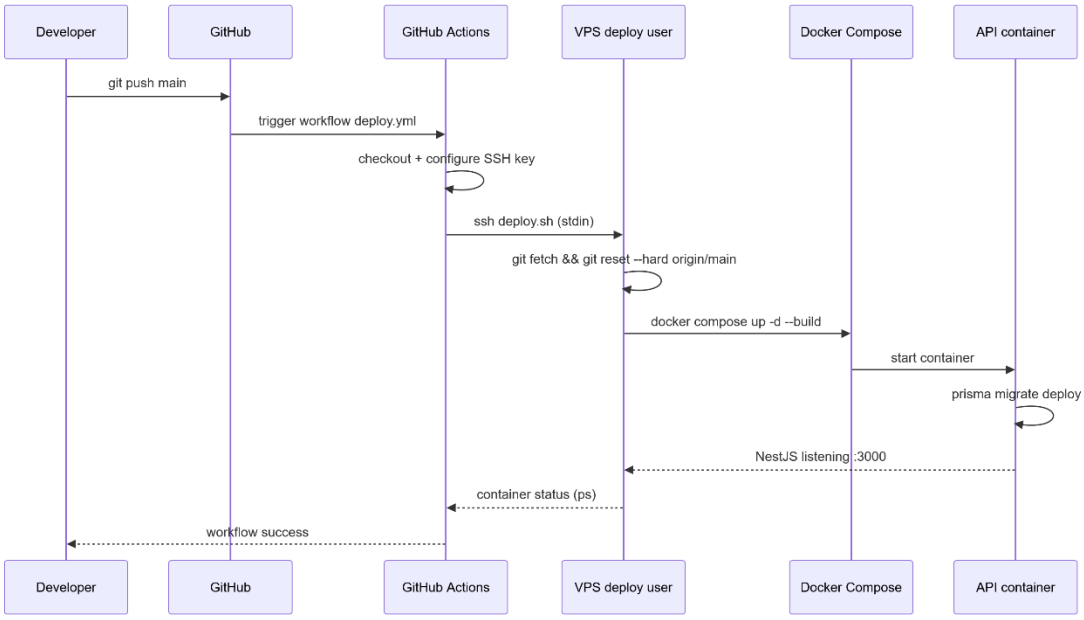


Рисунок 4.10 — Діаграма послідовності автоматичного розгортання (CI/CD)

4.2.7. Міграції бази даних та експлуатація

Структура PostgreSQL версіонується міграціями Prisma (apps/api/prisma/migrations); при кожному старті контейнера api виконується prisma migrate deploy, що застосовує невиконані міграції до бази в томі pgdata без ручного втручання оператора. Перед оновленням із breaking-змінами схеми рекомендується резервне копіювання тома pgdata (pg_dump або snapshot диска VPS).

Щоденна експлуатація включає перегляд стану контейнерів (docker compose ps) і журналів (docker compose logs -f api), моніторинг вільного місця на диску. Політика restart: unless-stopped забезпечує автоматичний перезапуск після збою або перезавантаження VPS. Оновлення версії застосунку спричиняє короткочасну недоступність API під час перестворення контейнерів, що прийнятно для дипломного прототипу; для високої доступності в майбутньому потрібні реплікація API, балансування навантаження та окремий кластер БД.

4.3. Інструкція з використання програмного продукту

Нижче наведено покроковий опис основних сценаріїв роботи з платформою після входу в систему. Інтерфейс побудований на єдиному каркасі RootLayout із бічною навігацією; захищені розділи доступні лише автентифікованим користувачам.

4.3.1. Реєстрація та вхід

Користувач відкриває вебзастосунок за адресою, наданою після розгортання (локально — http://localhost:5173, на VPS — https://YOUR_DOMAIN). На публічних сторінках у бічній панелі доступна кнопка «Увійти», що відкриває модальне вікно автентифікації; для створення облікового запису перемикаються на вкладку реєстрації. Після успішного входу клієнт отримує access-токен, refresh зберігається в HttpOnly-кукі, стають доступними розділи «Колекція», «Зустрічі», «Друзі», «Повідомлення», «Профіль». При неправильному паролі (ТС-02) відображається уніфіковане

повідомлення про помилку без розкриття факту існування email у базі. Скріншот — рисунок 4.11.

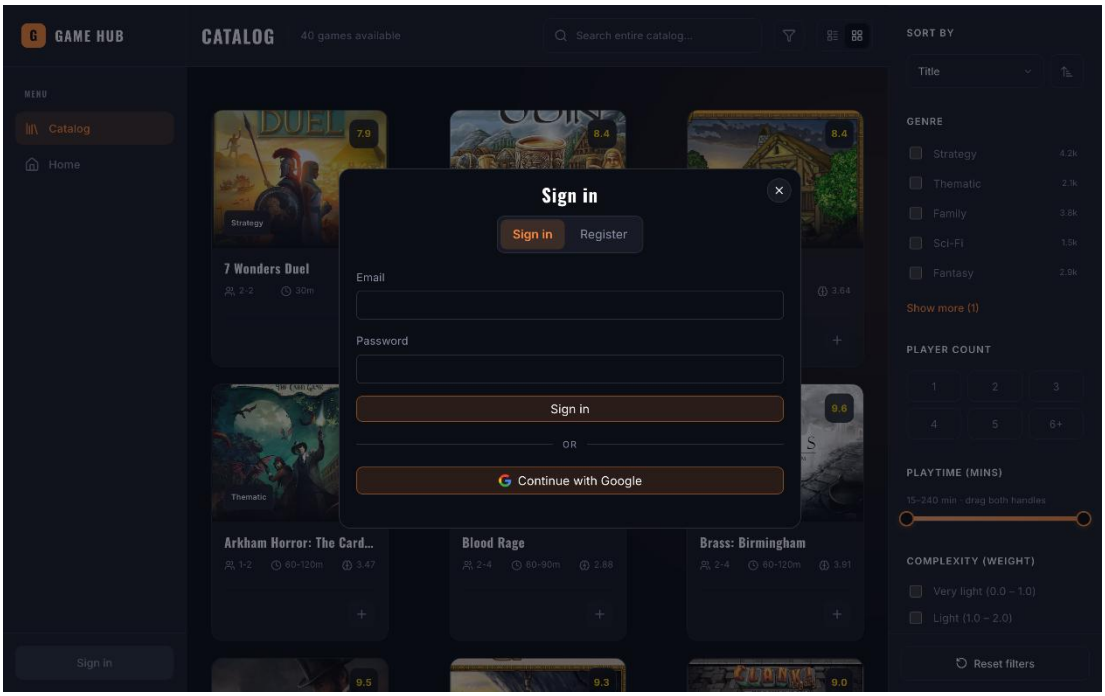


Рисунок 4.11 — Реєстрація та вхід у систему (тест-кейс ТС-01)

4.3.2. Каталог настільних ігор

Розділ «Каталог» (/games) відображає список ігор із пагінацією. Компонент CatalogSearchShell забезпечує пошук за назвою; CatalogFiltersPanel — фільтрацію за жанром, діапазоном кількості гравців, тривалістю партії та складністю. Параметри фільтрів синхронізуються з URL (validateSearch у TanStack Router): посилання на конкретну вибірку можна передати іншому користувачу, а кнопки «назад/вперед» браузера відновлюють стан. Клік по картці гри (CatalogGameCard) відкриває сторінку деталей. Скріншот — рисунок 4.12 (ТС-03).

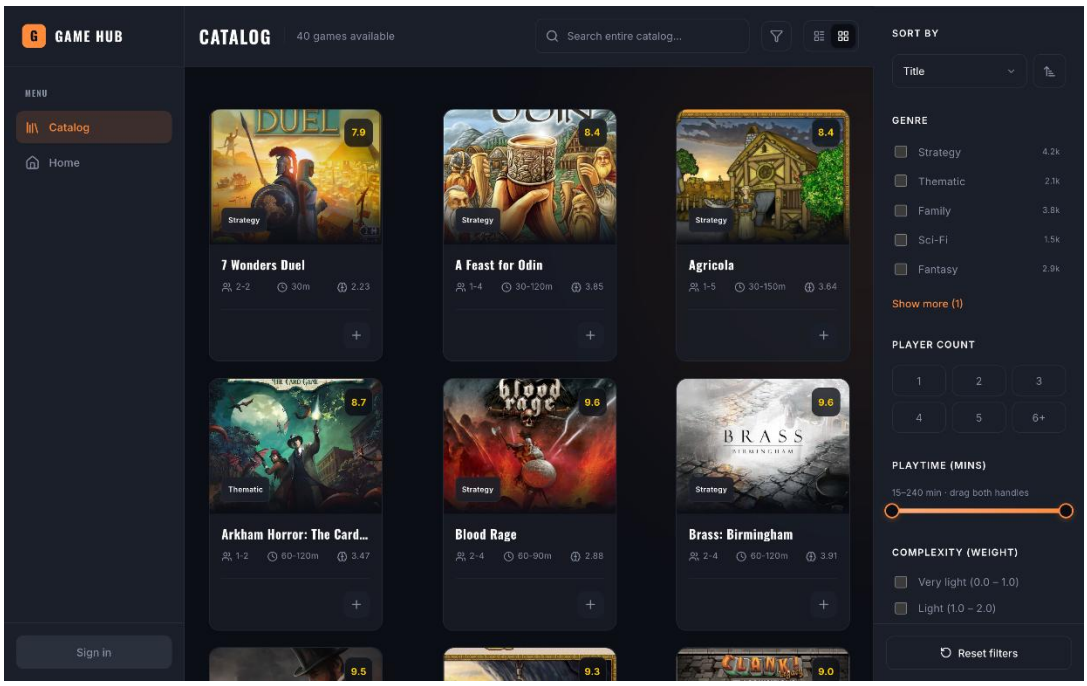


Рисунок 4.12 — Каталог ігор із пошуком і фільтрами (тест-кейс ТС-03)

4.3.3. Сторінка деталей гри

Маршрут /games/$slug відображає опис гри, агрегований рейтинг, відгуки інших користувачів і список майбутніх зустрічей, пов’язаних із цією грою; авторизований користувач може додати гру до колекції або залишити оцінку безпосередньо з цієї сторінки (рисунок 4.13).

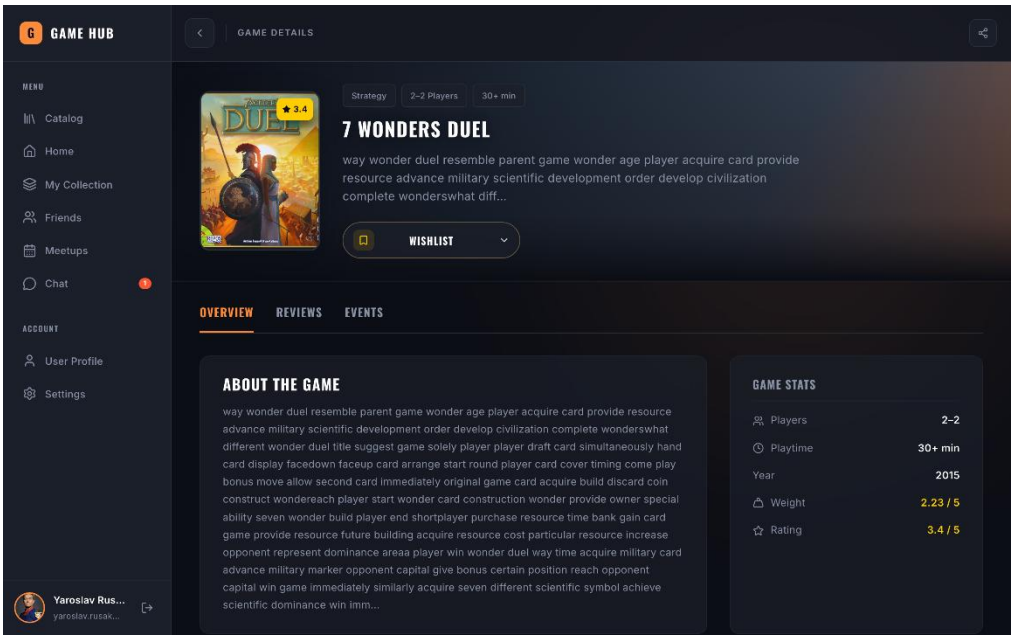


Рисунок 4.13 — Сторінка деталей настільної гри

4.3.4. Особиста колекція

Розділ «Колекція» (/collection) показує ігри користувача зі статусами OWNED, WISHLIST, PREVIOUSLY_OWNED. Доступна зміна статусу, додавання з каталогу та видалення запису (ТС-04, ТС-10). Скріншот — рисунок 4.14.

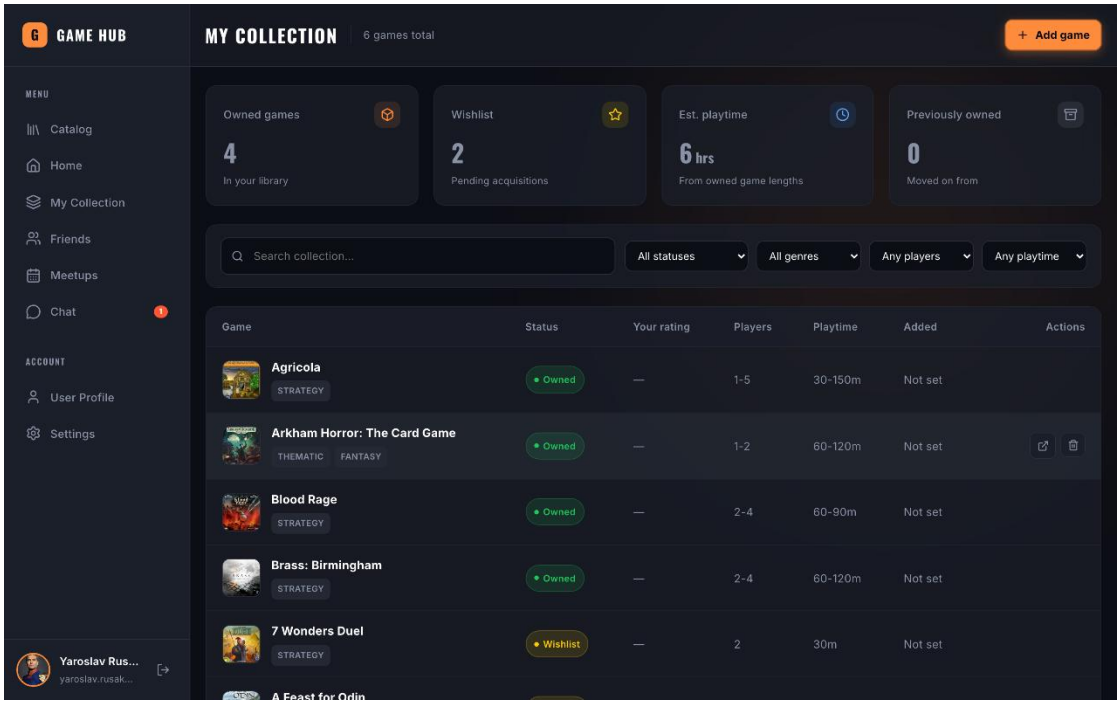


Рисунок 4.14 — Особиста колекція ігор (тест-кейси ТС-04, ТС-10)

4.3.5. Створення ігрової зустрічі

У розділі «Зустрічі» (/meetups) передбачено перемикання між списком і календарним виглядом. Кнопка створення веде на /meetups/new; форма MeetupForm містить поля назви, дати й часу, місця, опису, пов’язаної гри, максимальної кількості учасників (maxPlayers) та видимості (PUBLIC, FRIENDS, INVITE_ONLY). Після збереження публічна зустріч з’являється в загальному списку (ТС-06). Скріншот — рисунок 4.15.

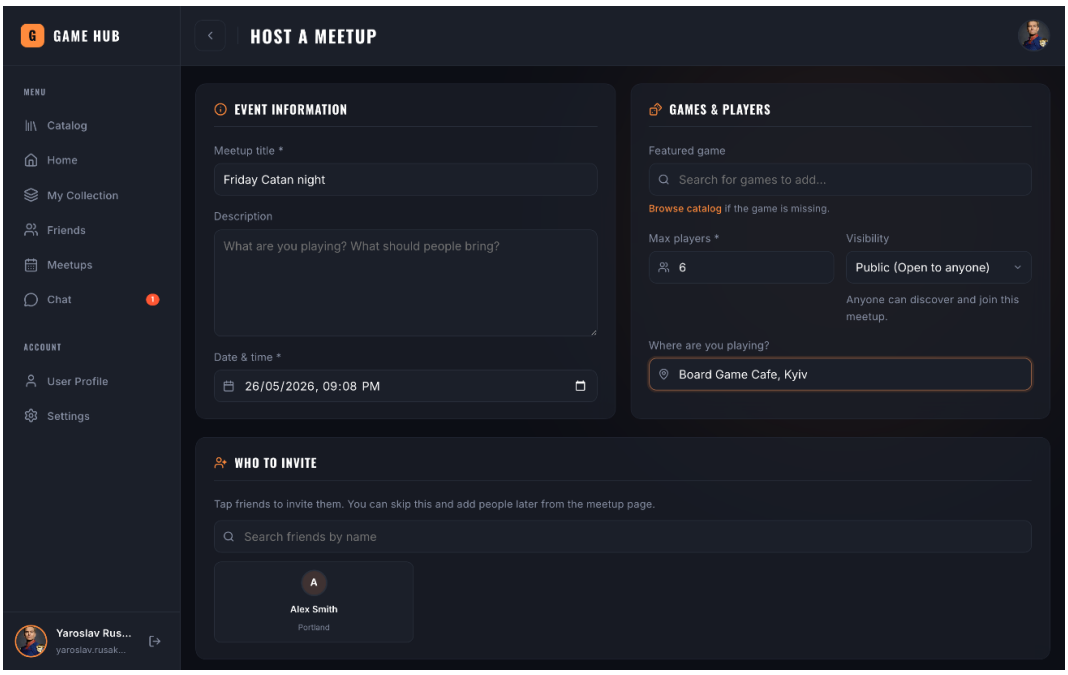


Рисунок 4.15 — Форма створення ігрової зустрічі (тест-кейс ТС-06)

4.3.6. Сторінка зустрічі

Маршрут /meetups/$meetupId показує деталі події, склад учасників і доступні дії: приєднатися (якщо виконуються правила видимості та залишилися вільні місця), покинути зустріч, перейти до запрошень або скасувати подію (для організатора). При переповненні клієнт відображає заблоковану кнопку приєднання (ТС-05, рисунок 4.3). Загальний вигляд сторінки — рисунок 4.16.

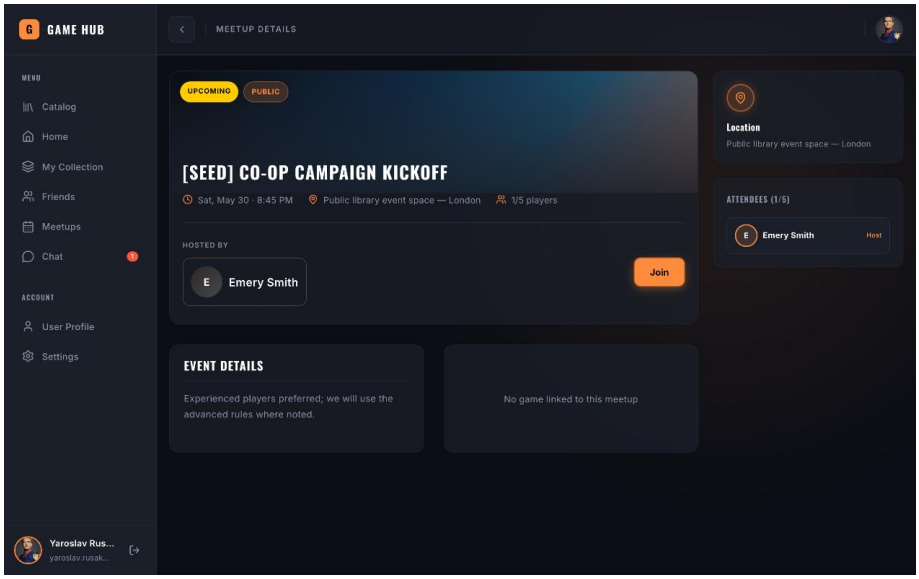


Рисунок 4.16 — Сторінка ігрової зустрічі зі списком учасників

4.3.7. Друзі

Розділ «Друзі» (/friends) містить пошук користувачів (discover), вкладки вхідних і вихідних запрошень та список прийнятих друзів. Статус стосунків (none, outgoing_pending, incoming_pending, friend) визначається сервером і відображається відповідною кнопкою без додаткових запитів (ТС-07). Скріншот — рисунок 4.17.

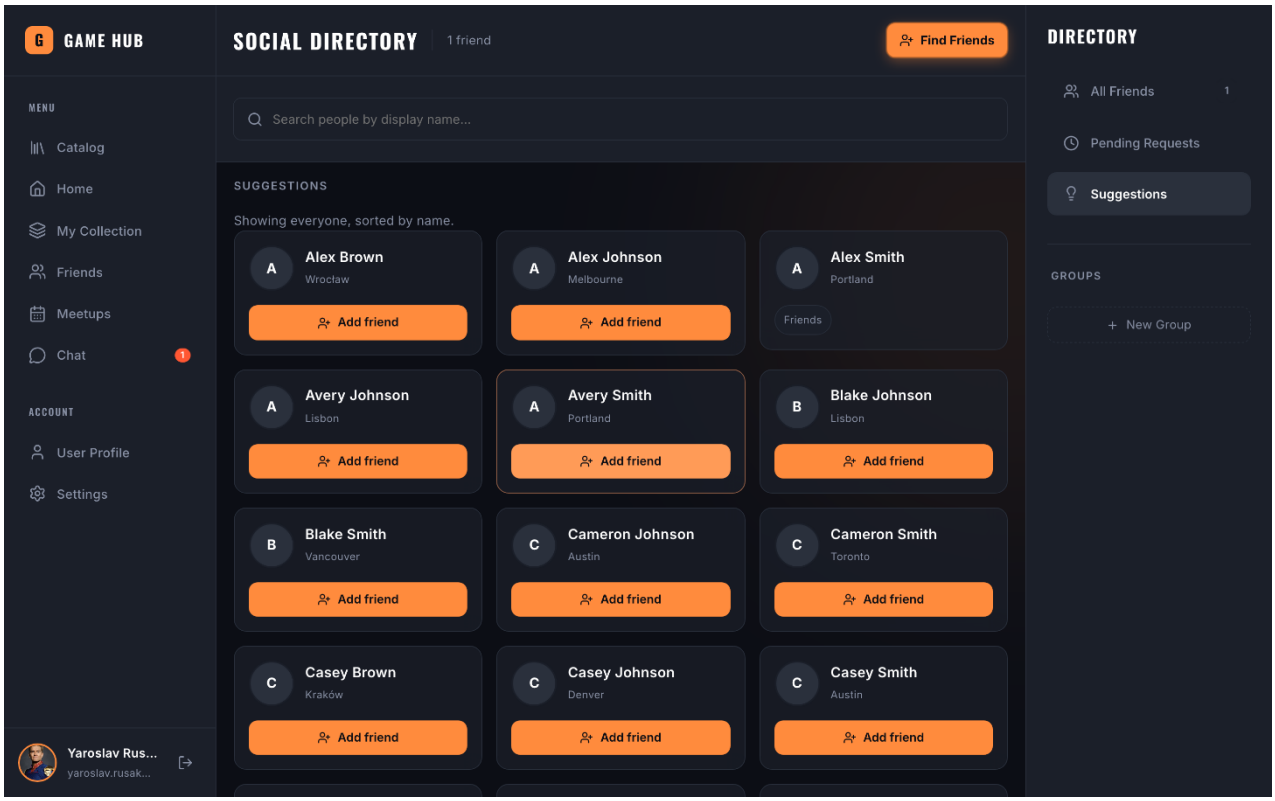


Рисунок 4.17 — Пошук користувачів і дружні запити (тест-кейс ТС-07)

4.3.8. Повідомлення

Розділ «Повідомлення» (/messages) реалізований як тришпальтовий макет: список розмов (ChatConversationsRail), стрічка повідомлень (ChatThread) і контекстна панель (ChatContextPanel) з учасниками або деталями зустрічі для розмов типу SESSION. Підтримуються direct-, group- і session-чати; нові повідомлення з’являються у співрозмовника в реальному часі без перезавантаження (ТС-08). Скріншот — рисунок 4.18.

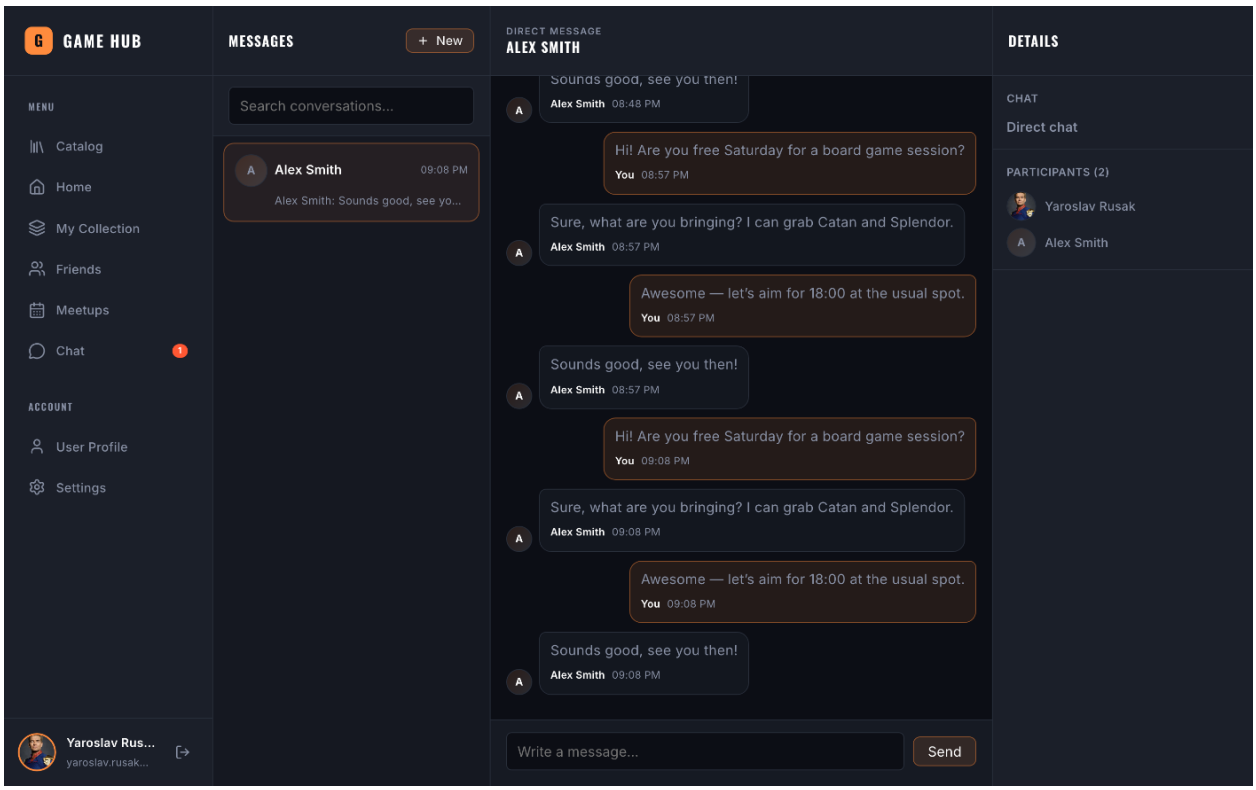


Рисунок 4.18 — Обмін повідомленнями в direct-чаті (тест-кейс ТС-08)

4.3.9. Профіль і налаштування

Сторінка «Профіль» (/profile) показує статистику, прев'ю колекції та дозволяє редагувати displayName і біо (ТС-11). Сторінка «Налаштування» (/settings) дозволяє перемкнути тему інтерфейсу (світла/темна); вибір зберігається локально в браузері. Інтерфейс адаптовано до вузьких екранів: бічна навігація та шпальти чату перемикаються відповідно до CSS-медіазапитів. Загальний вигляд профілю та налаштувань — рисунок 4.19.

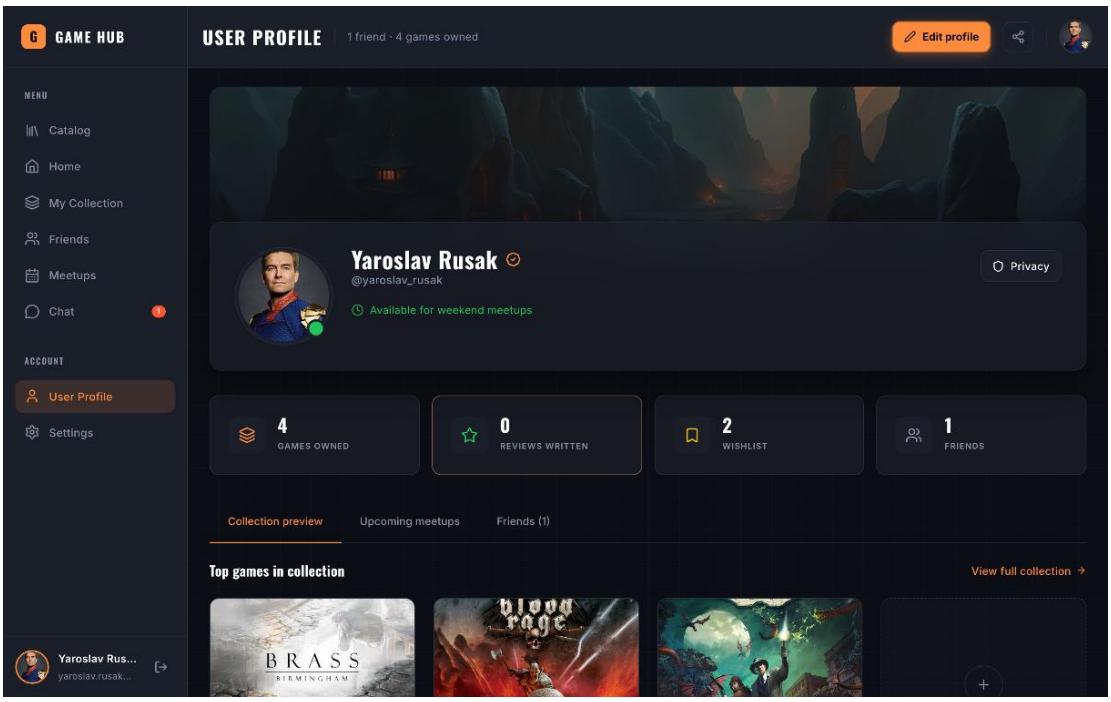


Рисунок 4.19 — Профіль користувача та сторінка налаштувань (тест-кейс ТС-11)

4.4. Аналіз отриманих результатів і обмеження

Розроблена платформа об’єднує в одному вебзастосунку функції, які в розділі 1 були розподілені між спеціалізованими каталогами (аналог BoardGameGeek), універсальними сервісами подій і месенджерами. Зіставлення вимог із реалізованими модулями та артефактами тестування наведено в таблиці 4.6.

Таблиця 4.6 — Зіставлення вимог розділу 1 з реалізацією та перевіркою

Вимога (розділ 1)	Модуль	Сторінка / API	Підтвердження
Каталог ігор	games	/games	ТС-03, Swagger
Колекція користувача	collections	/collection	ТС-04, ТС-10
Координація зустрічей	meetups	/meetups	ТС-05, ТС-06, unit-тест joinMeetup
Соціальні зв’язки	friends	/friends	ТС-07
Комунікація	chat	/messages	ТС-08, WebSocket
Безпека облікових записів	auth	/auth/*	Unit-тести bcrypt, JWT
Зберігання медіа	media	upload API	ТС-12

Прототип має такі обмеження: API та WebSocket-шлюз розгорнуті на одному VPS без балансування навантаження; відсутні push-сповіщення поза відкритою вкладкою браузера; завантаження зображень залежить від налаштування AWS S3; повне e2e-покриття всіх маршрутів UI не реалізоване; резервне копіювання та моніторинг залишаються на відповідальності оператора VPS.

До пріоритетних напрямів подальшого розвитку належать: мобільний клієнт (PWA або нативний); рекомендаційна система ігор на основі колекцій і оцінок; email- і push-сповіщення; горизонтальне масштабування модуля чату (окремий процес Socket.IO, Redis adapter); розширення набору автоматизованих тестів (e2e Playwright, інтеграційні тести з тестовою БД).

ВИСНОВКИ ДО РОЗДІЛУ

У четвертому розділі виконано перевірку якості розробленої соціальної вебплатформи для настільних ігор, описано порядок її розгортання на віртуальному сервері та наведено інструкцію з використання основних функцій. Перевірка побудована як комбінація автоматизованого модульного тестування серверної бізнес-логіки, ручного функціонального тестування інтерфейсу в браузері, контролю REST-контракту через Swagger UI та безпекового переліку, що дозволяє охопити різні рівні системи без невиправданого ускладнення процесу для дипломного прототипу.

Модульні тести Jest підтвердили коректність ключових інваріантів предметної області — хешування паролів класом BcryptPasswordHasher, обмеження учасників зустрічі за maxPlayers, правил видимості INVITE_ONLY та ротації refresh-сесії з відкликанням сім'ї токенів. Ручні тест-кейси TC-01–TC-12 продемонстровано в підрозділі 4.3 разом зі скріншотами інтерфейсу: вони охоплюють реєстрацію і вхід, каталог із пошуком і фільтрами, колекцію, створення й приєднання до зустрічі (включно зі сценарієм переповнення TC-05), друзів, чат у реальному часі та редагування профілю. Перевірка REST-контракту через Swagger UI і безпековий перелік (JWT-guard, refresh у HttpOnly-куках, rate limit, валідація DTO, CORS, HTTPS, 404 для приватних ресурсів) доповнюють картину якості продукту.

Розгортання на VPS оформлено як відтворювана інструкція в каталозі deploy/: Docker Compose описує три сервіси (PostgreSQL з томом pgdata, API з багатостадійним образом і статичний фронтенд за Nginx); зовнішній Nginx на хості перенаправляє HTTP на HTTPS, маршрутизує /api/ на контейнер API і обслуговує Socket.IO через окремий location, сертифікат випускається Let's Encrypt, секрети — у файлі .env.prod на сервері, а структура бази версіонується автоматичними міграціями Prisma при старті API. Безперервну доставку реалізовано через GitHub Actions: workflow за подією push у main виконує скрипт deploy.sh по SSH (git reset, docker compose up -d --build), а concurrency group deploy-vps запобігає паралельним деплоям.

Таким чином, продемонстровано, що розроблений програмний продукт відповідає сформульованим у розділі 1 функціональним вимогам, придатний до публікації на типовому VPS і доступний кінцевому користувачеві через браузер. Окреслені в підрозділі 4.4 обмеження прототипу — єдиний VPS без балансування, відсутність push-сповіщень, ручне резервне копіювання, неповне e2e-покриття — формують пріоритетні напрями подальшого розвитку платформи. Тим самим завершено цикл «аналіз — проєктування — реалізація — перевірка та публікація», передбачений структурою дипломної роботи.

ВИСНОВКИ

У дипломній роботі розв’язано задачу створення веборієнтованої системи для координації та взаємодії спільнот любителів настільних ігор, яка поєднує в одному цифровому середовищі каталог ігор, особисті колекції, оцінки й відгуки, систему друзів, організацію ігрових зустрічей та обмін повідомленнями. Мету роботи — створити систему, що спрощує організацію ігрових зустрічей і підтримує взаємодію між учасниками спільнот настільних ігор, — досягнуто: отримано працездатний програмний прототип, документований до рівня, придатного для відтворюваного розгортання на типовому віртуальному сервері та використання кінцевими користувачами через браузер.

За результатами виконаних досліджень і розробки отримано такі основні результати.

1. Виконано аналіз предметної області, визначено цільову авдиторію (новачки, досвідчені гравці, організатори локальних зустрічей) та сформульовано основні сценарії взаємодії користувачів — перегляд каталогу настільних ігор, формування власної колекції, пошук інших гравців і встановлення соціальних зв’язків, організацію ігрових зустрічей, комунікацію учасників у контексті події та обмін враженнями після партії. На основі огляду існуючих рішень — глобальних інформаційних баз (BoardGameGeek), універсальних сервісів подій (Meetup, Facebook Events) та локальних месенджерів (Telegram, Discord, Viber) — установлено, що жоден тип засобу не закриває всіх виявлених сценаріїв одночасно, і обґрунтовано доцільність розробки спеціалізованої вебплатформи.

2. Спроєктовано архітектуру застосунку: обрано клієнт-серверну модель із модульним поділом серверної частини, реляційне зберігання взаємопов’язаних сутностей у PostgreSQL та поєднання REST-запитів із каналом реального часу для модуля повідомлень. Обґрунтовано вибір технологічного стека — Node.js і TypeScript, NestJS і Prisma ORM на сервері;

React, Vite, TanStack Router і TanStack React Query на клієнті; Socket.IO для чату; Docker Compose для відтворення середовища; AWS S3 для зберігання зображень. Окремо аргументовано вибір Mermaid для діаграм, pnpm для монорепозиторію, а також інструментів контролю якості — ESLint, Prettier і Jest.

3. Реалізовано серверну та клієнтську частини системи у вигляді монорепозиторію pnpm із трьох складових: серверного застосунку на NestJS із сімома доменними модулями (auth, games, collections, friends, meetups, chat, media), клієнтського застосунку на React і Vite та пакета спільних контрактів @boardgame/shared, що утримує типи запитів і відповідей синхронними між клієнтом і сервером. Серверну частину організовано за шаровою схемою (presentation, application, domain, infrastructure); клієнтську — як односторінковий застосунок із типізованою маршрутизацією, централізованим API-клієнтом і збереженням стану фільтрів у параметрах URL. Структуру бази даних версіоновано міграціями Prisma; правила цілісності задано безпосередньо у schema.prisma.

4. Забезпечено роботу ключових функцій, передбачених завданням: автентифікації користувачів за схемою «короткий JWT плюс опаковий refresh-токен у HttpOnly-куках» із ротацією сесій; каталогу ігор із пошуком, фільтрами та URL-сумісним станом; особистих колекцій зі статусами OWNED, WISHLIST, PREVIOUSLY_OWNED, оцінками й відгуками; системи друзів із вхідними/вихідними запрошеннями; ігрових зустрічей із правилами видимості (PUBLIC, FRIENDS, INVITE_ONLY) та контролем кількості учасників; гібридного чату (REST для історії, Socket.IO у просторі імен /chat для оновлень підключеним учасникам кімнати) для direct-, group- і session-розмов.

5. Виконано перевірку якості розробленого продукту, що поєднує автоматизоване модульне тестування серверної бізнес-логіки засобами Jest, ручне функціональне тестування інтерфейсу за дванадцятьма тест-кейсами TC-01–TC-12, контроль REST-контракту через Swagger UI та безпековий

перелік (хешування паролів, ротація refresh-токенів, глобальний JWT-guard, обмеження частоти запитів, валідація DTO, CORS, HTTPS і уніфікація відповіді 404 для приватних ресурсів). Підготовлено відтворюване розгортання системи на VPS під керуванням Ubuntu: контейнеризацію Docker Compose, зворотний проксі Nginx із HTTPS-сертифікатом Let’s Encrypt, конфігурацію через файл .env.prod, автоматичні міграції Prisma при старті API та безперервну доставку через GitHub Actions.

Практична цінність розробленої системи полягає в тому, що вона може бути використана як основа для онлайн-платформи локальної або ширшої спільноти гравців: її функціональність спрямована на зменшення організаційних бар’єрів, покращення комунікації між учасниками та створення єдиного простору, у якому зберігається інформація про ігри, користувачів і заплановані події. Це робить продукт корисним як для окремих гравців, так і для невеликих клубів або груп, які регулярно проводять ігрові зустрічі.

Перспективними напрямами подальшого розвитку системи є: реалізація мобільного клієнта (PWA або нативного застосунку); розроблення рекомендаційної системи ігор на основі колекцій і користувацьких оцінок; додавання email- і push-сповіщень про події в зустрічах і чатах поза відкритою вкладкою браузера; горизонтальне масштабування модуля чату (окремий процес Socket.IO та Redis adapter); розширення набору автоматизованих тестів (e2e Playwright, інтеграційні тести з тестовою базою даних); налаштування регулярного резервного копіювання тома PostgreSQL і моніторингу продуктивних метрик.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. BoardGameGeek [Електронний ресурс]. URL: https://boardgamegeek.com/ (дата звернення: 27.05.2026).

2. Meetup [Електронний ресурс]. URL: https://www.meetup.com/ (дата звернення: 27.05.2026).

3. Facebook Events [Електронний ресурс]. URL: https://www.facebook.com/events/ (дата звернення: 27.05.2026).

4. Telegram [Електронний ресурс]. URL: https://telegram.org/ (дата звернення: 27.05.2026).

5. Discord [Електронний ресурс]. URL: https://discord.com/ (дата звернення: 27.05.2026).

6. MDN Web Docs. HTTP Overview [Електронний ресурс]. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP (дата звернення: 27.05.2026).

7. RFC 6455. The WebSocket Protocol [Електронний ресурс]. URL: https://www.rfc-editor.org/rfc/rfc6455 (дата звернення: 27.05.2026).

8. NestJS Documentation [Електронний ресурс]. URL: https://docs.nestjs.com/ (дата звернення: 27.05.2026).

9. PostgreSQL Documentation [Електронний ресурс]. URL: https://www.postgresql.org/docs/ (дата звернення: 27.05.2026).

10. Prisma ORM Documentation [Електронний ресурс]. URL: https://www.prisma.io/docs (дата звернення: 27.05.2026).

11. Node.js Documentation [Електронний ресурс]. URL: https://nodejs.org/en/docs (дата звернення: 27.05.2026).

12. TypeScript Documentation [Електронний ресурс]. URL: https://www.typescriptlang.org/docs/ (дата звернення: 27.05.2026).

13. Swagger Documentation [Електронний ресурс]. URL: https://swagger.io/docs/ (дата звернення: 27.05.2026).

14. RFC 7519. JSON Web Token (JWT) [Електронний ресурс]. URL: https://www.rfc-editor.org/rfc/rfc7519 (дата звернення: 27.05.2026).

15. bcrypt for Node.js [Електронний ресурс]. URL: https://github.com/kelektiv/node.bcrypt.js (дата звернення: 27.05.2026).

16. React Documentation [Електронний ресурс]. URL: https://react.dev/ (дата звернення: 27.05.2026).

17. Vite Documentation [Електронний ресурс]. URL: https://vitejs.dev/guide/ (дата звернення: 27.05.2026).

18. TanStack Router Documentation [Електронний ресурс]. URL: https://tanstack.com/router/latest (дата звернення: 27.05.2026).

19. TanStack Query Documentation [Електронний ресурс]. URL: https://tanstack.com/query/latest (дата звернення: 27.05.2026).

20. RFC 9110. HTTP Semantics [Електронний ресурс]. URL: https://www.rfc-editor.org/rfc/rfc9110 (дата звернення: 27.05.2026).

21. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : Ph.D. dissertation. Irvine : University of California, 2000. 162 p. URL: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm (дата звернення: 27.05.2026).

22. Socket.IO Documentation [Електронний ресурс]. URL: https://socket.io/docs/v4/ (дата звернення: 27.05.2026).

23. Docker Documentation [Електронний ресурс]. URL: https://docs.docker.com/ (дата звернення: 27.05.2026).

24. Docker Compose Documentation [Електронний ресурс]. URL: https://docs.docker.com/compose/ (дата звернення: 27.05.2026).

25. Amazon Web Services S3 Documentation [Електронний ресурс]. URL: https://docs.aws.amazon.com/s3/ (дата звернення: 27.05.2026).

26. Mermaid Documentation [Електронний ресурс]. URL: https://mermaid.js.org/ (дата звернення: 27.05.2026).

27. ESLint Documentation [Електронний ресурс]. URL: https://eslint.org/docs/latest/ (дата звернення: 27.05.2026).

28. Prettier Documentation [Електронний ресурс]. URL: https://prettier.io/docs/en/ (дата звернення: 27.05.2026).

29. Jest Documentation [Електронний ресурс]. URL: https://jestjs.io/docs/getting-started (дата звернення: 27.05.2026).

30. MDN Web Docs. Cookies [Електронний ресурс]. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies (дата звернення: 27.05.2026).

31. Passport.js Documentation [Електронний ресурс]. URL: https://www.passportjs.org/docs/ (дата звернення: 27.05.2026).

32. Nginx Documentation [Електронний ресурс]. URL: https://nginx.org/en/docs/ (дата звернення: 27.05.2026).

33. Let's Encrypt Documentation [Електронний ресурс]. URL: https://letsencrypt.org/docs/ (дата звернення: 27.05.2026).

34. GitHub Actions Documentation [Електронний ресурс]. URL: https://docs.github.com/actions (дата звернення: 27.05.2026).

Додаток А

Система для координації та взаємодії спільнот любителів настільних ігор

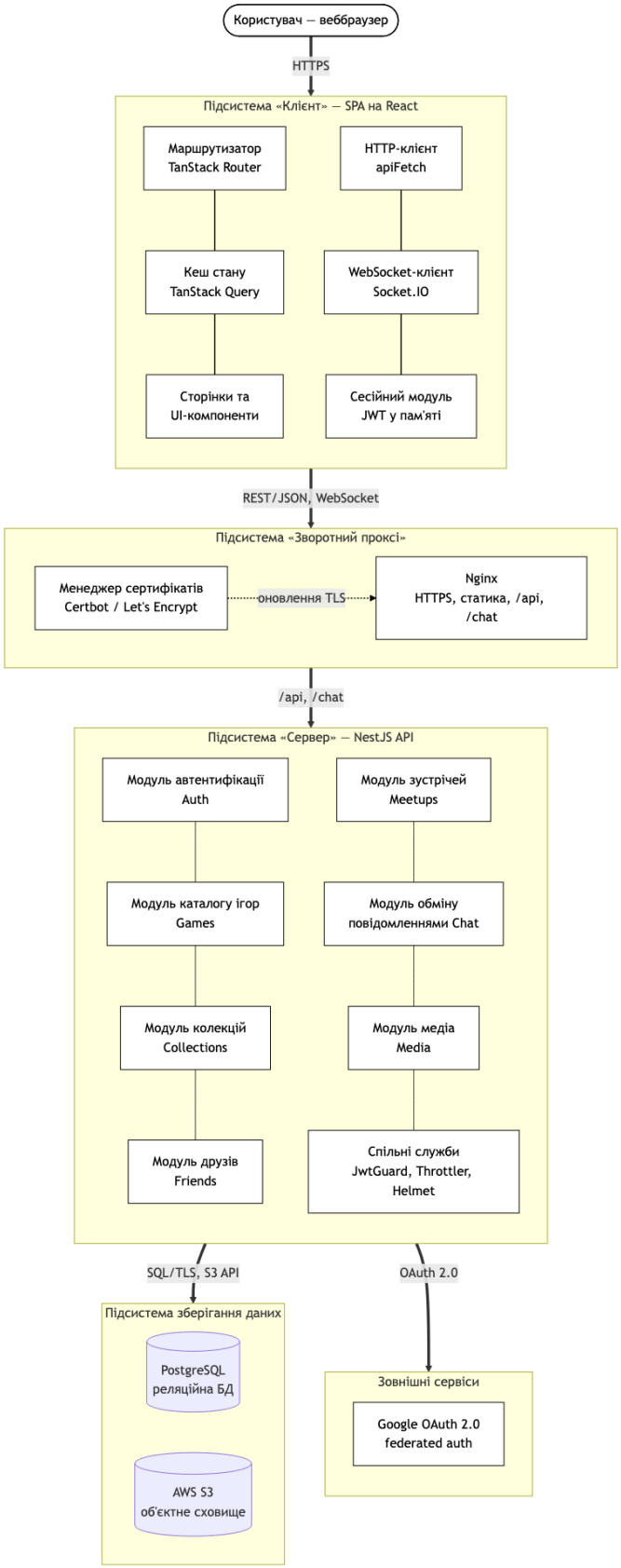
Структурна схема системи

(структурна схема)

ІАЛЦ.467200.004 Д1

Аркушів: 1

Київ 2026 р



					ІАЛЦ.467200.004 Д1
		№ докум.	Підпис	Дата	
Розробив		Русак Я. А.			**Система для координації та взаємодії спільнот любителів настільних ігор** Структурна схема системи (структурна схема)
Перевірив		Ковальчук О. М.			
Н. Контр.		Коренко Д. В.			
Затвердив					

Літ.	Аркуш	Аркушів
	1	1

НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21

Додаток Б

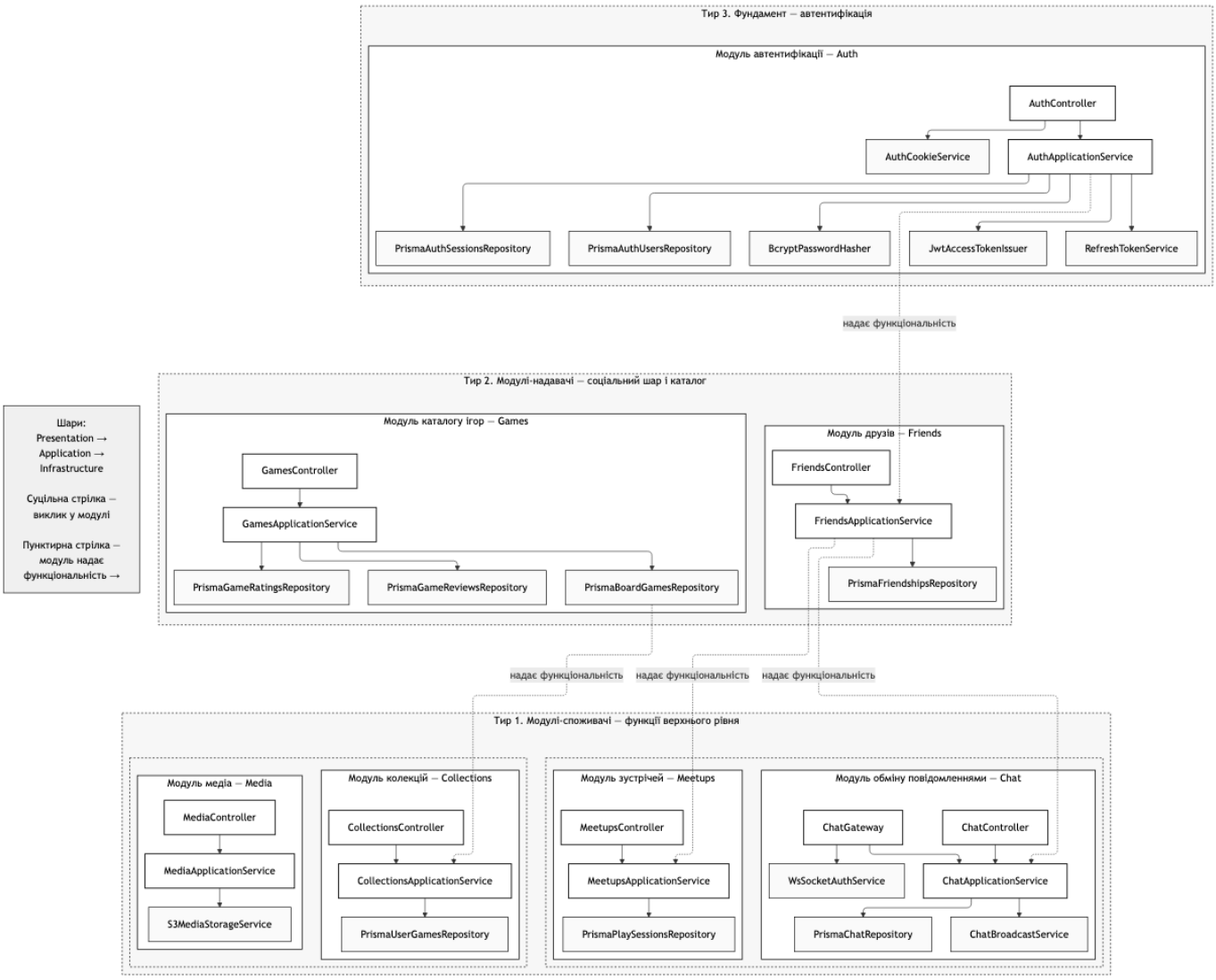
Система для координації та взаємодії спільнот любителів настільних ігор

Діаграма класів
(функціональна схема)

ІАЛЦ.467200.005 Д2

Аркушів: 1

Київ 2026 р



					ІАЛЦ.467200.005 Д2		
		№ докум.	Підпис	Дата			
Розробив		Русак Я. А.			**Система для координації та взаємодії спільнот любителів настільних ігор** Діаграма класів (функціональна схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Ковальчук О. М.					
						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21	
Н. Контр.		Коренко Д. В.					
Затвердив							

Додаток В

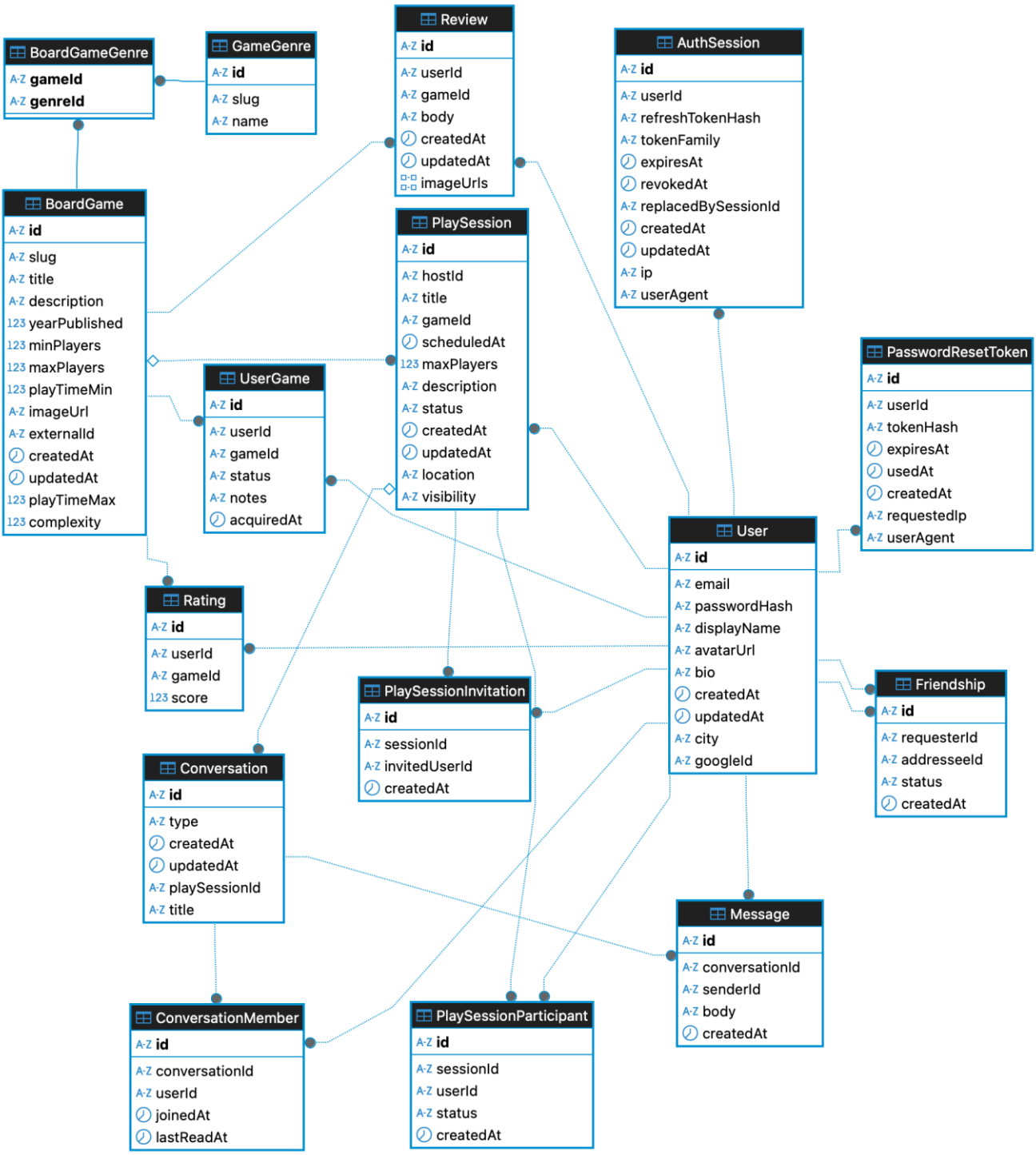
Система для координації та взаємодії спільнот любителів настільних ігор

ER-діаграма бази даних
(принципова схема)

ІАЛЦ.467200.006 Д3

Аркушів: 1

Київ 2026 р



					ІАЛЦ.467200.006 Д3	
		№ докум.	Підпис	Дата		
Розробив		Русак Я. А.			**Система для координації та взаємодії спільнот любителів настільних ігор** ЕР-діаграма бази даних (принципова схема)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Ковальчук О. М.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21
Н. Контр.		Коренко Д. В.				
Затвердив						

Додаток Г

Система для координації та взаємодії спільнот любителів настільних ігор

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів: 68

Київ 2026

apps/api/src/main.ts

```
import { ValidationPipe } from '@nestjs/common';
import { NestFactory } from '@nestjs/core';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';
import { IoAdapter } from '@nestjs/platform-socket.io';
import cookieParser from 'cookie-parser';
import helmet from 'helmet';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useWebSocketAdapter(new IoAdapter(app));
  app.use(cookieParser());
  const allowInsecureHttp = process.env.ALLOW_INSECURE_HTTP === 'true';
  app.use(
    helmet({
      contentSecurityPolicy: {
        useDefaults: true,
        directives: {
          defaultSrc: ["'self'"],
          imgSrc: ["'self'", 'data:', 'https:'],
          scriptSrc: ["'self'"],
          styleSrc: ["'self'", "'unsafe-inline'"],
          connectSrc: ["'self'", ...(process.env.WEB_ORIGIN ?? '').split(',')],
          ...(allowInsecureHttp ? { upgradeInsecureRequests: null } : {}),
        },
      },
      crossOriginResourcePolicy: { policy: 'cross-origin' },
      strictTransportSecurity: allowInsecureHttp ? false : undefined,
    }),
  );
  const webOrigins = (process.env.WEB_ORIGIN ?? 'http://localhost:5173').split(
    ',',
  );
  app.enableCors({
    origin: webOrigins,
    credentials: true,
  });
```

					ІАЛЦ.467200.007 Д4		
		№ докум.	Підпис	Дата			
Розробив		Русак Я. А.			**Система для координації та взаємодії спільнот любителів настільних ігор** Текст програмного коду	Літ.	Аркуш 1 / Аркушів 68
Перевірив		Ковальчук О. М.					
						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21	
Н. Контр.		Коренко Д. В.					
Затвердив							

```
  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true,
      forbidNonWhitelisted: true,
      transform: true,
    }),
  );

  const swaggerConfig = new DocumentBuilder()
    .setTitle('Board game social API')
    .setDescription(
      'REST API for catalog, reviews, ratings, collections, profiles, friends,
chat, meetups, and social features',
    )
    .setVersion('0.1')
    .addBearerAuth(
      { type: 'http', scheme: 'bearer', bearerFormat: 'JWT' },
      'access-token',
    )
    .build();
  const document = SwaggerModule.createDocument(app, swaggerConfig);
  SwaggerModule.setup('docs', app, document);

  await app.listen(process.env.PORT ?? 3000);
}
void bootstrap();
```

apps/api/src/app.module.ts

```
import { Module } from '@nestjs/common';
import { APP_GUARD } from '@nestjs/core';
import { ConfigModule } from '@nestjs/config';
import { ThrottlerGuard, ThrottlerModule } from '@nestjs/throttler';
import { AppApplicationService } from
'./app/application/app.application.service';
import { AppController } from './app/presentation/http/app.controller';
import { AuthModule } from './auth/auth.module';
import { ChatModule } from './chat/chat.module';
import { JwtAuthGuard } from './auth/presentation/http/guards/jwt-auth.guard';
import { CollectionsModule } from './collections/collections.module';
```

```
import { FriendsModule } from './friends/friends.module';
import { GamesModule } from './games/games.module';
import { MeetupsModule } from './meetups/meetups.module';
import { MediaModule } from './media/media.module';
import { PrismaModule } from './prisma/prisma.module';

@Module({
  imports: [
    ConfigModule.forRoot({ isGlobal: true }),
    ThrottlerModule.forRoot([
      {
        ttl: 60_000,
        limit: 120,
      },
    ]),
    PrismaModule,
    AuthModule,
    GamesModule,
    CollectionsModule,
    FriendsModule,
    ChatModule,
    MeetupsModule,
    MediaModule,
  ],
  controllers: [AppController],
  providers: [
    AppApplicationService,
    { provide: APP_GUARD, useClass: JwtAuthGuard },
    { provide: APP_GUARD, useClass: ThrottlerGuard },
  ],
})
export class AppModule {}
```

apps/api/src/auth/application/auth.application.service.ts

```
import {
  ConflictException,
  Injectable,
  Logger,
  NotFoundException,
  UnauthorizedException,
```

```
} from '@nestjs/common';
import type {
  AuthUser,
  PublicProfileSummary,
  PublicUserCard,
} from '@boardgame/shared';
import { BcryptPasswordHasher } from '../infrastructure/crypto/bcrypt-password-hasher';
import { JwtAccessTokenIssuer } from '../infrastructure/jwt/jwt-access-token-issuer';
import { PrismaAuthSessionsRepository } from '../infrastructure/persistence/prisma-auth-sessions.repository';
import { PrismaAuthUsersRepository } from '../infrastructure/persistence/prisma-auth-users.repository';
import { RefreshTokenService } from '../infrastructure/tokens/refresh-token.service';
import { MediaApplicationService } from '../../media/application/media.application.service';

export type RegisterUserProps = {
  email: string;
  password: string;
  displayName: string;
};

export type LoginUserProps = {
  email: string;
  password: string;
};

type AuthRequestMeta = {
  ip: string | null;
  userAgent: string | null;
};

export type LoginWithGoogleProps = {
  googleId: string;
  email: string;
  displayName: string;
  avatarUrl: string | null;
};
```

```
@Injectable()
export class AuthApplicationService {
  private readonly logger = new Logger(AuthApplicationService.name);

  constructor(
    private readonly authUsersRepository: PrismaAuthUsersRepository,
    private readonly authSessionsRepository: PrismaAuthSessionsRepository,
    private readonly passwordHasher: BcryptPasswordHasher,
    private readonly accessTokenIssuer: JwtAccessTokenIssuer,
    private readonly refreshTokenService: RefreshTokenService,
    private readonly mediaApplicationService: MediaApplicationService,
  ) {}

  async register(props: RegisterUserProps, requestMeta: AuthRequestMeta) {
    const email = props.email.toLowerCase();
    const exists = await this.authUsersRepository.existsByEmail(email);
    if (exists) {
      throw new ConflictException('Email already registered');
    }
    const passwordHash = await this.passwordHasher.hash(props.password);
    const avatarUrl = this.mediaApplicationService.getDefaultAvatarUrl(email);
    const user = await this.authUsersRepository.createRegisteredUser({
      email,
      passwordHash,
      displayName: props.displayName.trim(),
      avatarUrl,
    });
    return this.createLoginResponse(user, requestMeta);
  }

  async login(props: LoginUserProps, requestMeta: AuthRequestMeta) {
    const email = props.email.toLowerCase();
    const row =
      await this.authUsersRepository.findWithCredentialsByEmail(email);
    if (!row) {
      throw new UnauthorizedException('Invalid email or password');
    }
    const ok = await this.passwordHasher.verify(
      props.password,
      row.passwordHash,
```

```
    );
    if (!ok) {
      throw new UnauthorizedException('Invalid email or password');
    }
    const { passwordHash, ...user } = row;
    if (row.googleId !== null) {
      throw new UnauthorizedException('Use Google sign-in for this account');
    }
    void passwordHash;
    return this.createLoginResponse(user, requestMeta);
  }

  async loginWithGoogle(
    props: LoginWithGoogleProps,
    requestMeta: AuthRequestMeta,
  ) {
    const email = props.email.toLowerCase();
    const normalizedGoogleAvatarUrl = this.normalizeGoogleAvatarUrl(
      props.avatarUrl,
    );
    let user = await this.authUsersRepository.findByGoogleId(props.googleId);
    if (
      user &&
      normalizedGoogleAvatarUrl &&
      user.avatarUrl !== normalizedGoogleAvatarUrl
    ) {
      user = await this.authUsersRepository.updateProfile(user.id, {
        avatarUrl: normalizedGoogleAvatarUrl,
      });
    }
    if (!user) {
      user = await this.authUsersRepository.linkGoogleAccountByEmail({
        email,
        googleId: props.googleId,
        avatarUrl: normalizedGoogleAvatarUrl,
      });
    }
    if (!user) {
      const passwordHash = await this.passwordHasher.hash(
        this.refreshTokenService.generateOpaqueToken(),
      );
```

```
    user = await this.authUsersRepository.createGoogleUser({
      email,
      googleId: props.googleId,
      displayName: props.displayName.trim() || 'Player',
      avatarUrl:
        normalizedGoogleAvatarUrl ??
        this.mediaApplicationService.getDefaultAvatarUrl(email),
      passwordHash,
    });
  }
  return this.createLoginResponse(user, requestMeta);
}

async refresh(refreshToken: string, requestMeta: AuthRequestMeta) {
  const parsed = this.parseStructuredToken(refreshToken);
  if (!parsed) {
    throw new UnauthorizedException('Invalid refresh token');
  }
  const currentSession = await this.authSessionsRepository.findSessionById(
    parsed.tokenId,
  );
  if (!currentSession) {
    throw new UnauthorizedException('Invalid refresh token');
  }
  if (currentSession.revokedAt || currentSession.expiresAt < new Date()) {
    throw new UnauthorizedException('Refresh token expired');
  }
  const tokenHash = this.refreshTokenService.hashToken(parsed.tokenSecret);
  if (currentSession.refreshTokenHash !== tokenHash) {
    await this.authSessionsRepository.revokeSessionFamily(
      currentSession.tokenFamily,
    );
    throw new UnauthorizedException('Refresh token reuse detected');
  }
  const user = await this.authUsersRepository.findPublicProfileById(
    currentSession.userId,
  );
  if (!user) {
    throw new UnauthorizedException();
  }
```

```
    const nextTokenSecret = this.refreshTokenService.generateOpaqueToken();
    const nextSession = await this.authSessionsRepository.createSession({
      userId: user.id,
      refreshTokenHash: this.refreshTokenService.hashToken(nextTokenSecret),
      tokenFamily: currentSession.tokenFamily,
      expiresAt: this.buildRefreshTokenExpiryDate(),
      ip: requestMeta.ip,
      userAgent: requestMeta.userAgent,
    });
    await this.authSessionsRepository.rotateSession({
      sessionId: currentSession.id,
      nextSessionId: nextSession.id,
    });

    return {
      accessToken: this.accessTokenIssuer.issueAccessToken({
        sub: user.id,
        email: user.email,
        sid: nextSession.id,
      }),
      refreshToken: `${nextSession.id}.${nextTokenSecret}`,
      user,
    };
  }

  async logoutCurrent(accessTokenPayload: { sid?: string }): Promise<void> {
    if (!accessTokenPayload.sid) {
      return;
    }
    await this.authSessionsRepository.revokeSession(accessTokenPayload.sid);
  }

  logoutAll(userId: string): Promise<number> {
    return this.authSessionsRepository.revokeAllUserSessions(userId);
  }

  async forgotPassword(emailRaw: string, requestMeta: AuthRequestMeta) {
    const email = emailRaw.toLowerCase();
    const user = await this.authUsersRepository.findIdByEmail(email);
    if (!user) {
      return { ok: true as const };
```

```
    }

    const secret = this.refreshTokenService.generateOpaqueToken();
    const created = await this.authSessionsRepository.createPasswordResetToken({
      userId: user.id,
      tokenHash: this.refreshTokenService.hashToken(secret),
      expiresAt: new Date(Date.now() + 1000 * 60 * 20),
      requestedIp: requestMeta.ip,
      userAgent: requestMeta.userAgent,
    });
    const resetToken = `${created.id}.${secret}`;
    this.logger.log(
      `DEV password reset token for user ${user.id}: ${resetToken}`,
    );
    return { ok: true as const };
  }

  async resetPassword(token: string, nextPassword: string): Promise<void> {
    const parsed = this.parseStructuredToken(token);
    if (!parsed) {
      throw new UnauthorizedException('Invalid password reset token');
    }
    const row = await this.authSessionsRepository.findActiveResetTokenById(
      parsed.tokenId,
    );
    if (!row || row.expiresAt < new Date()) {
      throw new UnauthorizedException('Password reset token expired');
    }
    const hash = this.refreshTokenService.hashToken(parsed.tokenSecret);
    if (hash !== row.tokenHash) {
      throw new UnauthorizedException('Invalid password reset token');
    }

    const passwordHash = await this.passwordHasher.hash(nextPassword);
    await                this.authUsersRepository.updatePasswordHash(row.userId,
passwordHash);
    await this.authSessionsRepository.markPasswordResetTokenUsed(row.id);
    await this.authSessionsRepository.revokeAllUserSessions(row.userId);
  }

  me(user: AuthUser) {
```

```
    return user;
  }

  async getPublicProfile(userId: string): Promise<PublicUserCard> {
    const card = await this.authUsersRepository.findUserCardById(userId);
    if (!card) {
      throw new NotFoundException('User not found');
    }
    return card;
  }

  async getPublicProfileSummary(userId: string): Promise<PublicProfileSummary>
{
    const summary =
      await this.authUsersRepository.findPublicProfileSummaryById(userId);
    if (!summary) {
      throw new NotFoundException('User not found');
    }
    return summary;
  }

  findPublicProfileById(userId: string): Promise<AuthUser | null> {
    return this.authUsersRepository.findPublicProfileById(userId);
  }

  findPublicUserCardById(userId: string): Promise<PublicUserCard | null> {
    return this.authUsersRepository.findUserCardById(userId);
  }

  searchPublicUserCards(params: {
    q?: string;
    meCity?: string;
    excludeUserId: string;
    skip: number;
    take: number;
  }): Promise<{ items: PublicUserCard[]; total: number }> {
    return this.authUsersRepository.searchPublicUserCards(params);
  }

  async updateMyProfile(
    user: AuthUser,
```

```
  patch: {
    displayName?: string;
    bio?: string;
    city?: string;
    avatarUrl?: string;
  },
): Promise<AuthUser> {
  const data = {
    ...(patch.displayName !== undefined && {
      displayName: patch.displayName.trim(),
    }),
    ...(patch.bio !== undefined && {
      bio: patch.bio.trim() === '' ? null : patch.bio.trim(),
    }),
    ...(patch.city !== undefined && {
      city: patch.city.trim() === '' ? null : patch.city.trim(),
    }),
    ...(patch.avatarUrl !== undefined && {
      avatarUrl:
        patch.avatarUrl.trim() === '' ? null : patch.avatarUrl.trim(),
    }),
  };
  if (Object.keys(data).length === 0) {
    return user;
  }
  return this.authUsersRepository.updateProfile(user.id, data);
}

private async createLoginResponse(
  user: AuthUser,
  requestMeta: AuthRequestMeta,
) {
  const tokenFamily = this.refreshTokenService.generateTokenFamily();
  const tokenSecret = this.refreshTokenService.generateOpaqueToken();
  const session = await this.authSessionsRepository.createSession({
    userId: user.id,
    refreshTokenHash: this.refreshTokenService.hashToken(tokenSecret),
    tokenFamily,
    expiresAt: this.buildRefreshTokenExpiryDate(),
    ip: requestMeta.ip,
    userAgent: requestMeta.userAgent,
```

```
    });

    return {
      accessToken: this.accessTokenIssuer.issueAccessToken({
        sub: user.id,
        email: user.email,
        sid: session.id,
      }),
      refreshToken: `${session.id}.${tokenSecret}`,
      user,
    };
  }

  private buildRefreshTokenExpiryDate(): Date {
    return new Date(Date.now() + 1000 * 60 * 60 * 24 * 14);
  }

  private parseStructuredToken(
    token: string,
  ): { tokenId: string; tokenSecret: string } | null {
    const [tokenId, tokenSecret] = token.split('.');
    if (!tokenId || !tokenSecret) {
      return null;
    }
    return { tokenId, tokenSecret };
  }

  private normalizeGoogleAvatarUrl(url: string | null): string | null {
    if (!url) {
      return null;
    }
    const trimmed = url.trim();
    if (trimmed.length === 0) {
      return null;
    }
    const withProtocol = trimmed.startsWith('//')
      ? `https:${trimmed}`
      : trimmed;
    try {
      const parsed = new URL(withProtocol);
      const isGoogleHost =
```

```
        parsed.hostname.endsWith('googleusercontent.com') ||
        parsed.hostname.endsWith('ggpht.com');
      if (!isGoogleHost) {
        return withProtocol;
      }
      parsed.protocol = 'https:';
      return parsed.toString();
    } catch {
      return null;
    }
  }
}
```

apps/api/src/auth/presentation/http/auth.controller.ts

```
import {
  Body,
  Controller,
  Get,
  Headers,
  Patch,
  Post,
  Req,
  Res,
  UnauthorizedException,
  UseGuards,
} from '@nestjs/common';
import { ApiBearerAuth, ApiOperation, ApiTags } from '@nestjs/swagger';
import { AuthGuard } from '@nestjs/passport';
import { Throttle } from '@nestjs/throttler';
import { AuthApplicationService } from '../../application/auth.application.service';
import {
  AuthCookieService,
  REFRESH_COOKIE_NAME,
} from '../../infrastructure/tokens/auth-cookie.service';
import {
  CurrentUser,
  type AuthUser,
} from './decorators/current-user.decorator';
import { Public } from './decorators/public.decorator';
import { ForgotPasswordDto } from './dto/forgot-password.dto';
import { LoginDto } from './dto/login.dto';
```

```
import { RegisterDto } from './dto/register.dto';
import { ResetPasswordDto } from './dto/reset-password.dto';
import { UpdateProfileDto } from './dto/update-profile.dto';
import type { Request, Response } from 'express';

@ApiTags('auth')
@Controller('auth')
export class AuthController {
  constructor(
    private readonly authApplicationService: AuthApplicationService,
    private readonly authCookieService: AuthCookieService,
  ) {}

  @Public()
  @Get('google')
  @UseGuards(AuthGuard('google'))
  googleAuth(): void {}

  @Public()
  @Get('google/callback')
  @UseGuards(AuthGuard('google'))
  googleCallback(
    @Req()
    request: Request & {
      user?: { accessToken: string; refreshToken: string; user: AuthUser };
    },
    @Res() response: Response,
  ) {
    const auth = request.user;
    if (!auth) {
      throw new UnauthorizedException('Google authentication failed');
    }
    this.authCookieService.setRefreshTokenCookie(response, auth.refreshToken);
    const webOrigin = process.env.WEB_ORIGIN?.split(',')[0]?.trim();
    const target = webOrigin && webOrigin.length > 0 ? webOrigin : '/';
    response.redirect(target);
  }

  @Public()
  @Post('register')
  async register(
```

```
  @Body() dto: RegisterDto,
  @Req() request: Request,
  @Res({ passthrough: true }) response: Response,
) {
  const auth = await this.authApplicationService.register(
    {
      email: dto.email,
      password: dto.password,
      displayName: dto.displayName,
    },
    this.getRequestMeta(request),
  );
  this.authCookieService.setRefreshTokenCookie(response, auth.refreshToken);
  return { accessToken: auth.accessToken, user: auth.user };
}

@Public()
@Post('login')
@Throttle({ default: { limit: 10, ttl: 60_000 } })
async login(
  @Body() dto: LoginDto,
  @Req() request: Request,
  @Res({ passthrough: true }) response: Response,
) {
  const auth = await this.authApplicationService.login(
    {
      email: dto.email,
      password: dto.password,
    },
    this.getRequestMeta(request),
  );
  this.authCookieService.setRefreshTokenCookie(response, auth.refreshToken);
  return { accessToken: auth.accessToken, user: auth.user };
}

@Public()
@Post('refresh')
@Throttle({ default: { limit: 30, ttl: 60_000 } })
async refresh(
  @Req() request: Request,
  @Res({ passthrough: true }) response: Response,
```

```
    @Headers('origin') origin?: string,
  ) {
    this.assertTrustedOrigin(origin);
    const token = this.extractRefreshToken(request);
    if (typeof token !== 'string' || token.length === 0) {
      throw new UnauthorizedException('Missing refresh token');
    }
    const auth = await this.authApplicationService.refresh(
      token,
      this.getRequestMeta(request),
    );
    this.authCookieService.setRefreshTokenCookie(response, auth.refreshToken);
    return { accessToken: auth.accessToken };
  }

  private extractRefreshToken(request: Request): string | null {
    const cookiesUnknown = request.cookies as unknown;
    if (typeof cookiesUnknown !== 'object' || cookiesUnknown === null) {
      return null;
    }
    const record = cookiesUnknown as Record<string, unknown>;
    const value = record[REFRESH_COOKIE_NAME];
    return typeof value === 'string' ? value : null;
  }

  @Post('logout')
  async logout(
    @CurrentUser() user: AuthUser & { sid?: string },
    @Res({ passthrough: true }) response: Response,
  ) {
    await this.authApplicationService.logoutCurrent({ sid: user.sid });
    this.authCookieService.clearRefreshTokenCookie(response);
    return { ok: true, userId: user.id };
  }

  @Post('logout-all')
  async logoutAll(
    @CurrentUser() user: AuthUser,
    @Res({ passthrough: true }) response: Response,
  ) {
    await this.authApplicationService.logoutAll(user.id);
```

```
  this.authCookieService.clearRefreshTokenCookie(response);
  return { ok: true };
}

@Public()
@Post('forgot-password')
@Throttle({ default: { limit: 5, ttl: 60_000 } })
forgotPassword(@Body() dto: ForgotPasswordDto, @Req() request: Request) {
  return this.authApplicationService.forgotPassword(
    dto.email,
    this.getRequestMeta(request),
  );
}

@Public()
@Post('reset-password')
@Throttle({ default: { limit: 5, ttl: 60_000 } })
async resetPassword(@Body() dto: ResetPasswordDto) {
  await this.authApplicationService.resetPassword(dto.token, dto.password);
  return { ok: true };
}

@Get('me')
@ApiOperation({ summary: 'Current user (full profile)' })
me(@CurrentUser() user: AuthUser) {
  return this.authApplicationService.me(user);
}

@Patch('me')
@ApiBearerAuth('access-token')
@ApiOperation({ summary: 'Update my profile' })
patchMe(@CurrentUser() user: AuthUser, @Body() dto: UpdateProfileDto) {
  return this.authApplicationService.updateMyProfile(user, dto);
}

private getRequestMeta(request: Request): {
  ip: string | null;
  userAgent: string | null;
} {
  return {
    ip: request.ip ?? null,
```

```
      userAgent:
        typeof request.headers['user-agent'] === 'string'
          ? request.headers['user-agent']
          : null,
    };
  }

  private assertTrustedOrigin(origin?: string): void {
    if (!origin) {
      return;
    }
    const allowed = (process.env.WEB_ORIGIN ?? 'http://localhost:5173')
      .split(',')
      .map((entry) => entry.trim());
    if (!allowed.includes(origin)) {
      throw new UnauthorizedException('Untrusted origin');
    }
  }
}
```

apps/api/src/meetups/application/meetups.application.service.ts

```
import {
  BadRequestException,
  ConflictException,
  ForbiddenException,
  Injectable,
  NotFoundException,
} from '@nestjs/common';
import { FriendsApplicationService } from '../../friends/application/friends.application.service';
import type {
  MeetupDetail,
  MeetupListItem,
  PlaySessionStatus,
  PlaySessionVisibility,
} from '@boardgame/shared';
import { PrismaPlaySessionsRepository } from '../infrastructure/persistence/prisma-play-sessions.repository';

@Injectable()
```

```
export class MeetupsApplicationService {
  constructor(
    private readonly playSessionsRepository: PrismaPlaySessionsRepository,
    private readonly friendsApplicationService: FriendsApplicationService,
  ) {}

  async listMeetups(params: {
    userId?: string;
    status?: PlaySessionStatus;
    upcomingOnly?: boolean;
    scheduledFrom?: Date;
    scheduledTo?: Date;
    gameId?: string;
    visibilityScope?: 'ALL' | 'PUBLIC' | 'FRIENDS';
    joinedByUserId?: string;
    titleContains?: string;
    page: number;
    limit: number;
  }): Promise<{
    data: MeetupListItem[];
    meta: { total: number; page: number; limit: number };
  }> {
    let scheduledFrom: Date | undefined;
    let scheduledTo: Date | undefined;
    if (params.scheduledFrom || params.scheduledTo) {
      scheduledFrom = params.scheduledFrom;
      scheduledTo = params.scheduledTo;
    } else if (params.upcomingOnly) {
      scheduledFrom = new Date();
    }
    const skip = (params.page - 1) * params.limit;
    const status = params.status ?? 'SCHEDULED';
    const visibilityScope = params.visibilityScope ?? 'ALL';
    const titleContains = params.titleContains?.trim() || undefined;
    const { items, total } = params.userId
      ? await this.playSessionsRepository.findManyVisibleToUser({
          userId: params.userId,
          status,
          ...(scheduledFrom ? { scheduledFrom } : {}),
          ...(scheduledTo ? { scheduledTo } : {}),
          ...(params.gameId?.trim() ? { gameId: params.gameId.trim() } : {}),
```

```
        visibilityScope,
        ...(params.joinedByUserId
          ? { joinedByUserId: params.joinedByUserId }
          : {}),
        ...(titleContains ? { titleContains } : {}),
        skip,
        take: params.limit,
      })
    : await this.playSessionsRepository.findManyForList({
        status,
        visibility: 'PUBLIC',
        ...(scheduledFrom ? { scheduledFrom } : {}),
        ...(scheduledTo ? { scheduledTo } : {}),
        ...(params.gameId?.trim() ? { gameId: params.gameId.trim() } : {}),
        ...(titleContains ? { titleContains } : {}),
        skip,
        take: params.limit,
      });
  return {
    data: items,
    meta: { total, page: params.page, limit: params.limit },
  };
}

async getMeetup(id: string, requesterId?: string): Promise<MeetupDetail> {
  const row = await this.playSessionsRepository.findDetailById(id);
  if (!row) {
    throw new NotFoundException('Meetup not found');
  }
  const canSee = await this.canAccessSession(row, requesterId);
  if (!canSee) {
    throw new NotFoundException('Meetup not found');
  }
  return row;
}

async createMeetup(
  hostId: string,
  props: {
    title: string;
    scheduledAt: Date;
```

```
    gameId?: string | null;
    location?: string | null;
    maxPlayers?: number | null;
    description?: string | null;
    visibility?: PlaySessionVisibility;
  },
): Promise<MeetupDetail> {
  if (props.gameId) {
    const ok = await this.playSessionsRepository.gameExists(props.gameId);
    if (!ok) {
      throw new NotFoundException('Game not found');
    }
  }
  return this.playSessionsRepository.create({
    hostId,
    title: props.title.trim(),
    scheduledAt: props.scheduledAt,
    gameId: props.gameId ?? null,
    location: props.location?.trim() || null,
    maxPlayers: props.maxPlayers ?? null,
    description: props.description?.trim() || null,
    visibility: props.visibility ?? 'PUBLIC',
  });
}

async updateMeetup(
  userId: string,
  id: string,
  patch: {
    title?: string;
    scheduledAt?: Date;
    gameId?: string | null;
    location?: string | null;
    maxPlayers?: number | null;
    description?: string | null;
    visibility?: PlaySessionVisibility;
  },
): Promise<MeetupDetail> {
  const hostId = await this.playSessionsRepository.getHostId(id);
  if (!hostId) {
    throw new NotFoundException('Meetup not found');
```

```
    }
    if (hostId !== userId) {
      throw new ForbiddenException('Only the host can update this meetup');
    }
    const meta = await this.playSessionsRepository.getSessionMeta(id);
    if (!meta || meta.status !== 'SCHEDULED') {
      throw new ConflictException('This meetup can no longer be edited');
    }
    if (patch.gameId) {
      const ok = await this.playSessionsRepository.gameExists(patch.gameId);
      if (!ok) {
        throw new NotFoundException('Game not found');
      }
    }
    const repoPatch = {
      ...(patch.title !== undefined && { title: patch.title.trim() }),
      ...(patch.scheduledAt !== undefined && {
        scheduledAt: patch.scheduledAt,
      }),
      ...(patch.gameId !== undefined && { gameId: patch.gameId }),
      ...(patch.location !== undefined && {
        location: patch.location?.trim() || null,
      }),
      ...(patch.maxPlayers !== undefined && { maxPlayers: patch.maxPlayers }),
      ...(patch.description !== undefined && {
        description: patch.description?.trim() || null,
      }),
      ...(patch.visibility !== undefined && { visibility: patch.visibility }),
    };
    if (Object.keys(repoPatch).length === 0) {
      return this.getMeetup(id, userId);
    }
    const updated = await this.playSessionsRepository.updateById(id, repoPatch);
    if (!updated) {
      throw new NotFoundException('Meetup not found');
    }
    return updated;
  }

  async cancelMeetup(userId: string, id: string): Promise<MeetupDetail> {
    const hostId = await this.playSessionsRepository.getHostId(id);
```

```
    if (!hostId) {
      throw new NotFoundException('Meetup not found');
    }
    if (hostId !== userId) {
      throw new ForbiddenException('Only the host can cancel this meetup');
    }
    const meta = await this.playSessionsRepository.getSessionMeta(id);
    if (!meta || meta.status !== 'SCHEDULED') {
      throw new ConflictException('This meetup is not scheduled');
    }
    const updated = await this.playSessionsRepository.setStatus(
      id,
      'CANCELLED',
    );
    if (!updated) {
      throw new NotFoundException('Meetup not found');
    }
    return updated;
  }

  async joinMeetup(userId: string, id: string): Promise<MeetupDetail> {
    const meta = await this.playSessionsRepository.getSessionMeta(id);
    if (!meta || meta.status !== 'SCHEDULED') {
      throw new NotFoundException('Meetup not found');
    }
    if (meta.hostId === userId) {
      throw new BadRequestException('You are already hosting this meetup');
    }
    if (meta.visibility === 'FRIENDS') {
      const areFriends =
        await this.friendsApplicationService.areAcceptedFriends(
          userId,
          meta.hostId,
        );
      if (!areFriends) {
        throw new ForbiddenException('This meetup is for friends of the host');
      }
    }
    if (meta.visibility === 'INVITE_ONLY') {
      const invited = await this.playSessionsRepository.hasInvitation(
        id,
```

```
      userId,
    );
    if (!invited) {
      throw new ForbiddenException(
        'You need an invitation to join this meetup',
      );
    }
  }
  const existing = await this.playSessionsRepository.findParticipantStatus(
    id,
    userId,
  );
  if (existing) {
    throw new ConflictException('Already joined this meetup');
  }
  const joined =
    await this.playSessionsRepository.countJoinedParticipants(id);
  if (meta.maxPlayers != null && 1 + joined >= meta.maxPlayers) {
    throw new ConflictException('This meetup is full');
  }
  await this.playSessionsRepository.addParticipant(id, userId);
  return this.getMeetup(id, userId);
}

async leaveMeetup(userId: string, id: string): Promise<MeetupDetail> {
  const meta = await this.playSessionsRepository.getSessionMeta(id);
  if (!meta) {
    throw new NotFoundException('Meetup not found');
  }
  if (meta.hostId === userId) {
    throw new BadRequestException(
      'Host cannot leave — cancel the meetup instead',
    );
  }
  const ok = await this.playSessionsRepository.removeParticipant(id, userId);
  if (!ok) {
    throw new NotFoundException('You are not in this meetup');
  }
  return this.getMeetup(id, userId);
}
```

```
async inviteToMeetup(
  hostId: string,
  meetupId: string,
  invitedUserId: string,
): Promise<void> {
  if (hostId === invitedUserId) {
    throw new BadRequestException('Host cannot invite themselves');
  }
  const meta = await this.playSessionsRepository.getSessionMeta(meetupId);
  if (!meta) {
    throw new NotFoundException('Meetup not found');
  }
  if (meta.hostId !== hostId) {
    throw new ForbiddenException('Only the host can invite players');
  }
  if (meta.status !== 'SCHEDULED') {
    throw new ConflictException('This meetup can no longer be updated');
  }
  const alreadyParticipant =
    await this.playSessionsRepository.findParticipantStatus(
      meetupId,
      invitedUserId,
    );
  if (alreadyParticipant) {
    throw new ConflictException('User is already in this meetup');
  }
  const hasInvite = await this.playSessionsRepository.hasInvitation(
    meetupId,
    invitedUserId,
  );
  if (hasInvite) {
    throw new ConflictException('User already invited');
  }
  await this.playSessionsRepository.addInvitation(meetupId, invitedUserId);
}

private async canAccessSession(
  meetup: MeetupDetail,
  requesterId?: string,
): Promise<boolean> {
  if (meetup.visibility === 'PUBLIC') {
```

```
      return true;
    }
    if (!requesterId) {
      return false;
    }
    if (meetup.host.id === requesterId) {
      return true;
    }
    if (
      meetup.participants.some(
        (participant) => participant.userId === requesterId,
      )
    ) {
      return true;
    }
    if (meetup.visibility === 'FRIENDS') {
      return this.friendsApplicationService.areAcceptedFriends(
        requesterId,
        meetup.host.id,
      );
    }
    return this.playSessionsRepository.hasInvitation(meetup.id, requesterId);
  }
}
```

apps/api/src/meetups/presentation/http/meetups.controller.ts

```
import {
  Body,
  Controller,
  Get,
  Param,
  Patch,
  Post,
  Query,
} from '@nestjs/common';
import {
  ApiBearerAuth,
  ApiOperation,
  ApiParam,
  ApiTags,
```

```
} from '@nestjs/swagger';
import {
  CurrentUser,
  type AuthUser,
} from '../../../auth/presentation/http/decorators/current-user.decorator';
import { Public } from '../../../auth/presentation/http/decorators/public.decorator';
import { MeetupsApplicationService } from '../../application/meetups.application.service';
import { CreateMeetupInvitationDto } from './dto/create-meetup-invitation.dto';
import { CreateMeetupDto } from './dto/create-meetup.dto';
import { QueryMeetupsDto } from './dto/query-meetups.dto';
import { UpdateMeetupDto } from './dto/update-meetup.dto';

@ApiTags('meetups')
@Controller('meetups')
export class MeetupsController {
  constructor(
    private readonly meetupsApplicationService: MeetupsApplicationService,
  ) {}

  @Get()
  @ApiBearerAuth('access-token')
  @ApiOperation({ summary: 'List meetups visible to current user' })
  list(@CurrentUser() user: AuthUser, @Query() query: QueryMeetupsDto) {
    const upcomingOnly = query.upcoming !== 'false';
    const scheduledFrom = query.from ? new Date(query.from) : undefined;
    const scheduledTo = query.to ? new Date(query.to) : undefined;
    return this.meetupsApplicationService.listMeetups({
      userId: user.id,
      ...(query.status !== undefined ? { status: query.status } : {}),
      upcomingOnly,
      ...(scheduledFrom ? { scheduledFrom } : {}),
      ...(scheduledTo ? { scheduledTo } : {}),
      ...(query.gameId !== undefined ? { gameId: query.gameId } : {}),
      ...(query.visibility !== undefined
        ? { visibilityScope: query.visibility }
        : {}),
      ...(query.joined === 'me' ? { joinedByUserId: user.id } : {}),
      ...(query.q !== undefined ? { titleContains: query.q } : {}),
      page: query.page ?? 1,
```

```
    limit: query.limit ?? 20,
  });
}

@Public()
@Get('public')
@ApiOperation({ summary: 'List public meetups only' })
listPublic(@Query() query: QueryMeetupsDto) {
  const upcomingOnly = query.upcoming !== 'false';
  const scheduledFrom = query.from ? new Date(query.from) : undefined;
  const scheduledTo = query.to ? new Date(query.to) : undefined;
  return this.meetupsApplicationService.listMeetups({
    ...(query.status !== undefined ? { status: query.status } : {}),
    upcomingOnly,
    ...(scheduledFrom ? { scheduledFrom } : {}),
    ...(scheduledTo ? { scheduledTo } : {}),
    ...(query.gameId !== undefined ? { gameId: query.gameId } : {}),
    ...(query.q !== undefined ? { titleContains: query.q } : {}),
    page: query.page ?? 1,
    limit: query.limit ?? 20,
  });
}

@Post()
@ApiBearerAuth('access-token')
@ApiOperation({ summary: 'Create a meetup (you are the host)' })
create(@CurrentUser() user: AuthUser, @Body() dto: CreateMeetupDto) {
  return this.meetupsApplicationService.createMeetup(user.id, {
    title: dto.title,
    scheduledAt: new Date(dto.scheduledAt),
    gameId: dto.gameId,
    location: dto.location,
    maxPlayers: dto.maxPlayers,
    description: dto.description,
    visibility: dto.visibility,
  });
}

@Post(':id/cancel')
@ApiBearerAuth('access-token')
@ApiOperation({ summary: 'Cancel meetup (host only)' })
```

```
@ApiParam({ name: 'id' })
cancel(@CurrentUser() user: AuthUser, @Param('id') id: string) {
  return this.meetupsApplicationService.cancelMeetup(user.id, id);
}

@Post(':id/join')
@ApiBearerAuth('access-token')
@ApiOperation({ summary: 'Join a meetup as a player' })
@ApiParam({ name: 'id' })
join(@CurrentUser() user: AuthUser, @Param('id') id: string) {
  return this.meetupsApplicationService.joinMeetup(user.id, id);
}

@Post(':id/invitations')
@ApiBearerAuth('access-token')
@ApiOperation({
  summary: 'Invite a user to meetup (host only, required for invite-only)',
})
@ApiParam({ name: 'id' })
invite(
  @CurrentUser() user: AuthUser,
  @Param('id') id: string,
  @Body() dto: CreateMeetupInvitationDto,
) {
  return this.meetupsApplicationService.inviteToMeetup(
    user.id,
    id,
    dto.userId,
  );
}

@Post(':id/leave')
@ApiBearerAuth('access-token')
@ApiOperation({ summary: 'Leave a meetup (not the host)' })
@ApiParam({ name: 'id' })
leave(@CurrentUser() user: AuthUser, @Param('id') id: string) {
  return this.meetupsApplicationService.leaveMeetup(user.id, id);
}

@Patch(':id')
@ApiBearerAuth('access-token')
```

```
  @ApiOperation({ summary: 'Update meetup (host only, scheduled only)' })
  @ApiParam({ name: 'id' })
  update(
    @CurrentUser() user: AuthUser,
    @Param('id') id: string,
    @Body() dto: UpdateMeetupDto,
  ) {
    return this.meetupsApplicationService.updateMeetup(user.id, id, {
      ...(dto.title !== undefined && { title: dto.title }),
      ...(dto.scheduledAt !== undefined && {
        scheduledAt: new Date(dto.scheduledAt),
      }),
      ...(dto.gameId !== undefined && {
        gameId: dto.gameId.trim() === '' ? null : dto.gameId.trim(),
      }),
      ...(dto.location !== undefined && { location: dto.location }),
      ...(dto.maxPlayers !== undefined && { maxPlayers: dto.maxPlayers }),
      ...(dto.description !== undefined && { description: dto.description }),
      ...(dto.visibility !== undefined && { visibility: dto.visibility }),
    });
  }

  @Public()
  @Get('public/:id')
  @ApiOperation({ summary: 'Public meetup detail (public meetups only)' })
  @ApiParam({ name: 'id' })
  getOnePublic(@Param('id') id: string) {
    return this.meetupsApplicationService.getMeetup(id);
  }

  @ApiBearerAuth('access-token')
  @Get(':id')
  @ApiOperation({ summary: 'Meetup detail' })
  @ApiParam({ name: 'id' })
  getOne(@CurrentUser() user: AuthUser, @Param('id') id: string) {
    return this.meetupsApplicationService.getMeetup(id, user.id);
  }
}
```

apps/api/src/chat/application/chat.application.service.ts

```
import {
  BadRequestException,
  ConflictException,
  ForbiddenException,
  Injectable,
  NotFoundException,
} from '@nestjs/common';
import { FriendsApplicationService } from '../../friends/application/friends.application.service';
import { PrismaChatRepository } from '../infrastructure/persistence/prisma-chat.repository';
import { ChatBroadcastService } from '../infrastructure/realtime/chat-broadcast.service';
import type {
  ConversationIdResponse,
  ConversationListItem,
  ConversationMessages,
  MessageView,
} from '@boardgame/shared';

@Injectable()
export class ChatApplicationService {
  constructor(
    private readonly chatRepository: PrismaChatRepository,
    private readonly friendsApplicationService: FriendsApplicationService,
    private readonly chatBroadcastService: ChatBroadcastService,
  ) {}

  listConversations(userId: string): Promise<ConversationListItem[]> {
    return this.chatRepository.listConversationsForUser(userId);
  }

  async getOrCreateDirectConversation(
    meId: string,
    otherUserId: string,
  ): Promise<ConversationIdResponse> {
    if (meId === otherUserId) {
      throw new BadRequestException('Invalid recipient');
    }
    const friends = await this.friendsApplicationService.areAcceptedFriends(
      meId,
```

```
      otherUserId,
    );
    if (!friends) {
      throw new ForbiddenException('You can only message accepted friends');
    }
    let id = await this.chatRepository.findDirectConversationIdBetween(
      meId,
      otherUserId,
    );
    if (!id) {
      id = await this.chatRepository.createDirectConversation(
        meId,
        otherUserId,
      );
    }
    return { conversationId: id };
  }

  async createGroupConversation(
    creatorId: string,
    memberIds: string[],
    title?: string,
  ): Promise<ConversationIdResponse> {
    const normalizedMemberIds = Array.from(
      new Set(memberIds.map((id) => id.trim()).filter(Boolean)),
    ).filter((id) => id !== creatorId);
    if (normalizedMemberIds.length === 0) {
      throw new BadRequestException('At least one other member is required');
    }
    const normalizedTitle = title?.trim();
    if (!normalizedTitle) {
      throw new BadRequestException('Group chat title is required');
    }
    const areFriends = await Promise.all(
      normalizedMemberIds.map((id) =>
        this.friendsApplicationService.areAcceptedFriends(creatorId, id),
      ),
    );
    if (areFriends.some((ok) => !ok)) {
      throw new ForbiddenException(
        'You can only add accepted friends to group chats',
```

```
    );
  }
  const conversationId = await this.chatRepository.createGroupConversation({
    creatorId,
    memberIds: normalizedMemberIds,
    title: normalizedTitle,
  });
  return { conversationId };
}

async listMessages(
  userId: string,
  conversationId: string,
  page: number,
  limit: number,
): Promise<ConversationMessages> {
  const member = await this.chatRepository.isMember(userId, conversationId);
  if (!member) {
    throw new NotFoundException('Conversation not found');
  }
  const skip = (page - 1) * limit;
  const [{ items, total }, members, conversation] = await Promise.all([
    this.chatRepository.listMessages({
      conversationId,
      skip,
      take: limit,
    }),
    this.chatRepository.listConversationMembers(conversationId),
    this.chatRepository.getConversationSummary(conversationId),
  ]);
  if (!conversation) {
    throw new NotFoundException('Conversation not found');
  }
  return {
    conversation,
    data: items,
    members,
    meta: { total, page, limit },
  };
}
```

```
async sendMessage(
  userId: string,
  conversationId: string,
  body: string,
): Promise<MessageView> {
  const member = await this.chatRepository.isMember(userId, conversationId);
  if (!member) {
    throw new NotFoundException('Conversation not found');
  }
  const trimmed = body.trim();
  if (!trimmed) {
    throw new BadRequestException('Message cannot be empty');
  }
  const message = await this.chatRepository.createMessage({
    conversationId,
    senderId: userId,
    body: trimmed,
  });
  this.chatBroadcastService.emitNewMessage(conversationId, message);
  return message;
}

async inviteToConversation(
  inviterId: string,
  conversationId: string,
  invitedUserId: string,
): Promise<void> {
  const userId = invitedUserId.trim();
  if (!userId) {
    throw new BadRequestException('Invalid user');
  }
  if (userId === inviterId) {
    throw new BadRequestException('Cannot invite yourself');
  }
  const member = await this.chatRepository.isMember(
    inviterId,
    conversationId,
  );
  if (!member) {
    throw new NotFoundException('Conversation not found');
  }
```

```
    const conversation =
      await this.chatRepository.getConversationSummary(conversationId);
    if (!conversation) {
      throw new NotFoundException('Conversation not found');
    }
    if (conversation.type !== 'GROUP') {
      throw new BadRequestException(
        'Members can be invited only to group chats',
      );
    }
    const alreadyMember = await this.chatRepository.isMember(
      userId,
      conversationId,
    );
    if (alreadyMember) {
      throw new ConflictException('User is already in this conversation');
    }
    const areFriends = await this.friendsApplicationService.areAcceptedFriends(
      inviterId,
      userId,
    );
    if (!areFriends) {
      throw new ForbiddenException('You can only invite accepted friends');
    }
    await this.chatRepository.addConversationMember(conversationId, userId);
  }

  async assertMember(userId: string, conversationId: string): Promise<boolean>
{
    return this.chatRepository.isMember(userId, conversationId);
  }
}
```

apps/api/src/chat/presentation/ws/chat.gateway.ts

```
import { Logger } from '@nestjs/common';
import {
  ConnectedSocket,
  MessageBody,
  OnGatewayConnection,
  OnGatewayInit,
```

```
  SubscribeMessage,
  WebSocketGateway,
  WebSocketServer,
} from '@nestjs/websockets';
import type { Server, Socket } from 'socket.io';
import { WsSocketAuthService } from '../../../auth/application/ws-socket-auth.service';
import { ChatApplicationService } from '../../application/chat.application.service';
import { ChatBroadcastService } from '../../infrastructure/realtime/chat-broadcast.service';
import { ChatRoomDto } from './dto/chat-room.dto';

const webOrigins = (process.env.WEB_ORIGIN ?? 'http://localhost:5173').split(
  ',',
);

@WebSocketGateway({
  namespace: '/chat',
  cors: {
    origin: webOrigins,
    credentials: true,
  },
})
export class ChatGateway implements OnGatewayInit, OnGatewayConnection {
  @WebSocketServer() server!: Server;

  private readonly logger = new Logger(ChatGateway.name);

  constructor(
    private readonly wsSocketAuthService: WsSocketAuthService,
    private readonly chatApplicationService: ChatApplicationService,
    private readonly chatBroadcastService: ChatBroadcastService,
  ) {}

  afterInit(server: Server): void {
    this.chatBroadcastService.registerServer(server);
  }

  async handleConnection(client: Socket): Promise<void> {
    const user = await this.wsSocketAuthService.authenticate(client);
```

```
    if (!user) {
      this.logger.warn('Socket connection rejected: no token');
      client.disconnect();
    }
  }

  @SubscribeMessage('join')
  async join(
    @ConnectedSocket() client: Socket,
    @MessageBody() body: ChatRoomDto,
  ): Promise<{ ok: boolean; error?: string }> {
    const user = await this.wsSocketAuthService.authenticate(client);
    if (!user?.id) {
      return { ok: false, error: 'Unauthorized' };
    }
    const ok = await this.chatApplicationService.assertMember(
      user.id,
      body.conversationId,
    );
    if (!ok) {
      return { ok: false, error: 'Forbidden' };
    }
    await client.join(`conv:${body.conversationId}`);
    return { ok: true };
  }

  @SubscribeMessage('leave')
  async leave(
    @ConnectedSocket() client: Socket,
    @MessageBody() body: ChatRoomDto,
  ): Promise<{ ok: boolean }> {
    const user = await this.wsSocketAuthService.authenticate(client);
    if (!user?.id) {
      return { ok: false };
    }
    await client.leave(`conv:${body.conversationId}`);
    return { ok: true };
  }
}
```

apps/api/src/friends/application/friends.application.service.ts

```
import {
  BadRequestException,
  ConflictException,
  Injectable,
  NotFoundException,
} from '@nestjs/common';
import { AuthApplicationService } from '../../auth/application/auth.application.service';
import type {
  DiscoverUserRow,
  FriendConnection,
  FriendshipRelationship,
  PendingRequest,
} from '@boardgame/shared';
import { PrismaFriendshipsRepository } from '../infrastructure/persistence/prisma-friendships.repository';

@Injectable()
export class FriendsApplicationService {
  constructor(
    private readonly authApplicationService: AuthApplicationService,
    private readonly friendshipsRepository: PrismaFriendshipsRepository,
  ) {}

  private relationshipFromRow(
    meId: string,
    row: { requesterId: string; addresseeId: string; status: string },
  ): FriendshipRelationship {
    if (row.status === 'BLOCKED') {
      return 'blocked';
    }
    if (row.status === 'ACCEPTED') {
      return 'friend';
    }
    if (row.status === 'PENDING') {
      return row.requesterId === meId ? 'outgoing_pending' : 'incoming_pending';
    }
    return 'none';
  }
```

```
async discover(
  meId: string,
  q: string,
  page: number,
  limit: number,
): Promise<{
  data: DiscoverUserRow[];
  meta: { total: number; page: number; limit: number };
}> {
  const skip = (page - 1) * limit;
  const me = await this.authApplicationService.findPublicProfileById(meId);
  const { items, total } =
    await this.authApplicationService.searchPublicUserCards({
      q,
      meCity: me?.city?.trim() || undefined,
      excludeUserId: meId,
      skip,
      take: limit,
    });
  if (items.length === 0) {
    return { data: [], meta: { total, page, limit } };
  }
  const ids = items.map((u) => u.id);
  const rows =
    await this.friendshipsRepository.findManyBetweenUserAndCandidates(
      meId,
      ids,
    );
  const byOther = new Map<string, (typeof rows)[0]>();
  for (const r of rows) {
    const otherId = r.requesterId === meId ? r.addresseeId : r.requesterId;
    byOther.set(otherId, r);
  }
  const data: DiscoverUserRow[] = items.map((user) => {
    const row = byOther.get(user.id);
    return {
      ...user,
      relationship: row ? this.relationshipFromRow(meId, row) : 'none',
    };
  });
  return { data, meta: { total, page, limit } };
```

```
}

async listFriends(meId: string): Promise<FriendConnection[]> {
  const rows = await this.friendshipsRepository.listAcceptedFriends(meId);
  return rows.map((r) => ({
    friendshipId: r.friendshipId,
    user: r.otherUser,
    friendsSince: r.friendsSince.toISOString(),
  }));
}

async listIncomingRequests(meId: string): Promise<PendingRequest[]> {
  const rows = await this.friendshipsRepository.listIncomingPending(meId);
  return rows.map((r) => ({
    friendshipId: r.friendshipId,
    user: r.requester,
    createdAt: r.createdAt.toISOString(),
  }));
}

async listOutgoingRequests(meId: string): Promise<PendingRequest[]> {
  const rows = await this.friendshipsRepository.listOutgoingPending(meId);
  return rows.map((r) => ({
    friendshipId: r.friendshipId,
    user: r.addressee,
    createdAt: r.createdAt.toISOString(),
  }));
}

async sendFriendRequest(meId: string, toUserId: string): Promise<void> {
  if (meId === toUserId) {
    throw new BadRequestException('Cannot friend yourself');
  }
  const target =
    await this.authApplicationService.findPublicUserCardById(toUserId);
  if (!target) {
    throw new NotFoundException('User not found');
  }
  const existing = await this.friendshipsRepository.findPair(meId, toUserId);
  if (!existing) {
    await this.friendshipsRepository.createPendingRequest(meId, toUserId);
```

```
      return;
    }
    if (existing.status === 'BLOCKED') {
      throw new ConflictException('Cannot send a friend request');
    }
    if (existing.status === 'ACCEPTED') {
      throw new ConflictException('Already friends');
    }
    if (existing.status === 'PENDING') {
      if (existing.requesterId === meId) {
        throw new ConflictException('Request already sent');
      }
      const ok = await this.friendshipsRepository.acceptPendingRequest(
        toUserId,
        meId,
      );
      if (!ok) {
        throw new ConflictException('Could not accept request');
      }
    }
  }

  async acceptRequest(meId: string, fromUserId: string): Promise<void> {
    const row = await this.friendshipsRepository.findPair(meId, fromUserId);
    if (
      !row ||
      row.status !== 'PENDING' ||
      row.requesterId !== fromUserId ||
      row.addresseeId !== meId
    ) {
      throw new NotFoundException('No pending request from this user');
    }
    const ok = await this.friendshipsRepository.acceptPendingRequest(
      fromUserId,
      meId,
    );
    if (!ok) {
      throw new ConflictException('Could not accept request');
    }
  }
```

```
  async declineIncoming(meId: string, fromUserId: string): Promise<void> {
    const ok = await this.friendshipsRepository.deleteByPair(fromUserId, meId);
    if (!ok) {
      throw new NotFoundException('No pending request from this user');
    }
  }

  async cancelOutgoing(meId: string, toUserId: string): Promise<void> {
    const ok = await this.friendshipsRepository.deleteByPair(meId, toUserId);
    if (!ok) {
      throw new NotFoundException('No outgoing request to this user');
    }
  }

  async unfriend(meId: string, otherUserId: string): Promise<void> {
    if (meId === otherUserId) {
      throw new BadRequestException('Invalid user');
    }
    const ok = await this.friendshipsRepository.deleteAcceptedFriendshipBetween(
      meId,
      otherUserId,
    );
    if (!ok) {
      throw new NotFoundException('Not friends with this user');
    }
  }

  async areAcceptedFriends(userIdA: string, userIdB: string): Promise<boolean>
{
    if (userIdA === userIdB) {
      return false;
    }
    const row = await this.friendshipsRepository.findPair(userIdA, userIdB);
    return row?.status === 'ACCEPTED';
  }
}
```

apps/api/src/games/application/games.application.service.ts

```
import {
  ConflictException,
```

```
  Injectable,
  NotFoundException,
} from '@nestjs/common';
import type { AuthUser, OkResponse } from '@boardgame/shared';
import type {
  CreateGameProps,
  UpdateGamePatch,
} from '../domain/types/game.types';
import { PrismaBoardGamesRepository } from '../infrastructure/persistence/prisma-board-games.repository';
import { PrismaGameRatingsRepository } from '../infrastructure/persistence/prisma-game-ratings.repository';
import {
  DuplicateGameReviewError,
  PrismaGameReviewsRepository,
} from '../infrastructure/persistence/prisma-game-reviews.repository';
import {
  parseComplexityBands,
  parseGenreSlugs,
  parsePlayerCounts,
} from '../infrastructure/persistence/games-list-query.util';

export type ListGamesParams = {
  titleSearch?: string;
  page: number;
  limit: number;
  genres?: string;
  filterPlayTimeMin?: number;
  filterPlayTimeMax?: number;
  complexity?: string;
  players?: string;
  sort?: 'title' | 'year';
  sortOrder?: 'asc' | 'desc';
};

export type ListReviewsParams = {
  slug: string;
  page: number;
  limit: number;
};
```

```
@Injectable()
export class GamesApplicationService {
  constructor(
    private readonly boardGamesRepository: PrismaBoardGamesRepository,
    private readonly gameReviewsRepository: PrismaGameReviewsRepository,
    private readonly gameRatingsRepository: PrismaGameRatingsRepository,
  ) {}

  async listGames(query: ListGamesParams) {
    const skip = (query.page - 1) * query.limit;
    const genreSlugs = parseGenreSlugs(query.genres);
    const complexityBands = parseComplexityBands(query.complexity);
    const playerCounts = parsePlayerCounts(query.players);
    const { items, total } =
      await this.boardGamesRepository.findPaginatedForList({
        titleSearch: query.titleSearch,
        genreSlugs: genreSlugs.length > 0 ? genreSlugs : undefined,
        filterPlayTimeMin: query.filterPlayTimeMin,
        filterPlayTimeMax: query.filterPlayTimeMax,
        complexityBands:
          complexityBands.length > 0 ? complexityBands : undefined,
        playerCounts: playerCounts.length > 0 ? playerCounts : undefined,
        sort: query.sort,
        sortOrder: query.sortOrder ?? 'asc',
        skip,
        take: query.limit,
      });

    return {
      data: items,
      meta: { total, page: query.page, limit: query.limit },
    };
  }

  async getGameBySlug(slug: string) {
    const row = await this.boardGamesRepository.findWithRatingStatsBySlug(slug);
    if (!row) {
      throw new NotFoundException('Game not found');
    }
    return row;
  }
```

```
createGame(props: CreateGameProps) {
  return this.boardGamesRepository.createGame(props);
}

async updateGame(slug: string, patch: UpdateGamePatch) {
  const game = await this.boardGamesRepository.updateGameBySlug(slug, patch);
  if (!game) {
    throw new NotFoundException('Game not found');
  }
  return game;
}

async deleteGame(slug: string): Promise<OkResponse> {
  const ok = await this.boardGamesRepository.deleteGameBySlug(slug);
  if (!ok) {
    throw new NotFoundException('Game not found');
  }
  return { ok: true };
}

async listReviews(params: ListReviewsParams) {
  const skip = (params.page - 1) * params.limit;
  const result = await this.gameReviewsRepository.findPaginatedByGameSlug({
    slug: params.slug,
    skip,
    take: params.limit,
  });
  if (!result) {
    throw new NotFoundException('Game not found');
  }

  return {
    data: result.items,
    meta: {
      total: result.total,
      page: params.page,
      limit: params.limit,
    },
  };
}
```

```
async createReview(
  slug: string,
  user: AuthUser,
  body: string,
  imageUrls: string[],
) {
  try {
    const row = await this.gameReviewsRepository.createReviewByGameSlug({
      slug,
      userId: user.id,
      body,
      imageUrls,
    });
    if (!row) {
      throw new NotFoundException('Game not found');
    }
    return row;
  } catch (e) {
    if (e instanceof DuplicateGameReviewError) {
      throw new ConflictException('You already reviewed this game');
    }
    throw e;
  }
}

async updateReview(
  slug: string,
  user: AuthUser,
  body: string,
  imageUrls: string[],
) {
  const row = await this.gameReviewsRepository.updateReviewByGameSlug({
    slug,
    userId: user.id,
    body,
    imageUrls,
  });
  if (!row) {
    throw new NotFoundException('Game or review not found');
  }
```

```
    return row;
  }

  async deleteReview(slug: string, user: AuthUser): Promise<OkResponse> {
    const ok = await this.gameReviewsRepository.deleteReviewByGameSlug({
      slug,
      userId: user.id,
    });
    if (!ok) {
      throw new NotFoundException('Game or review not found');
    }
    return { ok: true };
  }

  async upsertRating(slug: string, user: AuthUser, score: number) {
    const row = await this.gameRatingsRepository.upsertRatingByGameSlug({
      slug,
      userId: user.id,
      score,
    });
    if (!row) {
      throw new NotFoundException('Game not found');
    }
    return row;
  }

  async deleteRating(slug: string, user: AuthUser): Promise<OkResponse> {
    const result = await this.gameRatingsRepository.deleteRatingByGameSlug({
      slug,
      userId: user.id,
    });
    if (result === 'no_game') {
      throw new NotFoundException('Game not found');
    }
    if (result === 'not_found') {
      throw new NotFoundException('Rating not found');
    }
    return { ok: true };
  }
}
```

apps/api/src/collections/application/collections.application.service.ts

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { PrismaBoardGamesRepository } from '../../games/infrastructure/persistence/prisma-board-games.repository';
import type { CollectionEntry, CollectionStatus } from '@boardgame/shared';
import { PrismaUserGamesRepository } from '../infrastructure/persistence/prisma-user-games.repository';

@Injectable()
export class CollectionsApplicationService {
  constructor(
    private readonly boardGamesRepository: PrismaBoardGamesRepository,
    private readonly userGamesRepository: PrismaUserGamesRepository,
  ) {}

  listMyCollection(
    userId: string,
    status?: CollectionStatus,
  ): Promise<CollectionEntry[]> {
    return this.userGamesRepository.listForUser({
      userId,
      ...(status ? { status } : {}),
    });
  }

  async addToCollection(
    userId: string,
    slug: string,
    status: CollectionStatus = 'OWNED',
  ): Promise<CollectionEntry> {
    const gameId = await this.boardGamesRepository.findIdBySlug(slug);
    if (!gameId) {
      throw new NotFoundException('Game not found');
    }
    return this.userGamesRepository.upsertForUser({
      userId,
      gameId,
      data: { status },
    });
  }
```

```
async updateCollectionItem(
  userId: string,
  slug: string,
  patch: {
    status?: CollectionStatus;
    notes?: string | null;
    acquiredAt?: Date | null;
  },
): Promise<CollectionEntry> {
  const updated = await this.userGamesRepository.updateByUserAndGameSlug({
    userId,
    gameSlug: slug,
    patch: {
      ...(patch.status !== undefined && { status: patch.status }),
      ...(patch.notes !== undefined && { notes: patch.notes }),
      ...(patch.acquiredAt !== undefined && { acquiredAt: patch.acquiredAt }),
    },
  });
  if (!updated) {
    throw new NotFoundException('Collection entry not found');
  }
  return updated;
}

async removeFromCollection(userId: string, slug: string): Promise<void> {
  const ok = await this.userGamesRepository.removeByUserAndGameSlug(
    userId,
    slug,
  );
  if (!ok) {
    throw new NotFoundException('Collection entry not found');
  }
}

myEntryForGame(
  userId: string,
  slug: string,
): Promise<CollectionEntry | null> {
  return this.userGamesRepository.findEntryByUserAndGameSlug(userId, slug);
}
```

```
}
```

apps/web/src/main.tsx

```
import {
  MutationCache,
  QueryCache,
  QueryClient,
  QueryClientProvider,
} from '@tanstack/react-query';
import { RouterProvider } from '@tanstack/react-router';
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';
import { ToastProvider } from './components/ui/toast-provider';
import { AuthProvider } from './lib/auth-provider';
import { ApiError } from './lib/api';
import { toast } from './lib/toast';
import { syncThemeFromStorage } from './lib/theme';
import { router } from './router';
import './styles/global.scss';
import './index.css';

syncThemeFromStorage();

function toastClientError(error: unknown) {
  if (error instanceof ApiError) {
    return;
  }
  const message =
    error instanceof Error ? error.message : 'Something went wrong';
  toast.error(message);
}

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 30_000,
    },
  },
  queryCache: new QueryCache({
    onError: toastClientError,
  }),
```

```
  mutationCache: new MutationCache({
    onError: toastClientError,
  }),
});

createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <QueryClientProvider client={queryClient}>
      <AuthProvider>
        <RouterProvider router={router} />
        <ToastProvider />
      </AuthProvider>
    </QueryClientProvider>
  </StrictMode>,
);
```

apps/web/src/router.tsx

```
import {
  createRootRoute,
  createRoute,
  createRouter,
  redirect,
} from '@tanstack/react-router';
import { RootLayout } from './components/root-layout';
import { setPendingAuthModal } from './lib/auth-modal-intent';
import { getAccessToken, waitForAuthBootstrap } from './lib/auth-session';
import {
  gamesListSearchDefault,
  parseGamesListSearch,
} from './lib/games-route-defaults';
import { CollectionPage } from './pages/collection/collection-page';
import { parseCollectionSearch } from './lib/collection-route-defaults';
import { FriendsPage } from './pages/friends/friends-page';
import { fetchConversations } from './api/chat';
import { pickLatestConversation } from './lib/chat-labels';
import { MessagesChatPage } from './pages/messages/messages-chat-page';
import { GameDetailPage } from './pages/game-detail/game-detail-page';
import { GamesListPage } from './pages/games-list/games-list-page';
import { MeetupDetailPage } from './pages/meetup-detail/meetup-detail-page';
import { MeetupEditPage } from './pages/meetup-edit/meetup-edit-page';
```

```
import { MeetupInvitePage } from './pages/meetup-invite/meetup-invite-page';
import { MeetupNewPage } from './pages/meetup-new/meetup-new-page';
import { MeetupsListPage } from './pages/meetups-list/meetups-list-page';
import { ChangelogPage } from './pages/changelog/changelog-page';
import { HomePage } from './pages/home/home-page';
import { ProfilePage } from './pages/profile/profile-page';
import { PublicUserPage } from './pages/public-user/public-user-page';
import { SettingsPage } from './pages/settings/settings-page';

const FRIENDS_TABS = ['discover', 'friends', 'requests'] as const;
type FriendsTabRoute = (typeof FRIENDS_TABS)[number];

function parseFriendsTab(raw: unknown): FriendsTabRoute {
  return typeof raw === 'string' &&
    FRIENDS_TABS.includes(raw as FriendsTabRoute)
    ? (raw as FriendsTabRoute)
    : 'discover';
}

function parseMeetupsUpcoming(raw: unknown): 'true' | 'false' {
  return raw === 'false' ? 'false' : 'true';
}

function parseMeetupsView(raw: unknown): 'calendar' | 'list' {
  return raw === 'list' ? 'list' : 'calendar';
}

function parseMeetupsWeek(raw: unknown): string {
  if (typeof raw !== 'string' || !/^\d{4}-\d{2}-\d{2}$/.test(raw)) {
    return '';
  }
  return raw;
}

function parseMeetupsVisibility(
  raw: unknown,
): 'ALL' | 'PUBLIC' | 'FRIENDS' {
  if (raw === 'PUBLIC' || raw === 'FRIENDS') {
    return raw;
  }
  return 'ALL';
```

```
}

function parseMeetupsJoined(raw: unknown): '' | 'me' {
  return raw === 'me' ? 'me' : '';
}

async function requireAuthOrRedirect(): Promise<void> {
  await waitForAuthBootstrap();
  if (getAccessToken()) {
    return;
  }
  setPendingAuthModal('login');
  throw redirect({ to: '/games', search: gamesListSearchDefault });
}

const rootRoute = createRootRoute({
  component: RootLayout,
});

const indexRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/',
  beforeLoad: () => {
    throw redirect({ to: '/games', search: gamesListSearchDefault });
  },
});

const homeRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/home',
  component: HomePage,
});

const changelogRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/changelog',
  component: ChangelogPage,
});

const gamesListRoute = createRoute({
  getParentRoute: () => rootRoute,
```

```
  path: '/games',
  validateSearch: (raw: Record<string, unknown>) => parseGamesListSearch(raw),
  component: GamesListPage,
});

const gameDetailRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/games/$slug',
  component: GameDetailPage,
});

const collectionRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/collection',
  validateSearch: (raw: Record<string, unknown>) => parseCollectionSearch(raw),
  beforeLoad: () => requireAuthOrRedirect(),
  component: CollectionPage,
});

const profileRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/profile',
  beforeLoad: () => requireAuthOrRedirect(),
  component: ProfilePage,
});

const settingsRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/settings',
  beforeLoad: () => requireAuthOrRedirect(),
  component: SettingsPage,
});

const publicUserRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/u/$userId',
  component: PublicUserPage,
});

function parseMessagesListSearch(raw: unknown): boolean {
  return raw === '1' || raw === 1 || raw === true;
```

```
}

const messagesListRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/messages',
  validateSearch: (raw: Record<string, unknown>) => ({
    list: parseMessagesListSearch(raw.list),
  }),
  beforeLoad: async ({ search }) => {
    await requireAuthOrRedirect();
    if (search.list) {
      return;
    }
    const conversations = await fetchConversations();
    const latest = pickLatestConversation(conversations);
    if (latest) {
      throw redirect({
        to: '/messages/$conversationId',
        params: { conversationId: latest.id },
      });
    }
  },
  component: MessagesChatPage,
});

const messagesThreadRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/messages/$conversationId',
  beforeLoad: () => requireAuthOrRedirect(),
  component: MessagesChatPage,
});

const meetupsListRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/meetups',
  validateSearch: (raw: Record<string, unknown>) => ({
    page:
      typeof raw.page === 'string'
        ? Math.max(1, Number.parseInt(raw.page, 10) || 1)
        : typeof raw.page === 'number' && Number.isFinite(raw.page)
          ? Math.max(1, raw.page)
```

```
          : 1,
    upcoming: parseMeetupsUpcoming(raw.upcoming),
    view: parseMeetupsView(raw.view),
    week: parseMeetupsWeek(raw.week),
    gameId: typeof raw.gameId === 'string' ? raw.gameId : '',
    visibility: parseMeetupsVisibility(raw.visibility),
    q: typeof raw.q === 'string' ? raw.q.slice(0, 120) : '',
    joined: parseMeetupsJoined(raw.joined),
  }),
  beforeLoad: () => requireAuthOrRedirect(),
  component: MeetupsListPage,
});

const meetupsNewRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/meetups/new',
  beforeLoad: () => requireAuthOrRedirect(),
  component: MeetupNewPage,
});

const meetupsDetailRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/meetups/$meetupId',
  beforeLoad: () => requireAuthOrRedirect(),
  component: MeetupDetailPage,
});

const meetupsInviteRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/meetups/$meetupId/invite',
  beforeLoad: () => requireAuthOrRedirect(),
  component: MeetupInvitePage,
});

const meetupsEditRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/meetups/$meetupId/edit',
  beforeLoad: () => requireAuthOrRedirect(),
  component: MeetupEditPage,
});
```

```
const friendsRoute = createRoute({
  getParentRoute: () => rootRoute,
  path: '/friends',
  validateSearch: (raw: Record<string, unknown>) => ({
    tab: parseFriendsTab(raw.tab),
    q: typeof raw.q === 'string' ? raw.q : '',
    page:
      typeof raw.page === 'string'
        ? Math.max(1, Number.parseInt(raw.page, 10) || 1)
        : typeof raw.page === 'number' && Number.isFinite(raw.page)
          ? Math.max(1, raw.page)
          : 1,
  }),
  beforeLoad: () => requireAuthOrRedirect(),
  component: FriendsPage,
});

const routeTree = rootRoute.addChildren([
  indexRoute,
  homeRoute,
  changelogRoute,
  gamesListRoute,
  gameDetailRoute,
  collectionRoute,
  profileRoute,
  settingsRoute,
  publicUserRoute,
  friendsRoute,
  meetupsListRoute,
  meetupsNewRoute,
  meetupsDetailRoute,
  meetupsInviteRoute,
  meetupsEditRoute,
  messagesListRoute,
  messagesThreadRoute,
]);

export const router = createRouter({ routeTree });

declare module '@tanstack/react-router' {
  interface Register {
```

```
    router: typeof router;
  }
}
```

apps/web/src/lib/api.ts

```
import { getAccessToken, setAccessToken } from './auth-session';
import { toast } from './toast';

export const AUTH_LOGOUT_EVENT = 'boardgame:auth-logout';

function notifyAuthLogout(): void {
  window.dispatchEvent(new Event(AUTH_LOGOUT_EVENT));
}

export function getApiBaseUrl(): string {
  const url = import.meta.env.VITE_API_URL;
  if (!url) {
    throw new Error('VITE_API_URL is not set');
  }
  return url.replace(/\/$/, '');
}

export class ApiError extends Error {
  status: number;
  body?: unknown;

  constructor(message: string, status: number, body?: unknown) {
    super(message);
    this.name = 'ApiError';
    this.status = status;
    this.body = body;
  }
}

export async function apiFetch<T>(
  path: string,
  init?: RequestInit & { skipAuth?: boolean; withCredentials?: boolean },
): Promise<T> {
  const response = await doFetch(path, init);
  if (response.status !== 401 || init?.skipAuth) {
```

```
    return parseResponse<T>(response);
  }

  const refreshed = await tryRefreshAccessToken();
  if (!refreshed) {
    notifyAuthLogout();
    toast.error('Your session has expired. Please sign in again.');
    throw new ApiError('Unauthorized', 401);
  }

  const retryResponse = await doFetch(path, init);
  if (retryResponse.status === 401) {
    notifyAuthLogout();
    toast.error('Your session has expired. Please sign in again.');
  }
  return parseResponse<T>(retryResponse);
}

async function doFetch(
  path: string,
  init?: RequestInit & { skipAuth?: boolean; withCredentials?: boolean },
): Promise<Response> {
  const { skipAuth, withCredentials, ...rest } = init ?? {};
  const headers = new Headers(rest.headers);
  if (!skipAuth) {
    const token = getAccessToken();
    if (token) {
      headers.set('Authorization', `Bearer ${token}`);
    }
  }
  if (
    rest.body !== undefined &&
    typeof rest.body === 'string' &&
    !headers.has('Content-Type')
  ) {
    headers.set('Content-Type', 'application/json');
  }
  return fetch(`${getApiBaseUrl()}${path}`, {
    ...rest,
    headers,
    credentials: withCredentials ? 'include' : rest.credentials,
```

```
  });
}

async function parseResponse<T>(res: Response): Promise<T> {
  const text = await res.text();
  let data: unknown = undefined;
  if (text) {
    try {
      data = JSON.parse(text) as unknown;
    } catch {
      data = text;
    }
  }

  if (!res.ok) {
    const msg =
      typeof data === 'object' &&
      data !== null &&
      'message' in data &&
      typeof (data as { message: unknown }).message === 'string'
        ? (data as { message: string }).message
        : res.statusText;
    const error = new ApiError(msg || 'Request failed', res.status, data);
    notifyApiError(error);
    throw error;
  }
  return data as T;
}

function notifyApiError(error: ApiError): void {
  if (error.status === 401 || error.status === 404) {
    return;
  }
  toast.error(error.message);
}

let refreshPromise: Promise<boolean> | null = null;

async function tryRefreshAccessToken(): Promise<boolean> {
  if (refreshPromise) {
    return refreshPromise;
```

```
  }
  refreshPromise = (async () => {
    try {
      const response = await doFetch('/auth/refresh', {
        method: 'POST',
        skipAuth: true,
        withCredentials: true,
      });
      if (!response.ok) {
        setAccessToken(null);
        return false;
      }
      const data = await parseResponse<{ accessToken: string }>(response);
      setAccessToken(data.accessToken);
      return true;
    } catch {
      setAccessToken(null);
      return false;
    } finally {
      refreshPromise = null;
    }
  })();
  return refreshPromise;
}
```

apps/web/src/lib/auth-session.ts

```
let accessToken: string | null = null;
let authBootstrapPromise: Promise<void> | null = null;

export function getAccessToken(): string | null {
  return accessToken;
}

export function setAccessToken(nextToken: string | null): void {
  accessToken = nextToken;
}

export function setAuthBootstrapPromise(next: Promise<void>): void {
  authBootstrapPromise = next.finally(() => {
    authBootstrapPromise = null;
```

```
  });
}

export function waitForAuthBootstrap(): Promise<void> {
  return authBootstrapPromise ?? Promise.resolve();
}
```

apps/web/src/lib/auth-provider.tsx

```
import { useQueryClient } from '@tanstack/react-query';
import {
  useCallback,
  useEffect,
  useMemo,
  useState,
  type ReactNode,
} from 'react';
import { AUTH_LOGOUT_EVENT } from './api';
import { AuthContext } from './auth-context';
import { logoutRequest, refreshAccessToken } from '../api/auth';
import {
  getAccessToken,
  setAccessToken,
  setAuthBootstrapPromise,
} from './auth-session';
import { disconnectSharedChatSocket } from './chat-socket';
import { queryKeys } from './query-keys';
import type { AuthUser } from '../api/types';

let authBootstrapStarted = false;

function startAuthBootstrap(
  onToken: (token: string | null) => void,
): Promise<void> {
  const bootstrap = refreshAccessToken()
    .then((result) => {
      setAccessToken(result.accessToken);
      onToken(result.accessToken);
    })
    .catch(() => {
      setAccessToken(null);
```

```
      onToken(null);
    });
  setAuthBootstrapPromise(bootstrap);
  return bootstrap;
}

export function AuthProvider({ children }: { children: ReactNode }) {
  const queryClient = useQueryClient();
  const [token, setTokenState] = useState<string | null>(() =>
    getAccessToken(),
  );

  useEffect(() => {
    if (authBootstrapStarted) {
      return;
    }
    authBootstrapStarted = true;
    void startAuthBootstrap(setTokenState);
  }, []);

  const signIn = useCallback(
    (accessToken: string, user: AuthUser) => {
      setAccessToken(accessToken);
      setTokenState(accessToken);
      queryClient.setQueryData(queryKeys.auth.me, user);
    },
    [queryClient],
  );

  const signOut = useCallback(() => {
    void logoutRequest().catch(() => undefined);
    setAccessToken(null);
    disconnectSharedChatSocket();
    setTokenState(null);
    queryClient.removeQueries({ queryKey: queryKeys.auth.me });
  }, [queryClient]);

  useEffect(() => {
    const onLogout = () => {
      setAccessToken(null);
      disconnectSharedChatSocket();
```

```
      setTokenState(null);
      queryClient.removeQueries({ queryKey: queryKeys.auth.me });
    };
    window.addEventListener(AUTH_LOGOUT_EVENT, onLogout);
    return () => window.removeEventListener(AUTH_LOGOUT_EVENT, onLogout);
  }, [queryClient]);

  const value = useMemo(
    () => ({ token, signIn, signOut }),
    [token, signIn, signOut],
  );

  return <AuthContext.Provider value={value}>{children}</AuthContext.Provider>;
}
```

apps/web/src/lib/chat-socket.ts

```
import { io, type Socket } from 'socket.io-client';
import { getApiBaseUrl } from './api';

export function createChatSocket(accessToken: string): Socket {
  return io(`${getApiBaseUrl()}/chat`, {
    auth: { token: accessToken },
    transports: ['websocket', 'polling'],
    autoConnect: false,
  });
}

let sharedSocket: Socket | null = null;
let sharedToken: string | null = null;

export function getSharedChatSocket(accessToken: string): Socket {
  if (sharedSocket && sharedToken === accessToken) {
    if (!sharedSocket.connected) {
      sharedSocket.connect();
    }
    return sharedSocket;
  }

  if (sharedSocket) {
    sharedSocket.disconnect();
```

```
  }

  sharedSocket = createChatSocket(accessToken);
  sharedToken = accessToken;
  sharedSocket.connect();
  return sharedSocket;
}

export function disconnectSharedChatSocket(): void {
  if (sharedSocket) {
    sharedSocket.disconnect();
    sharedSocket = null;
    sharedToken = null;
  }
}
```

apps/web/src/api/meetups.ts

```
import type {
  CreateMeetupPayload,
  MeetupDetail,
  MeetupListItem,
  PaginatedMeta,
  PatchMeetupPayload,
} from '@boardgame/shared';
import { apiFetch } from '../lib/api';

export type {
  CreateMeetupPayload,
  MeetupDetail,
  MeetupListItem,
  PatchMeetupPayload,
  PlaySessionVisibility,
} from '@boardgame/shared';

export function fetchMeetups(params: {
  page?: number;
  limit?: number;
  upcoming?: 'true' | 'false';
  from?: string;
  to?: string;
```

```
  gameId?: string;
  visibility?: 'ALL' | 'PUBLIC' | 'FRIENDS';
  joined?: 'me';
  q?: string;
}): Promise<{ data: MeetupListItem[]; meta: PaginatedMeta }> {
  const sp = new URLSearchParams();
  if (params.page != null) {
    sp.set('page', String(params.page));
  }
  if (params.limit != null) {
    sp.set('limit', String(params.limit));
  }
  if (params.upcoming != null) {
    sp.set('upcoming', params.upcoming);
  }
  if (params.from != null && params.from.length > 0) {
    sp.set('from', params.from);
  }
  if (params.to != null && params.to.length > 0) {
    sp.set('to', params.to);
  }
  if (params.gameId != null && params.gameId.length > 0) {
    sp.set('gameId', params.gameId);
  }
  if (params.visibility != null && params.visibility !== 'ALL') {
    sp.set('visibility', params.visibility);
  }
  if (params.joined != null) {
    sp.set('joined', params.joined);
  }
  if (params.q != null && params.q.trim().length > 0) {
    sp.set('q', params.q.trim());
  }
  const qs = sp.toString();
  return apiFetch<{ data: MeetupListItem[]; meta: PaginatedMeta }>(
    `/meetups${qs ? `?${qs}` : ''}`,
  );
}

export function fetchMeetup(id: string): Promise<MeetupDetail> {
  return apiFetch<MeetupDetail>(`/meetups/${encodeURIComponent(id)}`);
```

```
}

export function createMeetup(body: CreateMeetupPayload): Promise<MeetupDetail>
{
  return apiFetch<MeetupDetail>('/meetups', {
    method: 'POST',
    body: JSON.stringify(body),
  });
}

export function patchMeetup(
  id: string,
  body: PatchMeetupPayload,
): Promise<MeetupDetail> {
  return apiFetch<MeetupDetail>(`/meetups/${encodeURIComponent(id)}`, {
    method: 'PATCH',
    body: JSON.stringify(body),
  });
}

export function cancelMeetup(id: string): Promise<MeetupDetail> {
  return apiFetch<MeetupDetail>(`/meetups/${encodeURIComponent(id)}/cancel`, {
    method: 'POST',
  });
}

export function joinMeetup(id: string): Promise<MeetupDetail> {
  return apiFetch<MeetupDetail>(`/meetups/${encodeURIComponent(id)}/join`, {
    method: 'POST',
  });
}

export function leaveMeetup(id: string): Promise<MeetupDetail> {
  return apiFetch<MeetupDetail>(`/meetups/${encodeURIComponent(id)}/leave`, {
    method: 'POST',
  });
}

export function createMeetupInvitation(
  meetupId: string,
  userId: string,
```

```
): Promise<{ ok: boolean }> {
  return apiFetch<{ ok: boolean }>(
    `/meetups/${encodeURIComponent(meetupId)}/invitations`,
    {
      method: 'POST',
      body: JSON.stringify({ userId }),
    },
  );
}

export async function sendMeetupInvitations(
  meetupId: string,
  userIds: string[],
): Promise<{ failed: number; total: number }> {
  if (userIds.length === 0) {
    return { failed: 0, total: 0 };
  }
  const results = await Promise.allSettled(
    userIds.map((userId) => createMeetupInvitation(meetupId, userId)),
  );
  const failed = results.filter((result) => result.status ===
'rejected').length;
  return { failed, total: userIds.length };
}
```