МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ
Циклова комісія (кафедра) комп’ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему

СЕРВЕРНА ЧАСТИНА ВЕБЗАСТОСУНКУ ДЛЯ КАРТКОВОЇ ГРИ

Виконав: студент групи 2П-21
Спеціальності
121 Інженерія програмного забезпечення
Ілля ТКАЧЕНКО
Керівник:
Антон МАЛИЙ

Черкаси 2025

АНОТАЦІЯ

Кваліфікаційна робота «Серверна частина вебзастосунку для карткової гри» присвячена створенню надійної, масштабованої та безпечної архітектури бекенду для багатокористувацької онлайн-гри. Основна увага приділена реалізації функціоналу реєстрації користувачів, авторизації, ігрової логіки, а також підтримці обміну повідомленнями в реальному часі. У межах роботи розглянуто вибір технологічного стеку, архітектурні рішення та алгоритми взаємодії між гравцями через REST API та WebSocket.

Метою роботи є створення серверної частини вебзастосунку, здатної забезпечити стійке функціонування онлайн-карткової гри з підтримкою сесій, обробкою дій гравців та збереженням ігрового стану в базі даних.

У процесі дослідження було обґрунтовано доцільність використання таких технологій, як NestJS для побудови серверної логіки, PostgreSQL як основної бази даних, Prisma ORM для доступу до даних, а також Docker для контейнеризації застосунку. Забезпечено підтримку масштабування та модульного розгортання за рахунок структурованої архітектури.

Актуальність теми зумовлена зростанням популярності онлайн-ігор серед молоді та потребою у розробці контрольованих, безпечних і стабільних середовищ для ігрової взаємодії.

Ключові слова: ВЕБЗАСТОСУНОК, БЕКЕНД, ОНЛАЙН-ГРА, КАРТКОВА ГРА, ПОКЕР, NESTJS, POSTGRESQL, PRISMA, REST API, WEBSOCKET, DOCKER.

ABSTRACT

The qualification work «Backend of a Web Application for a Card Game» focuses on the development of a secure, scalable, and robust backend architecture for a multiplayer online card game. The project centers around implementing core features such as user registration, authentication, game logic, and real-time player communication. This paper explores technology stack selection, architectural design, and interaction algorithms using REST API and WebSocket.

The purpose of this work is to build a backend system capable of supporting card game sessions, processing player actions, and persisting game states in a database.

The chosen technologies include NestJS for server-side logic, PostgreSQL as the primary database, Prisma ORM for data access, and Docker for application containerization. The architecture enables modular deployment and system scaling.

The relevance of the topic is driven by the increasing popularity of online games among young users and the need to create stable and secure platforms for interactive digital entertainment.

Keywords: WEB APPLICATION, BACKEND, ONLINE GAME, CARD GAME, POKER, NESTJS, POSTGRESQL, PRISMA, REST API, WEBSOCKET, DOCKER.

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ

HTTP – HyperText Transfer Protocol – протокол прикладного рівня для передачі гіпертекстових документів у мережі.

SQL – Structured Query Language – мова структурованих запитів до реляційних баз даних.

ACID – Atomicity, Consistency, Isolation, Durability – набір властивостей, що гарантує надійність транзакцій у базах даних.

CRUD – Create, Read, Update, Delete – базові операції над даними в інформаційних системах.

REST – Representational State Transfer – архітектурний стиль взаємодії компонентів у розподілених системах.

API – Application Programming Interface – інтерфейс програмування застосунків, що дозволяє взаємодію між компонентами ПЗ.

XSS – Cross-Site Scripting – тип вразливості вебзастосунків, що дозволяє виконувати зловмисний JavaScript-код.

ORM – Object-Relational Mapping – підхід до взаємодії з базами даних через об'єктно-орієнтоване програмування.

СУБД – Система управління базами даних – програмне забезпечення для створення, управління та використання баз даних.

СКБД – Система керування базами даних – синонім до СУБД.

JSON – JavaScript Object Notation – текстовий формат для зберігання і передачі структурованих даних.

JSONB – JSON Binary – бінарний формат зберігання JSON-даних у PostgreSQL з підтримкою індексації.

CI/CD – Continuous Integration / Continuous Deployment – безперервна інтеграція та розгортання змін у програмному забезпеченні.

VS Code – Visual Studio Code – популярне середовище розробки з відкритим вихідним кодом від Microsoft.

CSRF – Cross-Site Request Forgery – атака, що змушує користувача виконати небажану дію на вебзастосунку, де він автентифікований, без його відома.

TCP – Transmission Control Protocol – протокол транспортного рівня, що забезпечує надійну доставку даних між пристроями в мережі шляхом встановлення з’єднання та контролю порядку пакетів.

JWT – JSON Web Token – відкритий стандарт для безпечної передачі інформації між сторонами у вигляді компактного, самодостатнього токена у форматі JSON, який часто використовується для автентифікації та авторизації.

ЗМІСТ

ВСТУП

Упродовж останнього десятиліття спостерігається стрімке зростання популярності інтернет-ігор серед молоді віком 14-25 років. Понад 80 % підлітків у розвинених країнах щоденно грають в онлайн-ігри, від казуальних до масових багатокористувацьких. Це зумовлено доступністю технологій, соціальною взаємодією в ігровому середовищі та психологічними стимулами, зокрема, нагороди й досягнення.

На тлі зменшення традиційних шкідливих звичок серед молоді, таких як куріння та вживання алкоголю, зростає час, проведений у мережі. Проте поряд із позитивними аспектами, наприклад, розвитком реакції чи стратегічного мислення, поширюються ігрова залежність, соціальна ізоляція та зниження мотивації до офлайн-активностей.

У цьому контексті постає питання державної політики: заборони часто призводять до зворотного ефекту, тоді як створення безпечних, контрольованих ігор із урахуванням потреб молоді може бути більш ефективною стратегією. Приклад Китаю, зокрема політика щодо гри Genshin Impact, демонструє, як уряд може впливати на індустрію через цензуру, обмеження ігрового часу та контроль витрат, поєднуючи захист молоді з економічними інтересами.

Об’єктом дослідження виступають вебзастосунки для ігор, зокрема карткових, а саме архітектури бекенду, що забезпечують обробку запитів, зберігання даних і обмін інформацією між клієнтами в режимі реального часу.

Предметом дослідження є підходи, алгоритми та технології, що використовуються для розроблення бекенду, забезпечення безперервної роботи, збереження даних і обробки подій у багатокористувацьких вебзастосунках.

Метою цього проєкту є створення серверної частини вебзастосунку для карткової гри, що стане основою для реалізації безпечного, контрольованого онлайн-продукту. У межах розробки реалізовано базову функціональність: реєстрацію користувача, авторизацію, особистий кабінет, логіку ігрового процесу, з’єднання з клієнтською частиною через REST API та WebSocket, а також збереження ігрових даних у базі даних PostgreSQL.

У межах дослідження, присвяченого розробці серверної частини вебзастосунку для онлайн-карткової гри, поставлено такі практичні завдання:

1. Провести аналіз сучасних технологій створення серверної частини вебзастосунків, зокрема архітектурних підходів, інструментів розробки та комунікаційних протоколів. Обґрунтувати вибір відповідного технологічного стеку для реалізації онлайн-карткової гри на прикладі техаського холдему.

2. Розробити та реалізувати серверну частину вебзастосунку, яка забезпечує базову функціональність гри, включаючи:

— обробку підключень гравців у реальному часі;

— зберігання та обробку інформації про користувачів та ігри в базі даних;

— реалізацію основної логіки гри та правил покеру;

— систему реєстрації та авторизації користувачів.

3. Інтегрувати механізми обміну повідомленнями в режимі реального часу за допомогою WebSocket або аналогічних технологій для забезпечення багатокористувацької взаємодії.

4. Виконати модульне та інтеграційне тестування основних компонентів системи з метою перевірки їхньої працездатності, стабільності та відповідності функціональним вимогам.

РОЗДІЛ 1
АНАЛІЗ СУЧАСНОГО СТАНУ РИНКУ ТА ТЕХНОЛОГІЙ ДЛЯ СТВОРЕННЯ СЕРВЕРНОЇ ЧАСТИНИ ОНЛАЙН-ІГОР

1.1 Аналіз сучасних архітектур і технологій бекенд-розробки

У межах цього розділу будуть розглянуті архітектурні підходи до побудови серверної частини вебзастосунків, зокрема в контексті реалізації онлайн-ігор, таких як покер. Також аналізуватимуться сучасні технології контейнеризації, що відіграють ключову роль у забезпеченні стабільної та масштабованої роботи застосунку.

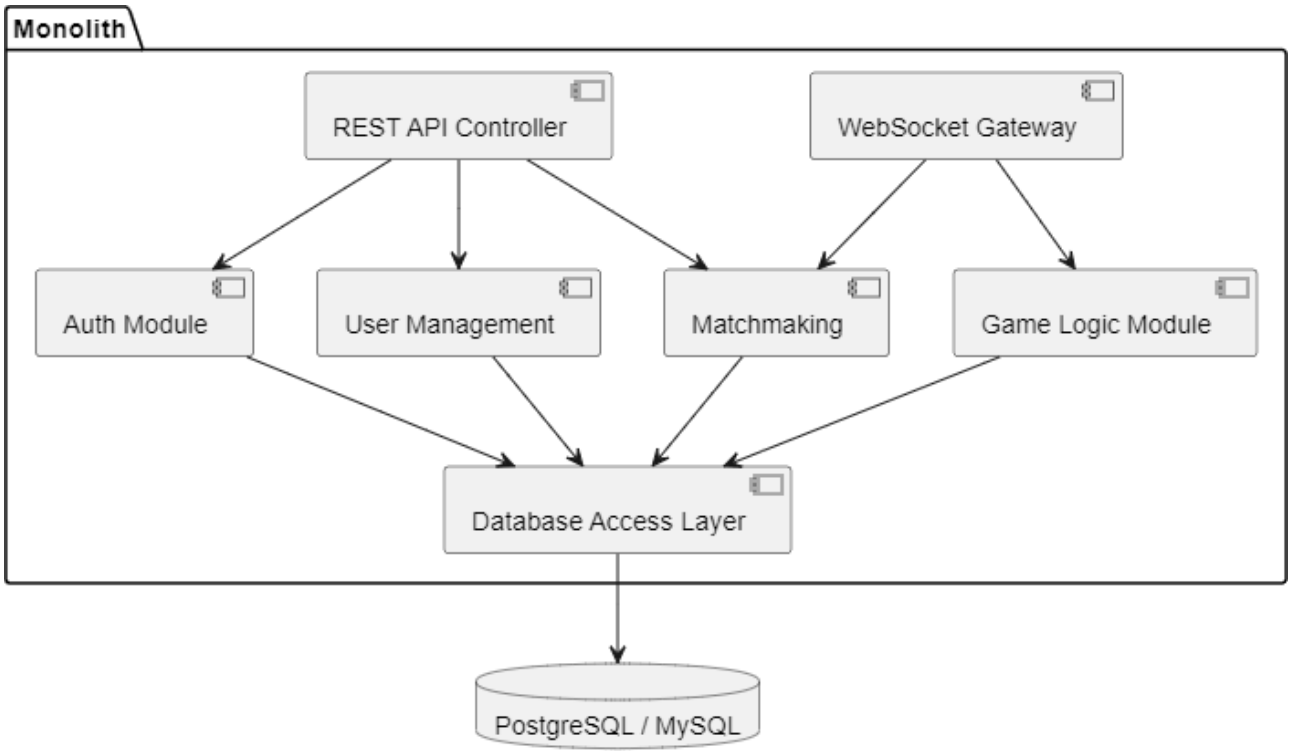

Рисунок 1.1 – Схема монолітного підходу до архітектури сайту

Монолітна архітектура передбачає, що всі компоненти застосунку реалізуються в межах єдиного коду, розгортаються одночасно й функціонують як єдине ціле. До переваг такого підходу можна віднести простоту розробки та

розгортання програмного рішення, а також швидкий його запуск. Проте варто враховувати й недоліки: обмежену масштабованість та ускладнення супроводу та управління великою кодовою базою. Звідси, для невеликого вебзастосунку на кшталт браузерної гри монолітна архітектура є доцільним вибором. Вона дає змогу швидко розпочати реалізацію проєкту та зосередитися на розробленні основної логіки. Проте з огляду на можливе зростання навантаження та кількості користувачів, у майбутньому може виникнути потреба в переході до більш гнучкого підходу. На рисунку 1.1 представлено загальну схему монолітної архітектури, яка ілюструє взаємодію між основними компонентами системи.

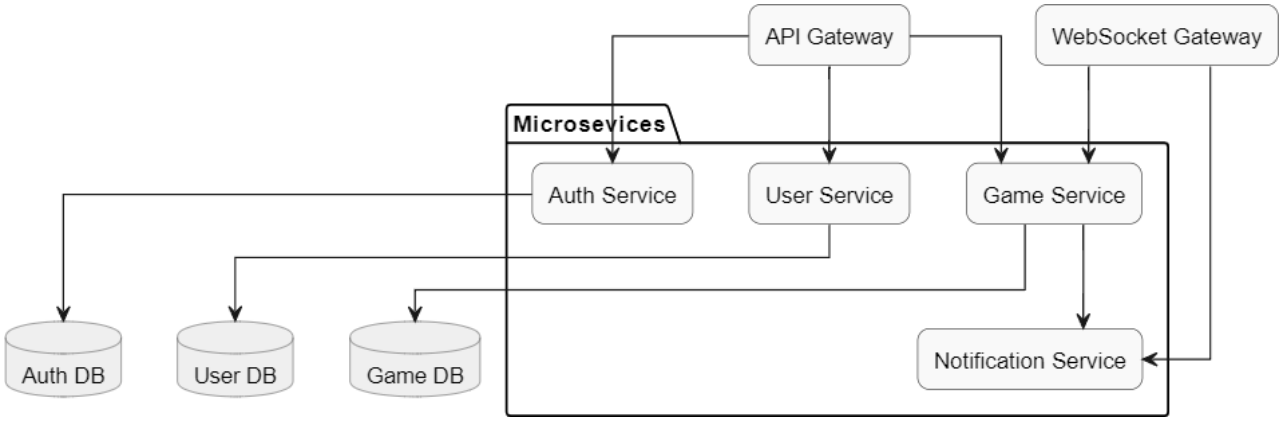

Рисунок 1.2 – Схема роботи мікросервісної архітектури

На рисунку 1.2 можна побачити схему взаємодії компонентів мікросервісної архітектури. Вона передбачає розподіл застосунку на низку незалежних сервісів, кожен з яких реалізує окрему бізнес-логіку та взаємодіє з іншими сервісами через чітко визначені інтерфейси. Такий підхід забезпечує високу модульність і дає змогу гнучко масштабувати окремі компоненти системи відповідно до навантаження. Проте мікросервіси також мають свої виклики – зокрема, потребують складнішої інфраструктури, ретельного налаштування міжсервісної взаємодії та ефективного управління розподіленими даними. З огляду на особливості проєкту, мікросервісна архітектура є доцільним вибором для систем, що очікують значного зростання навантаження, мають підвищені вимоги до доступності та потребують гнучкої підтримки окремих функціональних блоків. У випадку з браузерною грою, яка потенційно може

еволюціонувати у масштабний комерційний продукт, цей підхід може стати актуальним на етапі масштабування. Проте на початковій стадії розробки, враховуючи складність реалізації та додаткові витрати, впровадження мікросервісної архітектури є передчасним.

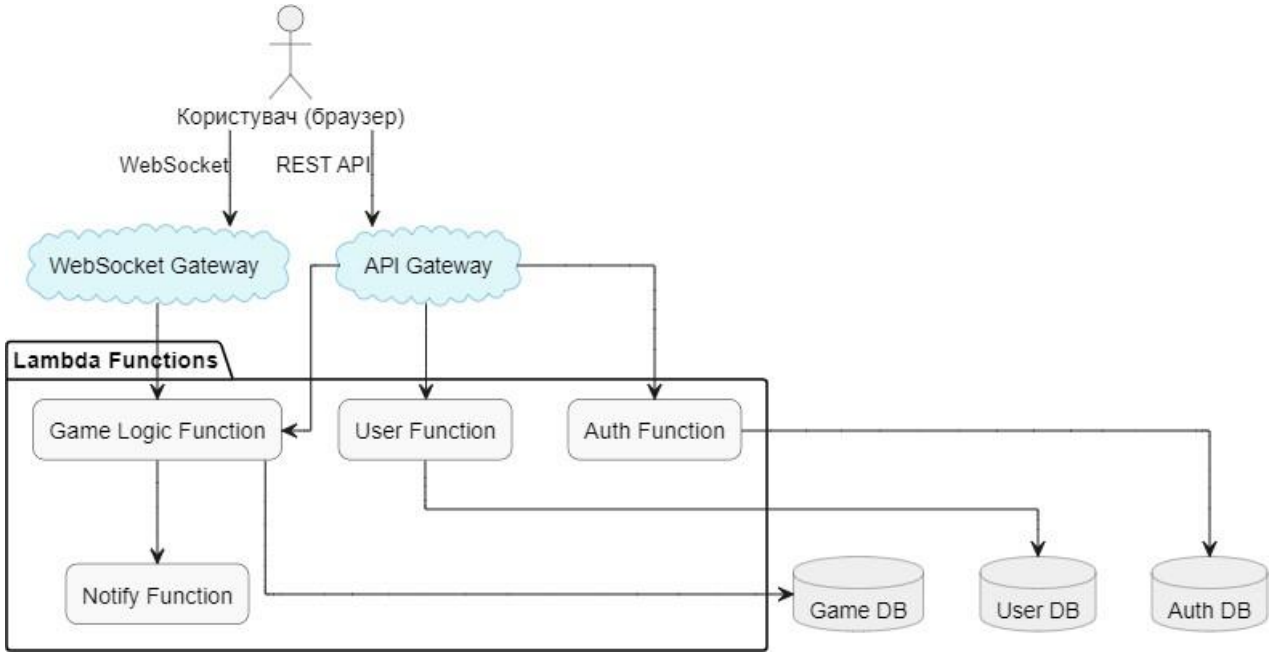

Рисунок 1.3 – Схема роботи serverless підходу

Serverless(безсерверна) архітектура ґрунтується на моделі виконання, за якої розробники можуть запускати окремі функції без необхідності управляти серверною інфраструктурою. Уся відповідальність за масштабування, моніторинг та обробку запитів покладається на хмарного провайдера. Це дозволяє значно спростити розгортання допоміжних компонентів застосунку, а також зменшити витрати завдяки оплаті лише за фактично виконані операції. Разом із тим варто враховувати певні обмеження – короткий час виконання функцій, складність збереження постійного стану та обмежена гнучкість для складних сценаріїв. З огляду на специфіку проєкту, serverless-підхід є доцільним рішенням для реалізації допоміжної логіки – наприклад, надсилання повідомлень, авторизації користувачів чи обробки результатів гри. Водночас основна ігрова логіка передбачає постійне збереження стану, оперативну взаємодію між гравцями та низьку затримку, що унеможливлює ефективне

використання повністю безсерверної архітектури для реалізації ядра гри. Візуалізацію ідеї безсерверної архітектури можна побачити на рисунку 1.2.

У разі впровадження serverless- або мікросервісного підходу важливого значення набуває контейнеризація – це сучасний підхід до розгортання програмного забезпечення, який передбачає упаковування застосунку разом з усіма його залежностями в ізольоване середовище – контейнер. Це дозволяє забезпечити стабільність і повторюваність виконання незалежно від середовища, у якому розгортається програма. Найбільш поширеним інструментом для цього є Docker [1], який забезпечує повторюваність середовища розробки, спрощує розгортання та полегшує управління масштабованими рішеннями.

Переваги контейнеризації досягаються завдяки низці технічних особливостей. Зокрема, портабельність забезпечується тим, що контейнер включає у себе все необхідне для роботи застосунку – операційне середовище, бібліотеки та конфігурації, – тож його можна запускати на будь-якій машині, де встановлено Docker. Швидке розгортання досягається завдяки легкості запуску контейнерів, що дозволяє миттєво масштабувати систему або переносити її між середовищами розробки, тестування та продакшну. Крім того, сумісність між різними середовищами реалізується за рахунок стандартизованої структури контейнерів, яка усуває залежність від особливостей конкретної операційної системи чи інфраструктури. Демонсрацію місця докера у розробці можна побачити на рисунок 1.4.

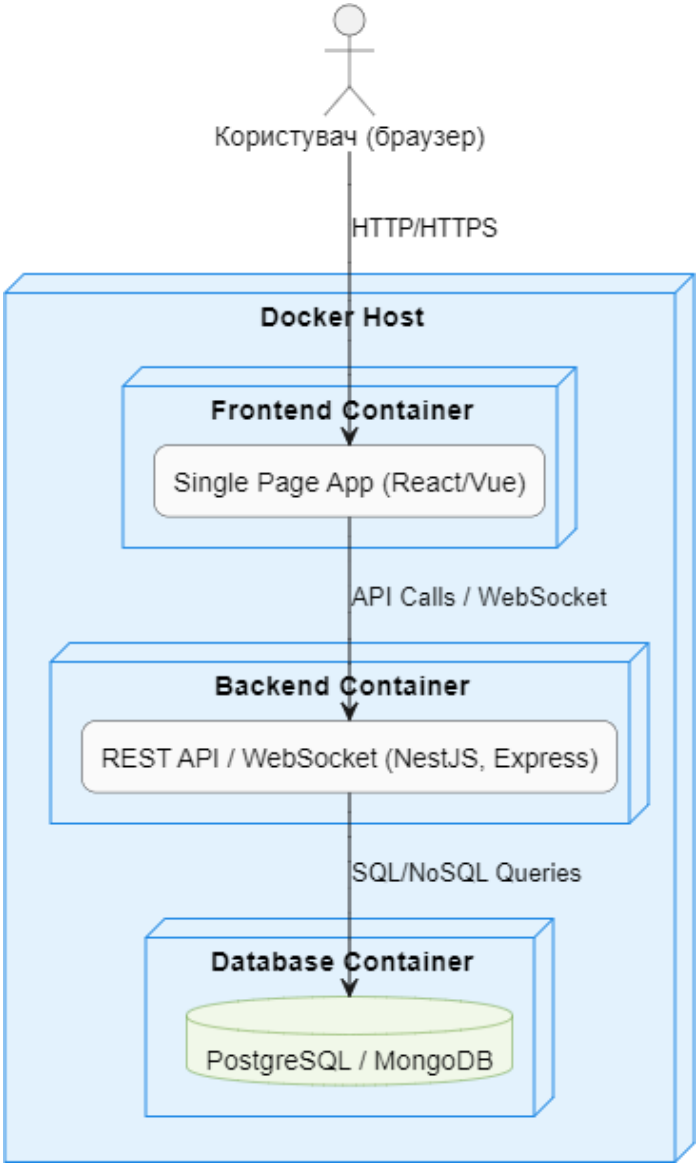

Рисунок 1.4 – Схема роль докера у розробці

У контексті створення вебзастосунку, зокрема браузерної гри, використання Docker є оптимальним рішенням уже на початковому етапі. Це значно спростить процес локальної розробки та прискорить розгортання. Крім того, контейнеризація закладає підґрунтя для майбутнього масштабування, оскільки дає змогу легко інтегрувати оркестраційні інструменти на кшталт Kubernetes [14] для автоматизації керування кількома сервісами.

З огляду на специфіку проєкту, доцільним архітектурним вибором на старті є монолітна структура, яка дозволяє швидко реалізувати основну ігрову логіку та зосередитися на функціональності. Проте завдяки використанню

контейнеризації застосунок залишатиметься гнучким і готовим до поступового переходу на мікросервісну архітектуру у разі зростання складності або навантаження.

1.2 Огляд технологій для розробки серверної частини

Таблиця 1.1 – Порівняння систем керування базами даних

Критерій	PostgreSQL	MySQL
Транзакції (ACID)	Повна підтримка ACID, гарантує цілісність даних навіть при паралельних запитах та збоях	Підтримує ACID лише в деяких рушіях (наприклад, InnoDB)
Складні SQL-запити	Підтримує підзапити, CTE (Common Table Expressions), віконні функції	Обмежена підтримка складних запитів
Можливості розширення	Плагіни, користувацькі типи, функції, індекси, JSONB	Менш розширювана структура
Реплікація	Нативна підтримка потокової реплікації	Може потребувати сторонні інструменти

Для створення серверної частини вебзастосунку карткової гри та проведено аналіз обраних рішень, наведено можливі альтернативи та обґрунтовано доцільність їхнього використання у межах даного проєкту.У якості основного інструменту для зберігання даних було визначено СКБД PostgreSQL. Це об’єктно-реляційна система керування базами даних з відкритим кодом, яка забезпечує підтримку складних SQL-запитів, транзакцій та розширень. PostgreSQL обрано через підтримку ACID-транзакцій, які гарантують збереження дій гравців навіть при одночасній взаємодії з системою, а також завдяки можливості ефективної реалізації складної логіки запитів для обробки стану ігрових сесій. Порівняння PostgreSQL з популярною альтернативою – MySQL було здійснено в таблиці 1.1.

Іншим програмним інструментом для досягнення поставленої мети може бути Prisma [2]. Це ORM-технологія для Node.js [11] і TypeScript [12], що генерує строго типізовані запити до бази даних на основі схеми. Альтернативним засобом в межах поточного етапу аналізу став TypeORM. Відповідне порівняння

характеристик вище згаданих технологій наведено в таблиці 1.2. Prisma забезпечує швидку інтеграцію з TypeScript, гарантує типову безпеку при роботі з даними та прискорює розробку завдяки зручному API.

Таблиця 1.2 – Порівняння ORM-технологій

Критерій	Prisma	TypeORM
Типізація	Автоматична генерація типів з бази даних у TypeScript	Часткова типізація, потребує додаткової конфігурації
Міграції	Вбудована підтримка схем і міграцій	Також підтримує міграції, але менш зручний синтаксис
Продуктивність	Генерує оптимізовані SQL-запити, мінімізує надлишкові запити	Деякі операції можуть бути менш ефективними
Простота у використанні	Мінімалістичний синтаксис, швидке навчання	Більше конфігурацій, складніший API

Важливу роль у вебзастосунках відіграє спосіб комунікації та обміну даними між програмними компонентами. Популярним рішенням у цьому контексті є REST API [3]. Це архітектурний стиль, що базується на HTTP-методах для взаємодії між клієнтом і сервером. REST API забезпечує швидку реалізацію серверної логіки та спрощує підтримку клієнтського інтерфейсу завдяки прямій відповідності між HTTP-методами та діями на сервері. Порівняльний аналіз було виконано для альтернативного рішення – GraphQL та систематизовано в таблиці 1.3.

Таблиця 1.3 – Порівняння способів комунікації фронтенду з бекндом

Критерій	REST API	Альтернатива (GraphQL)
Простота	Зрозумілий і стандартизований підхід, легкий у впровадженні	Потребує окремого парсера, складніша реалізація
Кешування	Легке використання HTTP-кешування	Кешування складніше реалізується
Гнучкість запитів	Стандартні CRUD-операції	Можна отримувати тільки потрібні поля
Документування	Широка підтримка OpenAPI/Swagger	Потрібна окрема схема, складніша документація

Потреба гри в реальному часі накладає додаткові вимоги до клієнт-серверного з’єднання. Типове рішення для такої задачі – WebSocket [4]. Це протокол, що забезпечує постійне двостороннє з’єднання між клієнтом і сервером, необхідне для обміну повідомленнями в реальному часі. Протокол WebSocket забезпечує своєчасну синхронізацію станів гри між усіма гравцями в режимі реального часу, що критично для карткової гри. Альтернативним підходом до забезпечення з’єднання в реальному часі може стати метод HTTP Long Polling. Порівняння характеристик цих технологій здійснено в таблиці 1.4.

Таблиця 1.4 – Порівняння засобів двостороннього зв’язку з сервером для реалізації ігор

Критерій	WebSocket	Альтернатива (HTTP Long Polling)
Затримка	Низька, майже миттєва передача повідомлень	Затримка між запитами, навантаження на сервер
Постійне з’єднання	Так	Потрібно повторне з’єднання
Обсяг трафіку	Мінімальний протокол, менше заголовків	Більший трафік через повторні HTTP-запити
Підтримка браузерами	Підтримується усіма сучасними браузерами	Також підтримується, але менш ефективно

Ще одним ключовим елементом технологічного стеку є Node.js / NestJS [5] – середовище виконання JavaScript і фреймворк для створення серверної частини застосунку. Node.js забезпечує неблокуючу асинхронну модель обробки запитів, що особливо важливо в умовах багатокористувацької взаємодії в режимі реального часу. Завдяки своїй подієвій природі Node.js дозволяє ефективно обробляти велику кількість одночасних підключень, не втрачаючи продуктивності. Фреймворк NestJS, побудований поверх Node.js і TypeScript, реалізує модульну архітектуру із чітким розділенням відповідальностей (контролери, сервіси, модулі), що значно підвищує структурованість і масштабованість коду. Крім того, NestJS має вбудовану підтримку WebSocket через пакет @nestjs/websockets, що дає змогу легко реалізовувати двосторонню комунікацію в реальному часі.

Вибір NestJS як серверного фреймворку в межах даного проєкту зумовлений кількома факторами: можливістю швидкої побудови як REST, так і WebSocket API; вбудованою інтеграцією з Prisma ORM; підтримкою типобезпеки завдяки TypeScript. Це дозволяє суттєво знизити складність розробки, прискорити впровадження функціоналу та забезпечити чисту архітектуру сервісу. Порівняльна оцінка NestJS та Express.js як альтернативного фреймворку наведена в таблиці 1.5.

Таблиця 1.5 – порівняння середовищ виконання коду для вебзастосунку

Критерій	Node.js + NestJS	Альтернатива (Python + Django / Flask)
Паралелізм	Асинхронна модель, неблокуючі операції	Підтримка асинхронності в Flask та Django обмежена
Єдина мова	Можливість використання TypeScript на фронтенді та бекенді	Потрібна інтеграція з окремою мовою для фронтенду
Структура проєкту	NestJS задає чітку структуру (контролери, сервіси, модулі)	Django – своя структура, Flask – мінімалістична
Підтримка WebSocket	Розширення через @nestjs/websockets	Потребує сторонніх бібліотек

У процесі розробки програмного забезпечення важливим компонентом інфраструктури є система контролю версій. У межах даного проєкту було обрано Git [6] – децентралізовану систему керування версіями, яка дозволяє ефективно відстежувати історію змін у кодовій базі, працювати з гілками та підтримувати командну розробку. Git забезпечує збереження всіх змін у репозиторії, дозволяє реалізовувати паралельну роботу над функціональністю, проводити злиття коду та управляти конфліктами. Завдяки широкій підтримці платформами GitHub, GitLab та Bitbucket, ця система інтегрується в сучасні CI/CD-процеси, спрощуючи автоматизацію тестування, деплою та перевірки якості коду. Серед переваг Git – підтримка складних стратегій злиття (merge, rebase), гнучкість у веденні історії змін і велика кількість навчальних матеріалів, що полегшує вхід у технологію як для новачків, так і для досвідчених розробників. У межах проєкту Git використовується як основа командної роботи, забезпечуючи надійне

відстеження змін, фіксацію стабільних релізів та ефективну організацію колективної розробки.

Альтернатива – Mercurial. Хоча ця система також належить до розподілених систем контролю версій, вона менш поширена та має обмежену підтримку сучасних інструментів і сервісів. Mercurial поступається Git за гнучкістю, має менше доступної документації та прикладів, що ускладнює її використання в рамках командної роботи та інтеграції з CI/CD. Порівняння Git і Mercurial наведено в таблиці 1.6.

Таблиця 1.6 – порівняння систем контролю версій

Критерій	Git	Альтернатива (Mercurial)
Популярність	Широко використовується, підтримується GitHub, GitLab, Bitbucket	Менш поширена, обмежена інтеграція
Гнучкість гілкування	Підтримка складних стратегій злиття, rebase	Менш розвинуті можливості
Навчання	Велика кількість навчальних матеріалів	Менше документації та прикладів

Як середовище для написання коду проєкту було обрано Visual Studio Code (VSCode) [7] – легкий, кросплатформний текстовий редактор з відкритим кодом, який забезпечує повноцінне середовище для розробки вебзастосунків. VSCode має вбудовану підтримку JavaScript та TypeScript, інтеграцію з Git, а також широкий спектр розширень для роботи з такими технологіями, як NestJS, Prisma та Docker. Його архітектура дозволяє налаштувати середовище під індивідуальні потреби розробника, зберігаючи при цьому високу швидкість роботи та низьке споживання ресурсів. Завдяки великій кількості безкоштовних розширень VSCode забезпечує комфортну підтримку автодоповнення коду, налагодження, перевірки типів, форматування та тестування. У проєкті VSCode використовується як основний інструмент розробки як для фронтенду, так і для серверної частини, забезпечуючи зручність переходу між модулями, швидкий запуск і легке налаштування середовища.

Альтернатива – WebStorm [13]. Це комерційне середовище розробки від

JetBrains із глибокою інтеграцією JavaScript фреймворків, яке зазвичай має вбудовану підтримку більшості функцій без необхідності встановлення додаткових плагінів. Водночас WebStorm може бути більш вимогливим до ресурсів і потребує платної ліцензії, що може бути обмеженням для невеликих команд або навчальних проєктів. Порівняння характеристик VSCode та WebStorm подано в таблиці 1.7.

Таблиця 1.7 – порівняння редакторів коду

Критерій	VSCode	Альтернатива (WebStorm)
Вартість	Безкоштовний	Комерційна ліцензія
Підтримка розширень	Велика кількість плагінів для NestJS, Prisma, Git	Вбудована підтримка більшості функцій
Швидкодія	Легкий, швидкий запуск	Може споживати більше ресурсів

Для реалізації серверної частини вебзастосунку гри в покер було обрано набір технологій, що дозволяє ефективно обробляти ігрову логіку, забезпечувати взаємодію між користувачами в реальному часі та зберігати дані без втрат. Для зберігання ігрової інформації використовується система управління базами даних PostgreSQL, що підтримує транзакції та складні SQL-запити. Доступ до бази реалізовано за допомогою Prisma ORM, яка забезпечує типізовані запити, спрощує роботу з даними та містить вбудовані механізми захисту від SQL-ін’єкцій.

Обмін інформацією між клієнтом і сервером здійснюється через REST API для стандартних запитів і WebSocket для постійного обміну подіями в реальному часі. Серверна логіка реалізована на основі Node.js у поєднанні з фреймворком NestJS, що забезпечує чітку модульну структуру застосунку. NestJS також має вбудовані засоби підвищення безпеки, зокрема захист від XSS-атак і CSRF, а також можливість централізованої обробки помилок та валідації вхідних даних.

Контроль версій та командна робота з кодом реалізовані за допомогою Git,

а розробка ведеться у середовищі Visual Studio Code, яке підтримує необхідні інструменти та розширення. Така технологічна база дає змогу реалізувати багатокористувацьку браузерну гру з можливістю масштабування, обробки дій гравців у режимі реального часу та з урахуванням базових вимог до безпеки застосунку.

1.3 Огляд програмних аналогів

У ході розробки серверної частини вебзастосунку для карткової гри доцільно проаналізувати вже наявні рішення, які реалізують схожу функціональність. Це дозволяє краще зрозуміти архітектурні підходи, використовувані технології та типові функціональні можливості таких систем. Як приклад карткової гри було обрано техаський холдем, одну з найпопулярніших варіацій покеру, яка широко представлена в онлайн-ігрових платформах.

PokerStars [8] є однією з найвідоміших онлайн-платформ для гри в покер. Серверна частина цієї системи розроблена з урахуванням масштабованості та високої надійності. З технічної точки зору, PokerStars використовує власний пропрієтарний ігровий рушій, який підтримує мільйони одночасних з’єднань. Комунікація між клієнтами та сервером реалізована за допомогою двостороннього з’єднання (websocket), що забезпечує миттєвий обмін подіями.

Рисунок 1.5 – зображення з сайту PokerStars

Можна виокремити такі особливості платформи: підтримка великої кількості одночасних гравців; розвинена система черг та турнірів; підтримка фінансових транзакцій і безпеки даних; потужна система валідації ігрових дій на сервері. До недоліків програмного рішення варто віднести: пропрієтарну систему, код якої недоступний, також вона унеможливлює гру в покер всередині браузеру; високу складність реалізації аналогічної функціональності з нуля.

Governor of Poker [9] – це серія ігор, що першочергово орієнтовані на поодиноких користувачів, однак у рамках окремих версій реалізовано і багатокористувацький функціонал у вебсередовищі.

Рисунок 1.6 – зображення з сайту Governor of Poker

Серверна частина таких рішень зазвичай базується на Node.js у поєднанні з бібліотекою socket.io, що дозволяє реалізувати двосторонній обмін подіями в режимі реального часу. Архітектура таких систем зазвичай є доволі простою, що робить їх зручними для невеликих проєктів або ігор з обмеженим колом гравців. Стан гри зберігається на стороні сервера, що дозволяє керувати логікою торгів та роздач без залежності від клієнта. Крім того, у грі реалізовано підтримку різних сценаріїв – від класичного техаського холдему до сюжетного режиму та багатокористувацьких матчів.

Попри ці переваги, платформа має низку обмежень. Зокрема, механізми ставок і торгів, вони у ній менш гнучкі, порівняно з професійними системами, що орієнтовані на реальні турніри чи грошові ігри. Крім того, спрощений ігровий процес позначається на обмеженій кастомізації правил та глибини механіки, що робить її менш придатною для створення масштабованої покерної платформи.

PokerTH [10] – це проєкт з відкритим програмним кодом, що реалізує базову логіку гри в техаський холдем. Серверна частина системи розроблена мовою C++ і передбачає використання TCP-з'єднання для комунікації з клієнтами. Особливістю проєкту є відкритість коду, що дозволяє за потреби адаптувати його під індивідуальні вимоги, додати нову функціональність або інтегрувати з іншими компонентами (наприклад, з базами даних через ORM на кшталт Prisma в окремих модифікованих версіях).

Рисунок 1.7 – зображення з клієнту PokerTH

Серед переваг слід відзначити наявність основної логіки гри: роздача карт, етапи торгів, визначення переможця. Це робить PokerTH зручним базовим прикладом для вивчення принципів серверної реалізації покеру. Водночас система має низку недоліків – зокрема, відсутність розвинених механізмів авторизації, логування та захисту від шахрайства. Крім того, обмежена підтримка гри через браузер знижує її придатність для створення

повноцінного онлайн-продукту.

1.4 Постановка задачі на розробку

На основі поставленої мети дослідження та проведеного аналізу існуючих архітектур, технологій і програмних аналогів, можна сформулювати основні задачі для реалізації в межах цієї дипломної роботи: розробити серверну частину вебзастосунку для карткової гри “Техаський холдем”, яка забезпечуватиме базову ігрову логіку, управління сесіями користувачів, зберігання даних та взаємодію з клієнтською частиною через програмний інтерфейс (API). Для досягнення цієї мети було поставлено такі основні задачі:

1. Проєктування архітектури серверної частини вебзастосунку. Необхідно визначити основні компоненти системи, принципи їхньої взаємодії, а також обрати відповідну архітектурну модель (наприклад, REST + WebSocket).
2. Вибір і впровадження сучасного стеку технологій. Сервер має бути розроблений з використанням актуальних інструментів бекенд-розробки. Передбачається використання Node.js як середовища виконання, Prisma ORM для взаємодії з базою даних PostgreSQL, а також WebSocket для забезпечення обміну повідомленнями в реальному часі між гравцями.
3. Проєктування та реалізація бази даних. Створити структуру бази даних, яка дозволяє зберігати інформацію про користувачів, ігрові сесії, поточний стан гри, історію роздач та інші сутності, важливі для функціонування гри.
4. Розробка модулів авторизації та автентифікації. Реалізувати механізми реєстрації, входу в систему, збереження сесій, захисту даних користувачів за допомогою JWT [15] або інших відповідних технологій.
5. Створення API та обробка ігрової логіки на сервері. Реалізувати REST API для клієнтської частини (створення ігор, підключення до столу, отримання стану гри) та WebSocket-інтерфейс для обміну подіями в

реальному часі (хід гравця, ставки, передача карт, визначення переможця тощо).

6. Забезпечення коректної взаємодії між користувачами в межах однієї гри. Сервер повинен синхронізувати дії всіх гравців, гарантувати правильний порядок ходів, обробляти помилки та забезпечувати стабільну роботу багатокористувацьких сесій.

7. Здійснення тестування реалізованої серверної частини. Передбачено проведення модульного та інтеграційного тестування для перевірки коректності логіки гри, функціонування API, безпеки та стабільності серверного застосунку.

Таким чином, результатом виконання поставлених задач має стати працездатна серверна частина вебзастосунку, яка здатна забезпечити базову ігрову функціональність техаського холдему в онлайн-режимі, а також виконувати обмін даними з фронтенд-частиною гри.

РОЗДІЛ 2
ПРОЄКТУВАННЯ І РЕАЛІЗАЦІЯ СЕРВЕРНОЇ ЧАСТИНИ САЙТУ КАРТКОВОЇ ГРИ

2.1 Визначення функціональних та нефункціональних вимог

Функціональні вимоги системи поділяються на чотири основні групи: вимоги до користувачів, вимоги до ігрового процесу, вимоги до обліку балансу та вимоги до сесій. Попередньо вимоги до користувачів було сформовано так:

1. Система повинна забезпечувати реєстрацію користувачів із електронної пошти та паролю;
2. Передбачено вхід до системи через електронну пошту та пароль. Паролі зберігаються в хешованому вигляді;
3. Інтерфейс гри доступний через веббраузер, мобільний застосунок не передбачено; вебзастосунок має бути сумісним із поширеними браузерами (Google Chrome, Mozilla Firefox тощо);
4. Користувач має доступ до особистого профілю, в якому може переглядати власні дані, історію ігор, статистику результатів та поточний ігровий баланс.

Перелік вимог до ігрового процесу містить такі позиції:

1. Гра має формат PvP (гравець проти гравця). Партії формуються між двома або більше реальними користувачами. Участь ботів або елементів штучного інтелекту не передбачена. Гравці самостійно обирають кімнати, в яких очікують на гру. Якщо в кімнаті двоє або більше гравців підтвердили готовність, розпочинається партія. Непідтверджені учасники можуть спостерігати за перебігом гри.
2. Серверна частина відповідає за логіку гри: роздачу карт, контроль черговості ходів, визначення переможця відповідно до встановлених правил. Інформація про стан гри (ходи, карти, рахунок) у реальному часі передається всім гравцям.

3. Після завершення партії визначається переможець. Відповідно до результату відбувається перерахунок ігрової валюти: виграш нараховується переможцю, а ставка списується з переможених.

Вимоги до обліку балансу сформульовано так:

1. У системі використовується внутрішня ігрова валюта, яка є еквівалентом реального доллару. Баланс кожного користувача зберігається на сервері.

2. Після кожної гри сервер автоматично оновлює баланс: списує ставку у програвшого та нараховує виграш переможцю. Усі транзакції з балансом зберігаються в історії для подальшого перегляду або аудиту.

3. Поповнення ігрового балансу за допомогою реальних коштів на момент реалізації не підтримується, проте архітектура системи дозволяє додати цю можливість у майбутньому.

Вимоги до сесій мають такий вигляд:

1. Кожна гра супроводжується окремою ігровою сесією, яка включає інформацію про гравців та стан партії. Кожна сесія має унікальний ідентифікатор.

2. Після завершення гри сесія закривається. Система фіксує технічну поразку як гравець: втратив зв’язок із сервером; не встиг зробити хід у відведений для нього час (30 секунд від початку його ходу). У майбутньому планується додати можливість повернення до сесії, якщо час на хід гравця ще не закінчився.

3. Один користувач може брати участь не більше ніж в одній активній ігровій сесії одночасно.

Нефункціональні вимоги охоплюють параметри продуктивності, безпеки, масштабованості, портативності (контейнеризації) та готовності до хмарного розгортання. У контексті продуктивності визначено такі критерії:

1. Серверна частина повинна забезпечувати мінімальну затримку в обробці дій користувачів. Очікуваний час реакції системи на ключові події (хід, передача карт, результат) має становити не більше 300 мілісекунд.

2. Система повинна підтримувати обробку десятків і навіть сотень одночасних користувачів без помітного зниження швидкості.

Забезпечення безпеки здійснюється з таких позицій:

1. Передбачено захист персональних даних: паролі користувачів зберігаються у хешованому вигляді.

2. Система повинна бути захищена від типових загроз вебдодатків, таких як SQL-ін’єкції, підробка міжсайтових запитів (CSRF).

3. Доступ до ігрових функцій надається лише автентифікованим користувачам. Адміністративна функціональність не передбачається.

Імплементація вимог до масштабованості передбачає наступне:

1. Архітектура повинна передбачати можливість горизонтального масштабування. У разі збільшення кількості активних гравців можна запускати додаткові серверні екземпляри без зупинки роботи системи.

2. Підвищення навантаження не повинно призводити до критичного зниження продуктивності (перевищення затримки відповіді від сервера в 300мс). Серверна частина повинна стабільно функціонувати навіть за зростання кількості одночасних ігрових сесій.

Вимоги до портативності (контейнеризації) полягають у тому, що всі компоненти серверної частини повиннні бути готові до упакування в контейнери (наприклад, Docker). Це дає змогу забезпечити переносимість та однакову роботу програмного забезпечення у різних середовищах – як у розробницькому, так і в виробничому.

2.2 Розроблення архітектури серверної частини

У поданні загальної архітектури (рис. 2.1) вся взаємодія починається в браузері – користувач здійснює реєстрацію та вхід через REST-запити, після чого керує власним профілем і обирає ігрову кімнату. Одночасно відкривається WebSocket-з’єднання, що забезпечує двосторонній потік повідомлень для ходів, ставок і змін стану гри. Така організація відокремлює разові дії (створення

облікового запису, редагування профілю, ініціацію кімнат) від безперервного обміну ігровими подіями.

Рисунок 2.1 – Діграма контексту моделі c4

Більш деталізована внутрішня структура сервера подана на рис. 2.2. Перший рівень обробки – REST API Gateway (NestJS) – приймає HTTP-запити для реєстрації, входу, перегляду та редагування профілю, а також для створення й приєднання до кімнат. Паралельно WebSocket Gateway виділено в окремий контейнер для обробки подій гри в реальному часі.

Авторизаційна логіка розділена між двома модулями: Users Module відповідає за створення та читання записів користувачів у базі даних, керує хешуванням паролів та повертає профільні дані, тоді як Auth Module приймає на вхід облікові дані чи JWT, перевіряє їх і видає нові токени. Після успішної автентифікації запити надходять у ігрову підсистему.

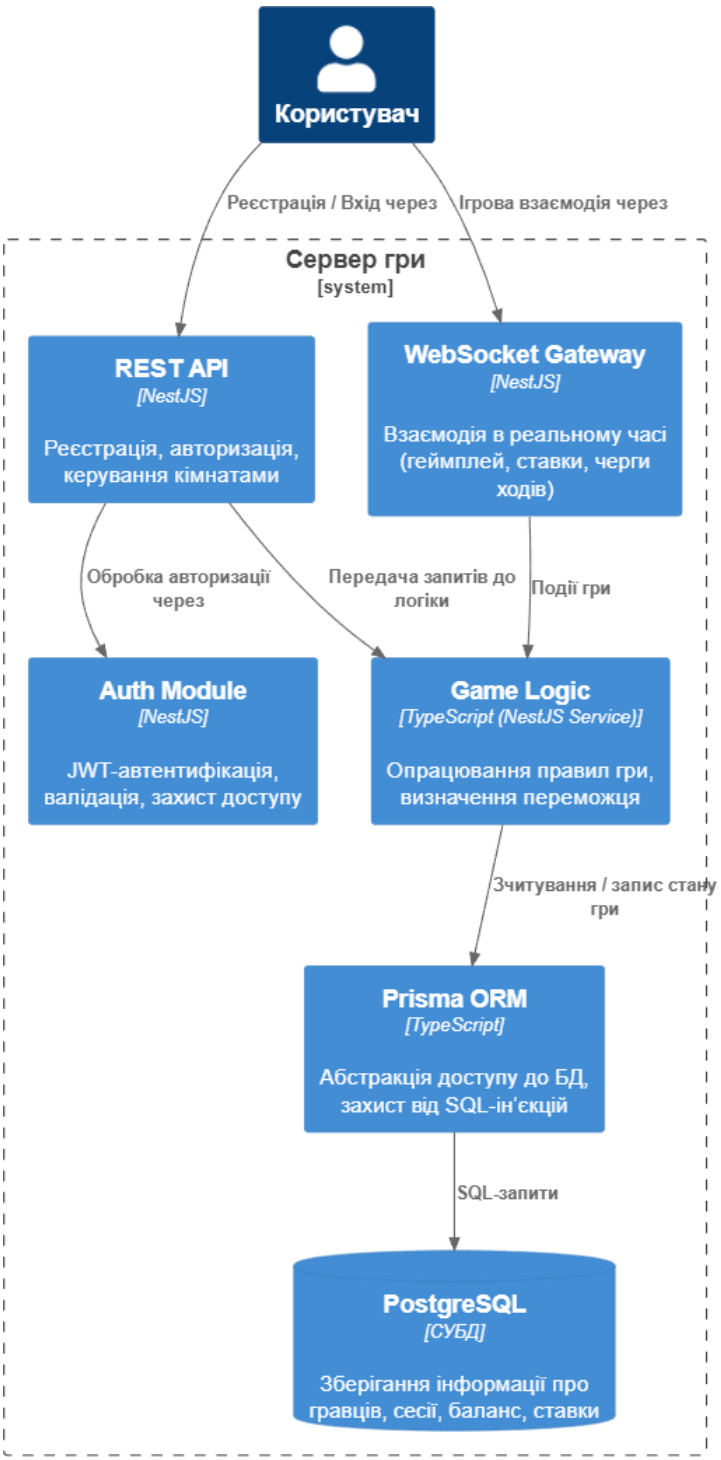

Рисунок 2.2 – Діаграма контейнерів моделі c4

Набір модулів, що реалізують ігрову логіку, включає: Rooms Module, який створює й налаштовує кімнати для очікування гравців; Player Module, що створює екземпляр гравця на основі профілю, коли користувач всередині кімнати сідає за стіл гри; Poker Module, який зберігає стан поточної сесії (столу), включаючи банк і список учасників; Step Module, що фіксує кожен окремий хід

(фолд, чек, колл, рейз, олл-ін) та пов’язані ставки.

Усі операції доступу до бази даних виконуються через Prisma ORM: вона формує типізовані запити, захищені від SQL-ін’єкцій, і перетворює результати у зручні об’єкти. Дані зберігаються в PostgreSQL, яке гарантує атомарність і цілісність транзакцій.

Поєднання цих компонентів забезпечує зрозумілу послідовність обробки: REST API приймає дозовані дії, WebSocket Gateway – інтерактивні події, Auth Module – валідацію доступу, Users Module – CRUD над користувачами, Rooms/Player/Poker/Step Modules – правила гри, ORM – абстракцію даних, а СУБД – збереження стану. Упакований у Docker-контейнер моноліт легко масштабується горизонтально, а за потреби окремі модулі можуть бути винесені до мікросервісів.

2.3 Проєктування бази даних PostgreSQL для зберігання інформації про користувачів та гри

З огляду на функціональні вимоги до системи, структура бази даних спроєктована як набір пов’язаних між собою таблиць, що чітко розділяють відповідальність за зберігання даних користувачів, ігрового процесу, дій гравців і взаємодії між ними. У якості рушія бази використано PostgreSQL – потужну реляційну СКБД, що забезпечує ACID-гарантії, високий рівень масштабованості та підтримку складних зв’язків. Логіка структури зберігається через використання зовнішніх ключів, обмежень цілісності та унікальних індексів.

Для кращого розуміння структури бази даних розроблено схему, яка представляє собою візуалізацію табличок бази даних, полів та зв’язків, які є між полями табличок.

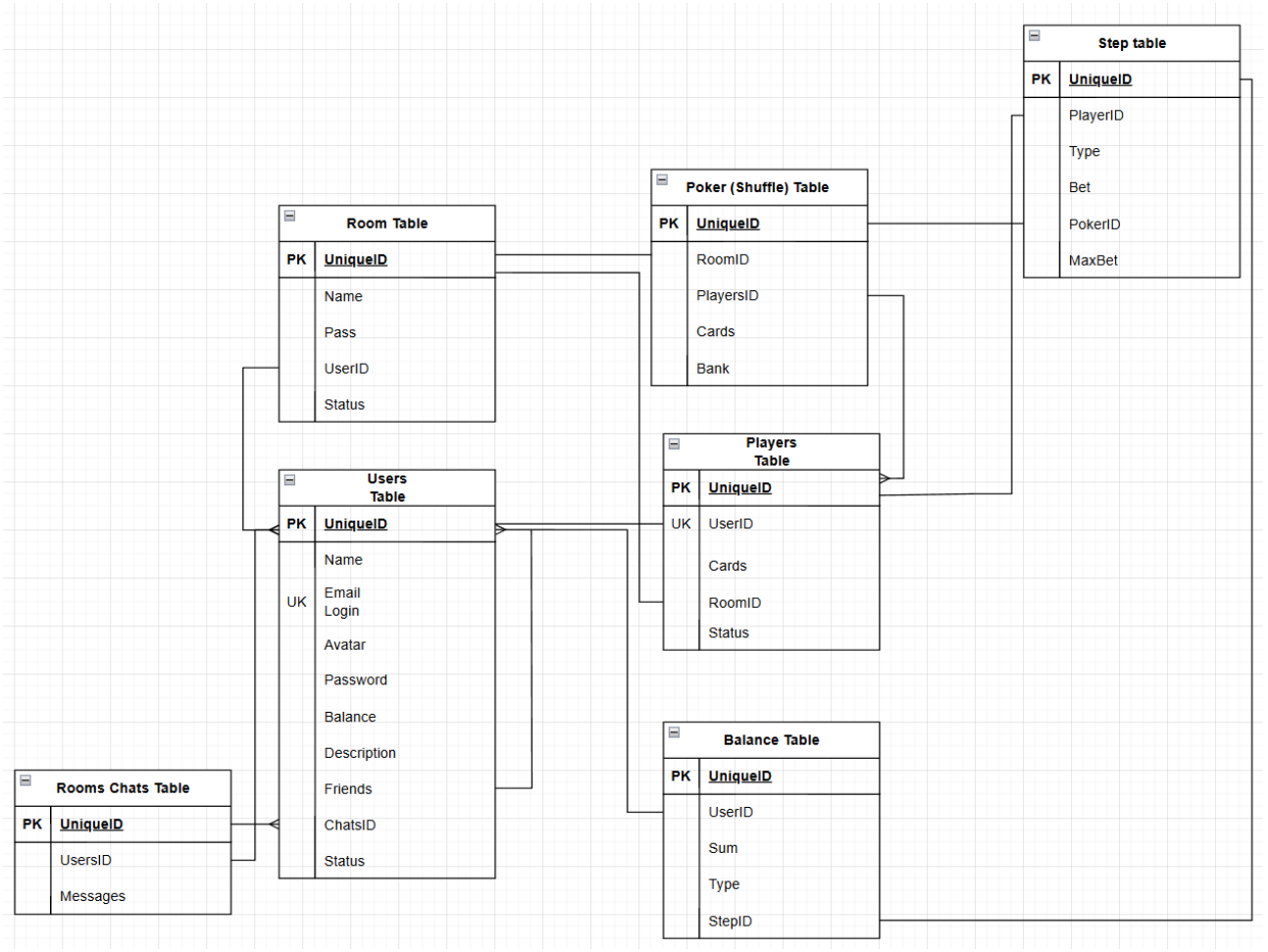

Рисунок 2.3 – Схематичне зображення бази даних

У центрі бази – таблиця users, яка виступає основною для зберігання облікових записів. Кожен запис містить email, пароль, нікнейм, аватар (якщо є), опис профілю, а також поточний статус користувача в системі. Додатково реалізовано масив друзів та список ідентифікаторів чатів. Для поля балансу використовується тип Decimal (10, 2), що дозволяє точно обраховувати внутрішню ігрову валюту без втрати точності. Через зовнішні ключі користувач може бути пов’язаний із записами в таблицях players (як гравець у грі) та balance (його фінансові дії).

Ігрова сесія відбувається всередині таблиці room, яка зберігає назву кімнати, статус (наприклад, “Waiting”, “Playing”, “Full”), а також список учасників, які до неї приєдналися. Для додаткової безпеки кімната може мати пароль. Один запис у room може бути пов’язаний із багатьма рядками у players, що представляють користувачів у ролі гравців. Кожен гравець у players має

посилання на користувача, набір карт (двохсимвольні значення, що зберігаються у вигляді масиву CHAR(2)), статус готовності та зв’язок з кімнатою. Ізоляція гравця від користувача дозволяє відслідковувати участь однієї особи у різних іграх без дублювання інформації.

Безпосередньо сама гра реалізована через таблицю poker, яка пов’язана з кімнатою через зовнішній ключ. Тут зберігається масив карт на столі, загальний банк і список гравців, що беруть участь у поточній партії. Одночасно з цим, для кожної гри ведеться таблиця step, яка логує послідовність дій кожного гравця: від першого ходу до фінального (фолд, чек, кол, рейз, оллін тощо). Для кожного ходу фіксуються тип дії, сума ставки (Decimal(10, 2)), гравець, якого вона стосується, та прив’язка до відповідної гри. Всі поля пов’язані через зовнішні ключі з таблицями poker та players.

Фінансова сторона гри реалізована через таблицю balance, яка логічно підв’язана до таблиці step, а також до users. Вона фіксує тип зміни балансу (внесок чи отримання), суму (bet) і пов’язану дію. Важливо, що всі оновлення балансу відбуваються у жорсткому зв’язку з конкретним кроком у грі, що забезпечує точне відтворення історії змін і прозорість кожної транзакції.

Для реалізації внутрішнього спілкування між користувачами використано таблицю chats. Вона містить список учасників (у вигляді масиву ідентифікаторів) і масив повідомлень, який зберігається у форматі JSON. Це рішення дозволяє швидко обробляти та оновлювати повідомлення у чатах, а також легко масштабувати їх у майбутньому.

Загалом, між усіма таблицями встановлені чіткі зв’язки через зовнішні ключі з обмеженням на каскадне видалення або обмежене оновлення, що забезпечує надійність структури при роботі з даними в умовах постійного навантаження. Така архітектура дозволяє не лише відтворювати логіку партії у покер, а й зберігати її історію, відслідковувати дії кожного гравця, вести фінансову звітність та підтримувати взаємодію між користувачами в режимі реального часу.

2.4 Реалізація системи реєстрації, авторизації та підтримки авторизованого стану

Ознайомитись із повним програмним кодом, що реалізує функціонал реєстрації, авторизації та підтримки авторизованого стану користувачів у системі, можна, перейшовши за посиланням на GitHub-репозиторій, наведеним у Додатку А. У цьому розділі представлено структурний опис основних компонентів, які відповідають за обробку облікових записів, валідацію даних, видачу токенів доступу та підтримку сесій користувачів.

Система реєстрації та авторизації користувачів у реалізованому вебзастосунку побудована за класичною JWT-моделлю з використанням модуля @nestjs/jwt. Центральну роль відіграють модулі auth та user, які відповідають за ідентифікацію користувача, створення облікового запису, валідацію облікових даних та підтримку активного стану користувача в системі.

У центрі функціоналу авторизації знаходиться AuthController, що обробляє запити на логін користувача. Він приймає електронну пошту та пароль як параметри запиту та викликає метод validateUser з AuthService. Якщо валідація не проходить, система повертає виняток UnauthorizedException, що запобігає несанкціонованому доступу. У разі успішної перевірки користувача, видається JWT-токен із корисним навантаженням (payload), у якому зберігається email, ідентифікатор користувача та його нікнейм.

Клас AuthService виконує перевірку користувача за email у базі даних, а також валідує відповідність наданого пароля за допомогою бібліотеки bcrypt. Метод Auth() у UserService відповідає за порівняння хешованого пароля з паролем, отриманим із запиту. Саме хешування відбувається при створенні нового облікового запису або оновленні пароля, що гарантує безпечне зберігання облікових даних.

Реєстрація реалізована через метод create() у UserController, який використовує DTO для валідації вхідних даних і створює нового користувача, застосовуючи хешування пароля. Кожен користувач зберігається в таблиці users

із вказанням email, нікнейма та хешованого пароля. Крім того, при створенні профілю користувача ініціалізується нульовий баланс та порожній аватар.

Після авторизації кожен запит користувача до захищених маршрутів, наприклад GET /user/profile, проходить через JwtAuthGuard, який декодує токен і передає інформацію про користувача до обробника запиту. Таким чином забезпечується перевірка автентичності на всіх етапах роботи з сервісом.

Особливістю реалізації є підтримка актуального онлайн-статусу користувача. Сервіс UserService використовує таймери для автоматичного перемикання статусу користувача на офлайн у разі відсутності активності протягом десяти хвилин. Метод updateUserOnlineStatus() оновлює відповідне поле в таблиці users, використовуючи службову карту Map, у якій зберігаються таймери для кожного активного користувача.

Сервіс UserService також реалізує допоміжні методи, зокрема пошук користувачів за email або нікнеймом, оновлення профілю (updateUser()), видалення акаунта з підтвердженням пароля (DeleteUser()), а також оновлення балансу (updateBalance()), який використовується при обліку внутрішньоігрових транзакцій.

Загалом, система автентифікації ідентифікує користувачів із гарантією безпеки, дотримуючись принципів OAuth-подібної моделі, та підтримує збереження авторизованого стану протягом сеансу користування, що критично важливо для інтерактивного середовища з реальною взаємодією, як у випадку багатокористувацької гри в покер.

2.5 Реалізація основних механік гри

Ознайомитись із повним вихідним кодом реалізації основних ігрових механік можна, перейшовши за посиланням на GitHub-репозиторій, наведеним у Додатку А. У цьому розділі описується структура WebSocket-шлюзу, який відповідає за обробку подій, пов’язаних зі створенням ігрового столу, підключенням гравців, початком гри, передачею карт, а також основними

етапами покерної партії: префлоп, флоп, терн і рівер.

Використано WebSocketGateway з простором імен /rooms, що дозволяє ефективно розмежовувати клієнтські підключення за кімнатами. Кожен гравець передає userId та roomId у заголовку запиту при з'єднанні. Ці параметри перевіряються, і за їх наявності користувача приєднують до відповідної кімнати. При першому вході в кімнату створюється ігровий стіл, який зберігається у базі даних, а його ідентифікатор асоціюється з відповідною кімнатою.

При отриманні події joinTable створюється новий запис про гравця, і його додають до відповідного ігрового столу. Після того як принаймні двоє гравців приєдналися до столу, запускається основний ігровий цикл – викликається метод handleGameStart.

Гра починається з формування повної колоди карт і її часткового розподілу: кожен гравець отримує по дві карти, а на стіл одразу викладається п'ять карт, які поступово відкриватимуться впродовж гри. Подальші дії обробляються відповідними методами: handlePreflop, handleFlop, handleTurn і handleRiver. Кожен із цих етапів включає коло ставок, яке реалізоване через методи betCircle та balancingCircle.

Метод betCircle ініціює початковий раунд ставок, де кожному активному гравцеві послідовно надсилається подія про початок його ходу. Гравець має 30 секунд для відповіді, після чого система або приймає ставку, або автоматично вважає, що гравець пасує. Залежно від контексту і ставки, система визначає тип дії: Call, Raise, Allin, Check, Fold або ReRaise. Після кожного кроку відбувається створення запису у таблиці step, а також надсилається подія stepDone усім гравцям у кімнаті.

Механізм balancingCircle використовується для вирівнювання ставок, коли після основного кола ще залишаються гравці з нерівними внесками. У цьому методі реалізована додаткова логіка перевірки коректності ставок та обмеження на надлишкове підвищення, з можливістю обробки некоректних дій з повідомленням про помилку (stepError).

Хоча логіка виграшу ще не реалізована повністю, структура класу вже

передбачає кроки для завершення гри, відкриття всіх карт і визначення переможця. Аналогічно, реалізація механіки Fold присутня, але потребує подальшої доробки для автоматичної елімінації пасивних гравців із активної фази гри.

Загалом, реалізований модуль забезпечує основний функціонал гри у покер у режимі реального часу, з обробкою ключових дій гравців, збереженням усіх змін у базі даних, та широким використанням WebSocket-комунікації для взаємодії між сервером і клієнтами.

РОЗДІЛ 3
МОДУЛЬНЕ ТА ІНТЕГРАЦІЙНЕ ТЕСТУВАННЯ ОСНОВНИХ ФУНКЦІЙ

3.1 Загальна стратегія тестування

Тестування є критичним етапом забезпечення якості та стабільності вебзастосунку. У випадку з багатокомпонентною системою на кшталт браузерної карткової гри доцільно розглядати тестування на кількох рівнях. Передусім це модульне тестування окремих частин логіки, зокрема AuthService, GameService чи UserService, що дозволяє перевірити кожен компонент в ізоляції. Далі інтеграційне тестування дає змогу впевнитися у коректній взаємодії між сервісами – як через REST API, так і через WebSocket чи ORM. Нарешті, інфраструктурне тестування дозволяє оцінити поведінку системи в умовах, наближених до реального розгортання: у контейнерах, з балансувальником та під навантаженням. У процесі роботи було використано Postman для ручного тестування REST API, однак існує ширший набір інструментів, які можуть бути застосовані для побудови повноцінної стратегії тестування, зокрема Jest, Supertest, Socket.io-client, Prisma test utils, Docker Compose, Artillery, K6, OWASP ZAP та інші.

Зокрема, за допомогою Postman буде протестовано основні REST API-ендпоінти, пов’язані з аутентифікацією, створенням та приєднанням до ігрових кімнат, а також оновленням інформації про користувача. Перевірка включає тестування коректних відповідей сервера на валідні запити, обробку помилок у випадку неправильних або неповних даних, а також дотримання стандартів безпеки — наприклад, наявність JWT-токена в заголовках запиту для захищених маршрутів. Крім того, у Postman буде створено колекцію запитів, яка дозволяє автоматизувати перевірку всього життєвого циклу користувача в системі: від реєстрації до виходу з гри.

3.2 Модульне тестування

Модульне тестування передбачає ізольовану перевірку логіки кожного сервісу окремо. У межах цього рівня слід провести наступні тести:

1. AuthService має бути протестований на відповідність логіці автентифікації та генерації токенів доступу. У межах модульного тестування цей сервіс підлягав перевірці за наступними напрямками: успішна авторизація за коректними обліковими даними; реакція на запит із неіснуючою поштою; обробка запиту з неправильним паролем; формування коректного JWT-токена з очікуваним набором даних (ID, email, nickname); перевірка строки життя access-токена.

Метою тестування було гарантувати, що система коректно ідентифікує користувача, обробляє типові помилки авторизації та формує JWT-токени згідно з очікуваними параметрами. Усі ці сценарії були детально перевірені вручну за допомогою Postman. Запити надсилалися до ендпоінту /auth/login з різними наборами вхідних даних для перевірки як позитивних, так і негативних гілок логіки.

За результатами тестування жодних помилок або відхилень від очікуваної поведінки не виявлено. AuthService продемонстрував стабільну роботу, правильну обробку помилок та надійне формування токенів відповідно до вимог системи безпеки.

2. UserService виконує критично важливі функції, пов’язані з обробкою профілів користувачів, тому тестування цього сервісу охоплювало широкий спектр перевірок. Основними аспектами тестування стали створення користувача, пошук користувача за ID, електронною адресою або нікнеймом, авторизація на основі пароля, оновлення особистих даних і видалення профілю.

Під час тестування вручну, через інструмент Postman, були змодельовані як валідні, так і невалідні сценарії. Серед них: створення нового облікового запису з коректними даними, спроба створити користувача з уже зареєстрованою електронною адресою, отримання профілю за правильним і неправильним ID,

оновлення облікових даних із валідним email, спроба оновлення профілю без передачі email, авторизація з неправильним паролем, а також видалення користувача з вірним і хибним паролем. Окрему увагу було приділено перевірці механізму автоматичного оновлення онлайн-статусу користувача, який спрацьовує при запиті /user/profile та перевіряє відсутність помилок у таймері відкладеної деактивації.

Крім основних функцій, також протестовано поведінку сервісу при обробці оновлення балансу користувача та коректність гешування паролів. Тестування охопило всі методи, реалізовані у UserService, включно з валідацією помилкових входів у базу даних через Prisma.

За підсумками тестування, жодних критичних помилок або непередбаченої поведінки виявлено не було. Усі методи сервісу працюють згідно з очікуваними результатами. Таким чином, UserService вважається повністю протестованим вручну, і його логіка відповідає вимогам до обробки користувацьких даних.

3. SessionService забезпечує створення і керування ігровими кімнатами (сесіями), до яких можуть приєднуватися користувачі, щоб брати участь у грі. Основні функції сервісу включають створення нових кімнат, приєднання користувачів до вже існуючих сесій, оновлення переліку учасників, видалення кімнат, а також координацію з WebSocket-з'єднаннями для оновлення інформації в реальному часі.

Стратегія тестування для цього сервісу охоплювала як HTTP-інтерфейси, так і події, що передаються через WebSocket. Основними тестовими сценаріями стали:

— Створення кімнати (POST /rooms/create) з валідними даними, включаючи параметри з і без пароля.

— Приєднання до кімнати (через WebSocket), включаючи: оновлення списку учасників у базі; надсилання відповідного повідомлення іншим клієнтам у кімнаті (userJoined); ігнорування дублікатів користувачів у списку usersid.

— Перевірка обмеження максимальної кількості гравців у кімнаті (максимум 6).

— Ініціація старту гри після приєднання другого гравця (через затримку на 5 секунд і подію prepare).

— Перевірка відправки повідомлень при відключенні користувача (Client_disconnected) та правильне оновлення списків usersid і RoomPlayersMap.

— Відлов помилкових запитів, таких як спроба приєднатися до неіснуючої кімнати або в кімнату.

— Обробка випадків, коли у гравця недостатньо балансу (вивід повідомлення tableConnectionError).

Всі вищезгадані сценарії були протестовані вручну за допомогою Postman для HTTP-запитів і інструментів WebSocket (таких як браузерні клієнти і плагіни) для перевірки роботи сокетів.

Крім того, особливу увагу було приділено стану RoomPlayersMap, який зберігає перелік активних гравців у кожній кімнаті, і взаємодії цього списку зі службами UserService та PlayerService. Також перевірено реакцію сервісу на розриви з'єднання та повторні з'єднання користувачів.

За підсумками ручного тестування багів або некоректної поведінки виявлено не було. Усі ключові функції SessionService працюють стабільно й відповідають очікуваному функціоналу. Сервіс готовий до інтеграції з іншими модулями системи.

4. GameService відповідає за основну ігрову логіку, включаючи старт гри, роздачу карт, організацію ставок, визначення черговості дій гравців, обробку усіх етапів гри (preflop, flop, turn, river), а також фінальне порівняння комбінацій та передачу результатів (шоудаун).

Стратегія тестування

Модуль GameService був протестований вручну через WebSocket-з'єднання, з використанням добровольців-тестерів. Тестування охоплювало повний життєвий цикл ігрової сесії:

— Старт гри (handleGameStart) – перевірка генерації унікальної колоди карт, роздачі по дві карти кожному гравцеві, збереження комбінацій у базу та

правильного повідомлення всім учасникам (gameStarted, yourCards).

— Фази гри – Flop, Turn, River: перевірка правильного відкриття карт, переходу між фазами, оновлення статусу гри (RoomGamestatusMap) та завершення кожної фази (FlopEND, TurnEND, RiverEND).

— Цикли ставок (betCircle, balancingCircle): перевірка мінімальних та максимальних допустимих ставок для кожного гравця; обробка подій makeYourStep, myStep, onMyStep – валідація ставок, визначення типу кроку (Call, Raise, Fold, Check, Allin); автоматичне завершення ходу при таймауті (30 секунд); обробка відмови від гри та списання ставок з банку.

Передача результатів (handleShowdown) – перевірка правильного завершення гри, формування переможця, виплати виграшу.

У процесі ручного тестування було виявлено кілька суттєвих помилок, зокрема:

— Неправильне визначення типу ставки (StepTypeEnum) – у певних ситуаціях, коли гравець робив Check, система помилково фіксувала хід як Call. Аналогічно, коли гравець йшов All-in, система визначала його дію як Raise або Re-raise, навіть якщо він просто виставляв увесь залишок фішок без перевищення попередньої ставки. Це призводило до некоректного трактування ігрової динаміки та подальшої обробки ставок.

Проблема була пов’язана з неточною логікою у функції stepTypeDefine, де не враховувався контекст (розмір стеку гравця, порівняння з поточною максимальною ставкою тощо). Баг було оперативно виправлено через уточнення умов розрізнення між Check, Call, Raise та All-in.

— Баги у визначенні сили комбінацій карт – на етапі шоудауну система неправильно порівнювала однакові за типом комбінації, зокрема Full House. У ситуації, коли один гравець мав трійку десяток і пару двійок, а інший – трійку четвірок і пару тузів, переможцем визначався другий гравець через наявність тузів, хоча за правилами покеру виграє комбінація з вищою трійкою, тобто десятки. При аналізі цієї проблеми стало очевидно, що аналогічна помилка

виникала також у комбінаціях Flush і Two Pair, де система некоректно оцінювала старші карти або другорядні пари. Зокрема, Flush не завжди визначався за найвищою картою масті, а Two Pair іноді віддавала перевагу меншій старшій парі через неправильний порядок порівняння.

Оцінку ваги у цих комбінаціях було повністю переглянуто: реалізовано коректне сортування карт за рангами перед порівнянням та оновлено логіку визначення переможця з урахуванням правил покеру. Після цього система успішно проходить всі перевірки та не виявляє подібних помилок.

5. BalanceService було протестовано вручну за допомогою Postman. Тестування охоплювало ключові сценарії, пов’язані з оновленням та відстеженням змін балансу користувача.

Перевірено наступні аспекти:

— Оновлення балансу: сервіс успішно виконує зміну балансу після ігрових подій (наприклад, списання ставки або нарахування виграшу). Усі значення коректно оновлюються в базі даних PostgreSQL, без округлень або втрати точності.

— Перевірка граничних значень: було протестовано сценарії з мінімальними (0.00) та великими сумами, при намаганні зробити баланс меншим за 0 повертається помилка, що сповіщає про неможливість такої операції.

— Журнал змін: сервіс успішно повертає історію змін балансу конкретного користувача, включно з типом транзакції (списання, поповнення, виграш, тощо), що дозволяє чітко простежити всі фінансові операції.

Результат: жодних багів або неточностей в логіці нарахування чи валідації балансу не виявлено. BalanceService виконує всі свої задачі згідно з очікуваннями, і готовий до інтеграції з іншими модулями застосунку.

Модульне тестування є фундаментом перевірки якості кожного окремого компонента системи. Воно дає змогу виявити помилки в бізнес-логіці ще до інтеграції сервісів між собою, що значно спрощує відлагодження та прискорює розробку. Тестування AuthService, UserService, SessionService, GameService та BalanceService в ізоляції забезпечує впевненість у правильності їхньої роботи та

готовність до подальшої інтеграції в загальну архітектуру застосунку.

3.3 Інтеграційне тестування

Інтеграційне тестування дозволяє перевірити коректність взаємодії між окремими сервісами системи, які пройшли попереднє модульне тестування. Мета цього рівня – переконатися, що окремі компоненти (AuthService, UserService, SessionService, GameService, BalanceService) працюють узгоджено, без логічних конфліктів, помилок синхронізації або проблем з обробкою спільних даних.

Тестування проводилося вручну, з використанням Postman для REST-запитів та браузерних WebSocket-клієнтів (включно з інструментами розробника та розширеннями) для перевірки реального часу.

1. АЗв’язок між AuthService, UserService та SessionService через REST API було перевірено в рамках повного циклу автентифікації та керування сесіями. У процесі тестування змодельовано типовий сценарій взаємодії користувача із застосунком: реєстрація нового акаунта здійснювалась через запит на /auth/register, при цьому автоматично створювався профіль у UserService. Подальша авторизація реалізовувалась через /auth/login, що дозволяло отримати access- і refresh-токени, а також перевірити механізм оновлення токена через /auth/refresh. Доступ до захищених маршрутів, таких як

/user/profile чи /rooms/create, успішно перевірявся за допомогою access-токена, що підтверджувало коректну роботу авторизаційних guard'ів. Окрему увагу було приділено обробці помилкових ситуацій – система передбачувано реагувала на запити без токена, з недійсним або протермінованим токеном, а також на спроби доступу до неіснуючих ресурсів чи повторну реєстрацію з уже використаною поштою. Усі ці перевірки підтвердили, що зв’язок між модулями реалізовано узгоджено й без логічних конфліктів.

Результат: усі сценарії обробляються очікувано. Взаємодія між сервісами побудована коректно, помилкові запити викликають передбачувані статус-коди та повідомлення.

2. Інтеграція GameService із WebSocket та SessionService перевірялась шляхом симуляції повного циклу ігрової сесії через з'єднання WebSocket. У процесі тестування підтверджено, що приєднання користувача до кімнати (roomJoin) коректно оновлює список учасників, транслює подію userJoined іншим клієнтам і готує сесію до старту. Після надходження події prepare, GameService успішно ініціює початок гри: формує унікальну колоду, роздає карти кожному гравцю та надсилає повідомлення gameStarted і yourCards. Черговість ходів у грі контролюється стабільно – система дозволяє зробити хід лише активному гравцеві, у визначений момент, ігноруючи будь-які передчасні або сторонні дії. Також протестовано ситуації, коли гравець залишає кімнату або втрачає з'єднання: у таких випадках GameService надсилає подію Client_disconnected, а RoomPlayersMap оновлюється відповідно до нового складу гравців. Усі ці дії свідчать про надійність взаємодії між GameService і SessionService, відсутність розсинхронізації та коректну обробку подій WebSocket на всіх етапах гри.

3. Інтеграція між GameService та BalanceService зосереджувалась на перевірці коректного обліку фінансових транзакцій під час гри. Особливу увагу було приділено синхронізації ігрових дій зі змінами балансу користувачів. Під час ставок у фазах betCircle та balancingCircle успішно фіксувалося зменшення балансу відповідного гравця одразу після здійснення ходу, із оновленням суми в базі даних. Після завершення гри, на етапі handleShowdown, система правильно визначала переможця (або кількох у разі поділу банку) та нараховувала виграш на їхній баланс, з дотриманням фінансової точності. Крім того, протестовано обробку помилкових сценаріїв – наприклад, у випадках, коли гравець намагався зробити ставку, що перевищувала його доступний баланс, система повертала відповідну помилку, не змінюючи стан банку чи бази. Таким чином, зв'язок між GameService та BalanceService показав себе надійним, із повною відповідністю очікуваним фінансовим правилам гри.

Результат: BalanceService правильно реагує на ігрові події, не допускає негативного балансу, фіксує зміни в історії, а GameService своєчасно оновлює

стан банку та балансу.

4. Інтеграція PrismaORM з PostgreSQL була спрямована на перевірку цілісності збережених даних і правильності взаємодії між сутностями бази. Було переконливо підтверджено, що всі етапи гри – від створення покер-сесії до фіксації ставок, визначення комбінацій і запису переможців – зберігаються у відповідних таблицях без втрати інформації або дублювання. Особливу увагу приділено валідації зовнішніх ключів: база не дозволяє створення ставки без відповідного запису про сесію, як і гравця без прив'язки до реального користувача. Усі спроби порушення референційної цілісності завершувалися очікуваними помилками, що свідчить про коректну побудову схеми. Також протестовано транзакційні операції – наприклад, при оновленні балансу користувача одночасно з завершенням гри: база гарантує, що в разі помилки жодна з пов’язаних змін не буде збережена частково. Це підтверджує стабільність, узгодженість та надійність зберігання ігрових та фінансових даних у межах всієї системи.

Результат: база даних поводиться стабільно, всі операції запису проходять з дотриманням референційної цілісності, помилки обробляються коректно.

Інтеграційне тестування підтвердило узгоджену роботу всіх ключових модулів системи. Жодних критичних помилок у логіці обміну даними, черговості запитів або реєстрації подій не виявлено. Зв’язки через REST API, WebSocket та базу даних функціонують надійно та дозволяють забезпечити повний цикл взаємодії користувача з ігровим застосунком.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи було створено серверну частину вебзастосунку для онлайн-карткової гри – покер. Розроблений програмний продукт має як технічне, так і соціальне значення. Зокрема, проєкт спрямований на формування альтернативи шкідливим звичкам серед молоді шляхом популяризації інтелектуальних онлайн-ігор.

У першому розділі проведено аналіз сучасного ринку інтернет-ігор, а також визначено соціальні аспекти їх поширення. Окрему увагу приділено вивченню сучасних технологій для створення серверної частини ігрових застосунків. Було обґрунтовано вибір таких інструментів, як NestJS, PostgreSQL, Prisma, WebSocket, Docker, які стали основою реалізації проєкту.

Другий розділ було присвячено безпосередньому проєктуванню та реалізацію серверної частини гри. Визначено функціональні та нефункціональні вимоги до системи, розроблено архітектуру, спроєктовано базу даних, реалізовано основні сервіси – авторизацію, особистий кабінет, ігрову логіку та підтримку реального часу через WebSocket-з’єднання. Архітектуру системи відображено на відповідних схемах.

У третьому розділі розглянуто методи тестування функціональних компонентів застосунку. Наведено опис модульного, інтеграційного. Водночас слід зазначити, що в майбутньому, коли проект стане більшим та прийде час розділити його на окремі сервіси, основа до чого закладена в архітектурі і обраних технологіях, потрібно буде додати тестування масштабованості.

Архітектура застосунку забезпечує модульність, масштабованість і стійкість до відмов. Використання Docker та балансувальника навантаження дозволяє підвищити гнучкість і ефективність розгортання системи. Комунікація між сервісами реалізована через REST API та WebSocket, централізоване управління базою даних здійснюється за допомогою Prisma ORM.

Таким чином, поставлені в роботі цілі досягнуто. Розроблено стабільну, масштабовану та безпечну серверну платформу для гри в покер із використанням сучасних підходів та інструментів бекенд-розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Docker: Docker Inc. Docker Documentation. URL: https://docs.docker.com. (Дата звернення: 30.05.2024.)

2. Prisma: Prisma Data Inc. Prisma ORM. Next-generation Node.js and TypeScript ORM. URL: https://www.prisma.io/docs (Дата звернення: 5.12.2025.)

3. REST API: Fielding R. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000. – URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (Дата звернення: 5.12.2024.)

4. WebSocket: Fette I., Melnikov A. The WebSocket Protocol. RFC 6455. – IETF, 2011. URL: https://datatracker.ietf.org/doc/html/rfc6455 – (Дата звернення: 5.12.2024.)

5. NestJS: NestJS. A progressive Node.js framework for building efficient, reliable and scalable server-side applications [Електронний ресурс]. – URL: https://docs.nestjs.com (Дата звернення: 30.05.2024.)

6. Git: Chacon S., Straub B. Pro Git. 2nd Edition [Електронний ресурс]. – Apress, 2014. – URL: https://git-scm.com/book/en/v2 (Дата звернення: 5.12.2025.)

7. Visual Studio Code (VSCode): Microsoft Corporation. Visual Studio Code Documentation. URL: https://code.visualstudio.com/docs (Дата звернення: 30.05.2025.)

8. PokerStars: PokerStars. Офіційний сайт. – URL: https://www.pokerstars.com (Дата звернення: 5.12.2024.)

9. Governor of Poker: Youda Games. Governor of Poker. URL: https://www.governorofpoker.com (Дата звернення: 6.12.2024.)

10. PokerTH: PokerTH Developers. PokerTH – The Open Source Texas Hold'em. – URL: https://www.pokerth.net (Дата звернення: 6.12.2024.)

11. Node.js: Node.js Documentation. – URL: https://nodejs.org/docs/latest/api/ (Дата звернення: 6.12.2024.)

12. TypeScript: Microsoft Corporation. TypeScript Documentation. – URL: https://www.typescriptlang.org/docs/ (Дата звернення: 6.12.2024.)

13. WebStorm: JetBrains. WebStorm – The Smartest JavaScript IDE. – URL: https://www.jetbrains.com/webstorm/ (Дата звернення: 6.12.2024.)

14. Kubernetes: Cloud Native Computing Foundation. Kubernetes Documentation. – URL: https://kubernetes.io/docs/home/ (Дата звернення: 6.12.2024.)

15. JWT: JSON Web Token Introduction. – URL: https://jwt.io/introduction (Дата звернення: 8.12.2024.)

ДОДАТОК А

Посилання на GitHub репозиторій: https://github.com/dimon289/Pocker.git