ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«УНІВЕРСИТЕТ ЕКОНОМІКИ ТА ПРАВА «КРОК»

КВАЛІФІКАЦІЙНА РОБОТА

Тема: «Інформаційна система управління даними гравців мобільної гри на основі мікросервісної архітектури»

Ступінь вищої освіти – бакалавр
Спеціальність – 122 «Комп’ютерні науки»
Освітня програма «Комп’ютерні науки»

ПОЯСНЮВАЛЬНА ЗАПИСКА

Виконав: здобувач 4 курсу
групи КН-21
Олександр ШЕПЕЛЬ
Керівник: к. військ. н. доцент кафедри
комп'ютерних наук
Володимир ТРОЦЬКО

Засвідчую, що кваліфікаційна робота оформлена відповідно до ДСТУ 3008:2015 та не містить запозичень з праць інших авторів без відповідних посилань.
Здобувач: ___________
(підпис)

м. Київ – 2025 рік

ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«УНІВЕРСИТЕТ ЕКОНОМІКИ ТА ПРАВА «КРОК»

ЗАТВЕРДЖУЮ:
завідувач кафедри
комп'ютерних наук
______Сергій МІЧКІВСЬКИЙ
«_____»____20___р

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ
Шепель Олександр Олександрови

Тема роботи	Інформаційна система управління даними гравців мобільної гри на основі мікросервісної архітектури
Номер та дата наказу про затвердження теми	№121-7 від 24 грудня 2024 року
Коротка постановка завдання	Розробка серверної архітектури на основі мікросервісів для організації торговельної системи між різними мобільними іграми, що дозволяє конвертацію ігрової валюти з однієї гри в іншу. Робота включає аналіз існуючих рішень, розробку архітектури, імплементацію ключових мікросервісів та тестування їх взаємодії.
Посилання на джерела інформації (не більше п'яти найменувань, які рекомендує науковий керівник)	1. Фаулер М. Мікросервіси: еволюційний дизайн розподілених систем // Мартін Фаулер. – Київ: Вид-во XYZ, 2021. – 320 с. 2. Hightower K., Burns B., Beda J. Kubernetes: Up & Running. – 3rd ed. – O'Reilly Media, 2022. – 368 p.
Вимоги до кваліфікаційної роботи	Кваліфікаційна робота має передбачити теоретичне, системотехнічне або експериментальне дослідження складного спеціалізованого завдання або практичної проблеми в галузі комп'ютерних наук, яке характеризується комплексністю та невизначеністю умов і потребує застосування теорій і методів інформаційних технологій.

Дата видачі завдання 24 грудня 2024 р.

Керівник Володимир ТРОЦЬКО

Здобувач освітнього ступеня бакалавра Олександр ШЕПЕЛЬ

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання	Примітка
Підготовчий етап			
1	Вибір напряму дослідження	02.12.2024 р.	*виконано*
2	Формування теми та призначення керівника	16.12.2024 р.	*виконано*
3	Затвердження теми кваліфікаційної роботи	23.12.2024 р.	*виконано*
4	Затвердження завдання на кваліфікаційну роботу	27.12.2024 р.	*виконано*
Основний етап			
5	Розробка концепції кваліфікаційної роботи	13.01.2025 р.	*виконано*
6	Підбір та вивчення джерел інформації з напряму дослідження. Огляд існуючих аналогів	20.01.2025 р.	*виконано*
7	Затвердження розширеної постановки завдання. Підготовка та подання керівникові розділу 1 кваліфікаційної роботи	10.03.2025 р.	*виконано*
8	Проєктування. Підготовка та подання керівникові розділу 2 кваліфікаційної роботи	24.03.2025 р.	*виконано*
9	Підготовка доповіді для експертизи стану виконання кваліфікаційної роботи (проміжний контроль)	31.03-04.04.2025 р.	*виконано*
10	Реалізація. Підготовка та подання керівникові розділу 3 кваліфікаційної роботи	07.04.2025 р.	*виконано*
11	Підготовка та подання керівнику першого варіанту всієї кваліфікаційної роботи	14.04.2025 р.	*виконано*
12	Доопрацювання кваліфікаційної роботи з урахуванням зауважень керівника та представлення керівникові доопрацьованого варіанту кваліфікаційної роботи	21.04.2025 р.	*виконано*
Завершальний етап			
13	Представлення рукопису для перевірки на плагіат	28.04-04.05.2025 р.	*виконано*
14	Підготовка презентації та доповіді на передзахист	05.05-11.05.2025 р.	*виконано*
15	Передзахист кваліфікаційної роботи	12.05-16.05.2025 р.	*виконано*
16	Доопрацювання роботи за результатами передзахисту	19.05-06.06.2025 р.	*виконано*
17	Експертиза роботи керівником та зовнішнім експертом	09.06-15.06.2025 р.	*виконано*
18	Доопрацювання доповіді та презентації для захисту	09.06-15.06.2025 р.	*виконано*
19	Захист кваліфікаційної роботи	16.06-22.06.2025 р.	*виконано*

Керівник Володимир ТРОЦЬКО

Здобувач освітнього ступеня бакалавра Олександр ШЕПЕЛЬ

Шепель О.О. Інформаційна система управління даними гравців мобільної гри на основі мікросервісної архітектури.

Пояснювальна записка до кваліфікаційної роботи за спеціальністю 122 – Комп’ютерні науки. – ВНЗ «Університет економіки та права «КРОК», Навчальнонауковий інститут інформаційних та комунікаційних технологій, кафедра комп’ютерних наук, Київ, 2025.

Описано розробку системи управління та контролю даними гравців мобільної гри на основі мікросервісної архітектури. З реалізацією інтеграційного пакету для використання в готових ігрових рішеннях.

Ключові слова: Мікро сервіси, серверна архітектура, бази даних, аналіз.

Рис.16. Бібліограф.: 29.

Shepel O. O. Information system of mobile game player data management based on microservice architecture.

Explanatory note to the qualification work in the specialty 122 - Computer Science - University of Economics and Law “KROK”, Educational and Research Institute of Information and Communication Technologies, Department of Computer Science, Kyiv, 2025.

The paper describes the development of a system for managing and controlling the data of mobile game players based on microservice architecture. With the implementation of an integration package for use in ready-made gaming solutions.

Keywords: Microservices, server architecture, databases, analysis.

Fig.16. Bibliography: 29.

ЗМІСТ

ЗМІСТ 5
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ 6
ВСТУП 8
РОЗДІЛ 1 ПОСТАНОВА ЗАВДАННЯ З РОЗРОБКИ 10
1.1 Опис предметної області 10
1.2 Визначення потенційних конкурентних переваг 12
1.3 Методичні підходи до дослідження 14
1.4 Постановка завдання 15
Висновки до розділу 19
РОЗДІЛ 2 ПРОЄКТУВАННЯ ТРЕЙДИНГОВОЇ СИСТЕМИ НА ОСНОВІ МІКРОСЕРВІСІВ 21
2.1 Моделювання даних та процесів трейдингової системи 21
2.3 Проєктування інтерфейсу API для інтеграції з Unity 26
2.4 Структура сховища даних 28
2.5 Використання програмного забезпечення 30
Висновки до розділу 30
РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ТРЕЙДИНГОВОЇ СИСТЕМИ . 32
3.1 Особливості реалізації програмного продукту 32
3.2 Реалізація сховища даних і обґрунтування його вибору 37
3.3 Результати тестування програмного продукту 39
3.4 Інструкція користувачеві та можливості використання 44
Висновки до розділу 46
ВИСНОВКИ 48
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 50
Додаток А 53
Додаток Б 55

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

API Інтерфейс програмування додатків — набір правил і засобів для обміну даними між програмами.[11]

Async/await Механізм асинхронного програмування — спосіб організації коду для одночасного виконання кількох завдань без блокування роботи програми. [12]

C# .NET Мова програмування C# і платформа .NET — інструменти для створення швидких і надійних програм, які працюють на різних пристроях. [13]

Entity Framework Бібліотека для роботи з базами даних — інструмент, який спрощує збереження та отримання даних у програмах. [14]

HTTPS Протокол безпечного зв'язку в інтернеті — система захисту даних під час їхньої передачі. [15]

JWT Токен JSON — спосіб безпечної передачі інформації між програмами для ідентифікації користувачів. [16]

Мікросервіси Модульна структура програми — підхід, коли система складається з невеликих незалежних частин, кожна з яких виконує свою задачу. [17]

NFT Невзаємозамінний токен — унікальний цифровий об'єкт, який може представляти активи в іграх. [18]

PostgreSQL Система управління базами даних — програма для зберігання та обробки великих обсягів інформації. [19]

Roblox Платформа для створення та гри у відеоігри — онлайн-середовище, де користувачі розробляють і діляться власними іграми. [20]

SDK Набір інструментів для розробки програм — комплект програм і документації для підключення до системи. [21]

SignalR Бібліотека для зв'язку в реальному часі — інструмент, який дозволяє програмам швидко обмінюватися даними. [22]

Steam Цифрова платформа для ігор — сервіс для купівлі, гри та торгівлі віртуальними предметами в іграх. [23]

Unity Ігровий рушій — платформа для створення відеоігор, популярна серед розробників. [24]

Серверна архітектура Структура програми на сервері — організація системи, яка працює в інтернеті й обробляє запити від користувачів. [25]

ВСТУП

Актуальність теми. У сучасних умовах стрімкого розвитку ігрової індустрії створення трейдингових систем для обміну валютами та активами між різними іграми набуває особливої актуальності. Зростання популярності кросплатформних економік і потреба в гнучких, масштабованих рішеннях підкреслюють важливість розробки серверних архітектур на основі мікросервісів. Проблематикою створення таких систем займалися як вітчизняні, так і закордонні дослідники: наприклад, роботи Дж. Льюїса та М. Фаулера розкривають принципи мікросервісної архітектури [2], а дослідження українських авторів, таких як О. Субботін, акцентують на інтеграції серверних рішень із клієнтськими платформами. Водночас питання ефективної інтеграції трейдингових систем із ігровими рушіями, такими як Unity, а також забезпечення універсальності для різних ігор залишаються недостатньо розв’язаними. Саме ці прогалини дана робота прагне усунути, пропонуючи нове рішення для ігрової індустрії.

Мета дослідження. Метою дослідження є розробка серверної архітектури на основі мікросервісів для організації трейдингової системи між різними іграми з можливістю обміну валютами, а також її інтеграція з ігровим середовищем Unity через спеціально створений SDK.

Завдання дослідження. Для реалізації поставленої мети визначено такі завдання:

- вивчити сучасні підходи до створення мікросервісних архітектур у контексті ігрових систем;
- описати принципи проєктування трейдингової системи для обміну валютами;
- розробити серверну частину системи з використанням C# .NET;
- створити SDK для інтеграції з іграми на Unity;
- встановити працездатність системи шляхом тестування на прикладі гри, розробленої на Unity;

Об'єктом дослідження є процеси створення та функціонування трейдингових систем у віртуальних ігрових середовищах.

Предметом дослідження є серверна архітектура на основі мікросервісів, призначена для організації обміну валютами та активами між іграми, та її інтеграція з ігровим середовищем Unity.

Методи дослідження. Для виконання дослідження я застосував кілька методів, які допомогли розібратися в темі та досягти поставлених цілей. Спочатку я провів аналіз літератури, щоб вивчити сучасні підходи до створення мікросервісних архітектур і оцінити, як їх використовують в ігрових системах. Далі я зосередився на проєктуванні: розробив архітектуру системи та SDK, використовуючи C# .NET і Unity, що дозволило створити основу для торгівельної платформи. Щоб переконатися в працездатності системи, я провів експериментальні тести, перевіряючи її функціональність на практиці. Нарешті, я порівняв отримані результати з існуючими аналогами, щоб оцінити ефективність і виявити сильні сторони розробленого рішення.

Інформаційні технології та програмне забезпечення. У процесі дослідження використано стандартне програмне забезпечення: середовище розробки JetBrains Rider для C# .NET, Unity для створення клієнтської частини, а також авторський код для реалізації серверної архітектури та SDK.

Практичне значення. Результати даного дослідження можуть бути використані розробниками ігрових платформ для створення гнучких екосистем, що підтримують крос-ігровий обмін ресурсами. Запропонована інформаційна система дозволить не лише підвищити рівень користувацького досвіду, а й зможе посприяти розробленню нових бізнес моделей. Крім того, досвід реалізації мікросервісної архітектури може бути застосований у суміжних галузях, зокрема у фінансових та блокчейн-технологіях.

Структура та обсяг пояснювальної записки. Кваліфікаційна робота складається зі вступу, трьох розділів, висновків та списку посилань (**29** найменувань). Пояснювальна записка містить **27** рисунків. Загальний обсяг пояснювальної записки складає 61 сторінку, основний зміст викладено на 53 сторінках.

РОЗДІЛ 1
ПОСТАНОВА ЗАВДАННЯ З РОЗРОБКИ

1.1 Опис предметної області

У межах дослідження розглядається створення торгівельних систем для віртуальних ігрових середовищ, що дозволяють обмінюватися валютами та активами між різними іграми. Економічні механізми в іграх відіграють ключову роль у дизайні, адже вони сприяють активнішій участі гравців і забезпечують розробникам додатковий дохід. За звітом Newzoo (2024), 65% гравців використовують кілька платформ, а за прогнозами Statista, ринок віртуальних товарів досягне $50 млрд у 2025 році. Історично економіка в іграх починалася з простих механізмів обміну, таких як торгівля предметами в Diablo II (2000), і з часом розвинулася до складних систем, як-от аукціони в World of Warcraft(на даний момент закритий) чи Steam Community Market, запущений у 2012 році.

Економічні системи в іграх сьогодні відіграють важливу роль через кілька ключових тенденцій. По-перше, зростає популярність кросплатформних ігор: згідно зі звітом Newzoo (2024), 65% гравців використовують кілька платформ, що створює потребу в універсальних рішеннях для обміну ресурсами. По-друге, торгівля та внутрішньоігрові покупки стали значним джерелом доходу: за прогнозами Statista, ринок віртуальних товарів досягне 50 мільярдів доларів у 2025 році. Нарешті, торгівля між гравцями сприяє соціальній взаємодії, що допомагає утримувати аудиторію в багатокористувацьких проєктах і робить ігровий досвід більш захопливим.

Мікросервісна архітектура, концепція якої була формалізована Дж. Льюїсом і М. Фаулером у 2014 році [2], ідеально підходить для таких систем завдяки її модульності, незалежності сервісів і здатності масштабуватися. Вона дозволяє розділити логіку торгівлі, обробку транзакцій і взаємодію з клієнтами, що особливо важливо для інтеграції між різними іграми. Проте

сучасні рішення, як-от у Roblox(рис. 1.2) чи Steam(рис. 1.1) [6][7], обмежені однією платформою, а універсальні системи для обміну між іграми досі не набули широкого поширення. Це відкриває простір для інновацій, які можуть змінити підхід до ігрових економік.

На рис. 1.1 зображено торговий майданчик Steam на якому видно предмети які продаються, видно тип предмету в лапках зазначено його назву, кількість товару на торговому майданчику та його ціну за одиницю, з правої сторони показано фільтри за іграми.

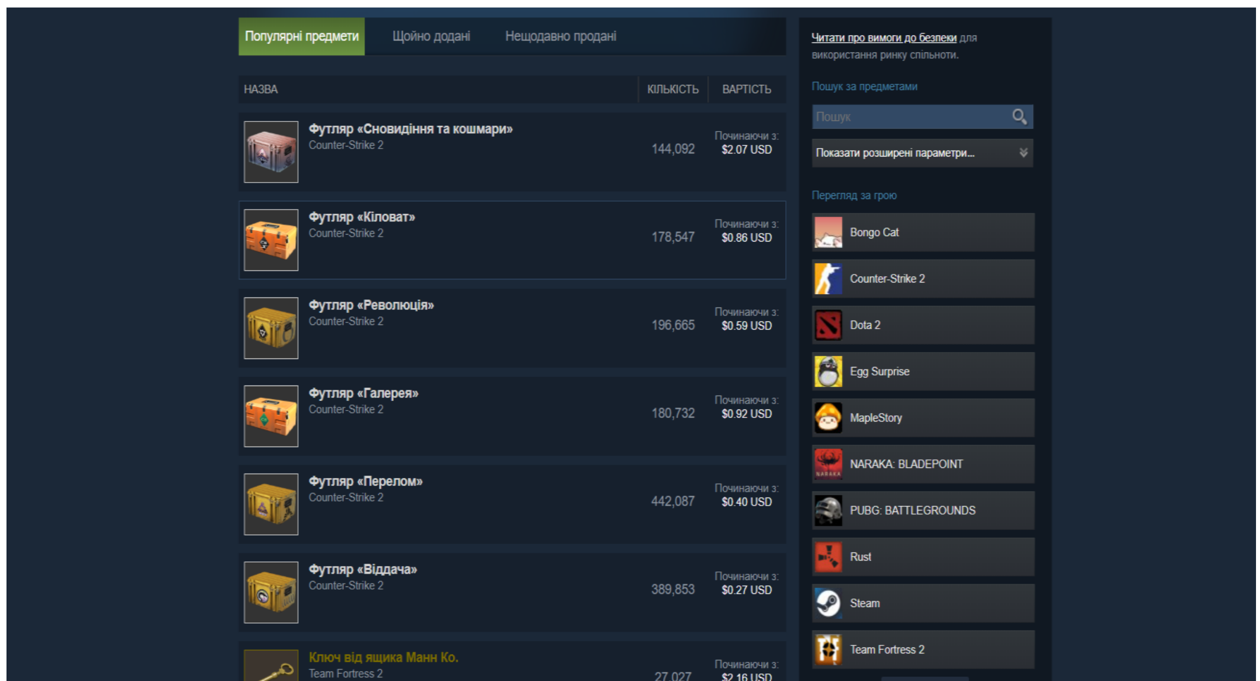

Рисунок 1.1 – Візуальний вигляд торгового майданчики в Steam

Джерело: [9]

На рис. 1.2 зображено торговий майданчик Roblox на якому видно фільтри категорій товарів які здебільшого використовуються для створення контенту в ігровому просторі які розроблюють самі гравці та вже в середині створеного ігрового простору можуть продавати іншим гравцям за внутрішньо ігрову валюту(Robux).

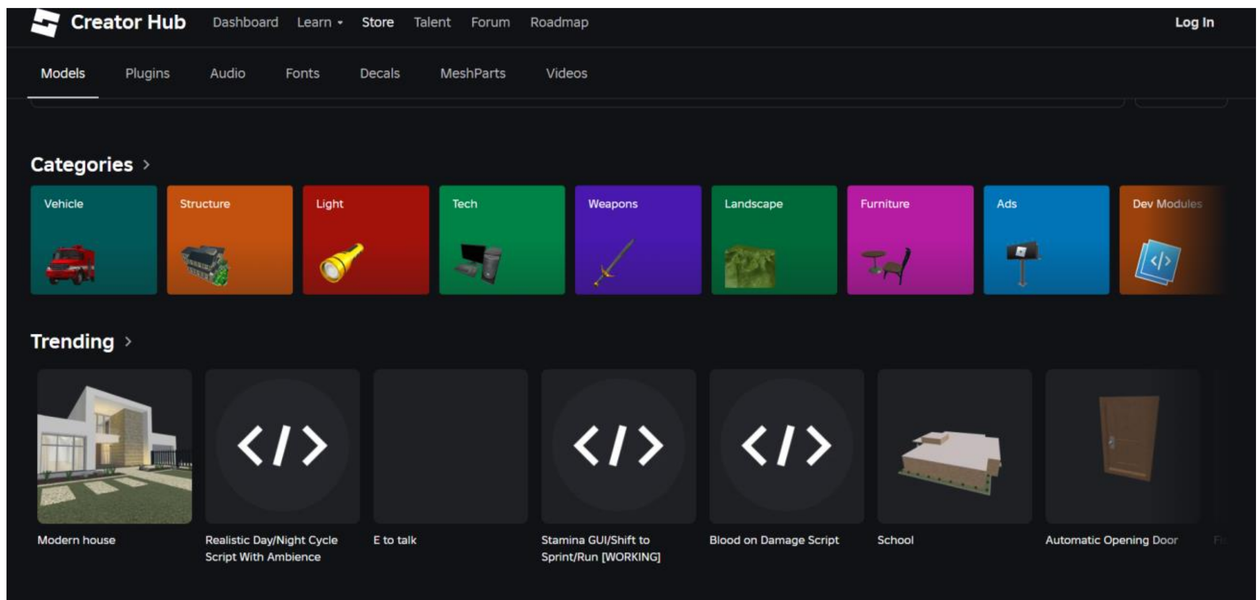

Рисунок 1.2 – Візуальний вигляд торгового майданчики в Roblox

Джерело: [10]

1.2 Визначення потенційних конкурентних переваг

Серверна архітектура, заснована на мікросервісах і доповнена SDK для Unity, має кілька сильних сторін, які, на мою думку, роблять її кращою за наявні аналоги:

– універсальність — на відміну від Steam Trading, яке працює лише в межах власної екосистеми, розроблена система дозволяє інтегруватися з будь-якою грою на Unity, а в майбутньому — і з іншими ігровими рушіями;

– масштабованість — завдяки використанню мікросервісної архітектури є можливість додавати нові функції, наприклад підтримку криптовалют чи NFT, без необхідності переробляти всю систему. Це є суттєвою перевагою над монолітними рішеннями [3];

– простота інтеграції — розроблений SDK значно спрощує підключення до системи: розробникам потрібно лише кілька днів, тоді як створення власної торгівельної системи може зайняти місяці [6];

– гнучкість — система забезпечує конвертацію валют між іграми з різними економічними моделями, наприклад обмін золота з RPG на кристали з мобільних ігор, що відкриває можливості для кроспромоції.

Ці особливості приносять користь усім учасникам процесу:

– гравці отримують змогу торгувати активами між різними іграми, що робить їхній час і вкладення більш цінними. Наприклад, обмін ресурсів однієї гри на валюту іншої може спонукати гравців відкривати нові проєкти [1];

– інвестори бачать потенціал для заробітку через комісії з транзакцій чи підписки, що підтверджують ринкові тенденції [5];

– ігрові студії, особливо невеликі, можуть легко впроваджувати економічні механізми й конкурувати з великими платформами завдяки кросплатформним можливостям [7].

Для наочності нижче наведено в табл. 1.1 порівняння теоретичної моделі з аналогами — Steam Community Market і Roblox Creator Hub:

Таблиця 1.1–Порівняння трейдингових систем

Критерій	*Steam Community Market[6]*	*Roblox Economy[7]*	*Теоретична модель[3]*
Платформна гнучкість	Обмежена Steam (закрита)	Обмежена Roblox	Універсальна (Unity + потенційно інші)
Тип архітектури	Монолітна	Монолітна	Мікросервісна
Час інтеграції	Місяці (власна розробка)	Не інтегрується ззовні	Дні (через SDK)
Масштабованість	Обмежена (залежить від Steam)	Середня (в межах Roblox)	Висока (додавання сервісів)
Конвертація валют	Відсутня	Відсутня	Підтримується
Продуктивність	До 5000 транзакцій/с (пікові)	До 1000 транзакцій/с	Ціль: 1000 транзакцій/с
Доступність для студій	Тільки партнери Steam	Тільки розробники Roblox	Будь-які студії з Unity

Ця таблиця демонструє, що теоретична модель може перевершувати аналоги за гнучкістю, масштабованістю та доступністю, що робить її привабливою для широкого кола користувачів і розробників.

1.3 Методичні підходи до дослідження

Для розробки серверної архітектури торгівельної системи між іграми на основі мікросервісів застосовано кілька методів, що охоплюють теоретичну базу, практичну реалізацію та перевірку результатів. Кожен метод детально адаптовано до специфіки предметної області, а саме створення економічних систем у ігрових екосистемах.

Теоретичний аналіз. Для аналізу сучасних ігрових економік і технологій мікросервісів було використано кілька авторитетних джерел. Зокрема, монографія Сема Ньюмана "Building Microservices" (2021) надала ґрунтовне розуміння принципів створення масштабованих систем, особливо для обробки транзакцій у реальному часі. Матеріали конференції Game Developers Conference (GDC) 2023 дали змогу ознайомитися з новими трендами монетизації, такими як зростання популярності NFT і кросплатформних економік. Дослідження Річарда Бартла (2004) про психологію гравців підтвердило важливість економічних стимулів для утримання аудиторії.

Окремо було проаналізовано роботу ігрових екосистем, зокрема Steam. За даними Steamworks (2024), їхній Community Market обробляє понад 1,5 мільйона транзакцій щодня, отримуючи прибуток з комісій у розмірі від 5 до 15 % [6]. У 2023 році Steam підтримував торгівлю в більш ніж 500 іграх, однак його система залишається закритою для інших платформ, а середній час обробки транзакції становить 200–300 мс. Ці показники стали орієнтиром для подальшої роботи, водночас продемонструвавши обмеження монолітної архітектури Steam, що стало аргументом на користь вибору мікросервісного підходу.

Системний підхід. Розглядаючи торгівельну систему як набір взаємопов’язаних компонентів, було спроєктовано кілька ключових мікросервісів: один відповідає за авторизацію з використанням JWT для забезпечення безпеки, інший — за обробку купівлі та продажу, ще один — за управління конвертацією валют із динамічними курсами, а останній — за

інтеграцію через API для Unity. Такий підхід дав змогу ізолювати функціональні модулі, що значно полегшило тестування й масштабування. Для порівняння, у Steam усі функції торгівлі об’єднані в єдину систему, що ускладнює оновлення окремих модулів.

Експериментальний метод. Для перевірки працездатності системи заплановано створення прототипу та його тестування. Передбачається моделювання навантаження від 1000 одночасних гравців, які здійснюють транзакції кожні дві секунди, а також перевірка інтеграції на демонстраційній грі в Unity із двома валютами, такими як "золото" та "кристали". Час відгуку системи буде порівняно з показниками Steam (200–300 мс), використовуючи інструменти JMeter для тестування навантаження та Unity Profiler для аналізу клієнтської частини.

Заплановано також проведення порівняльного аналізу за кількома критеріями: продуктивність (кількість транзакцій за секунду, де Steam досягає 5000 у пікові періоди), гнучкість (можливість інтеграції з різними іграми, на відміну від закритої платформи Steam) і масштабованість (час додавання нового сервісу, який у мікросервісній архітектурі займає дні, а в монолітних системах — тижні). Для прикладу, економічна система Roblox (Robux) обмежена внутрішньою торгівлею, тоді як запропоноване рішення підтримує кросплатформний обмін, що є його значною перевагою.

Ці підходи ґрунтуються на сучасних джерелах, таких як документація Microsoft .NET, статті з IEEE Transactions on Software Engineering і практичні кейси від Unity. Поєднання теоретичного аналізу та практичних експериментів заклало міцну основу для розробки й оцінки системи.

Методичні підходи до дослідження. У дослідженні було використано кілька ключових методів. Проведено аналіз наукових публікацій та технічної документації з метою вивчення управління ігровими ресурсами, мікросервісних архітектур і механізмів торгівлі. Розроблено схему взаємодії між мікросервісами, змодельовано процеси обміну ресурсами та їх збереження. Для практичної реалізації створено мінімально життєздатний

продукт (MVP) і виконано тести продуктивності та безпеки. Проведено порівняння розробленої системи з аналогами з метою оцінки її ефективності та визначення переваг.

1.4 Постановка завдання

Завдання кваліфікаційної роботи полягає в розробці серверної архітектури на основі мікросервісів для трейдингової системи, яка забезпечить обмін валютами та активами між іграми, з інтеграцією через SDK для Unity. Задача включає створення масштабованого, надійного програмного продукту, здатного обробляти торговельні операції в реальному часі.

Компоненти архітектури. Вибір C# .NET.

Під час відбору технологій для торгівельної системи пріоритетними критеріями були продуктивність і зручність розробки для навантажених сценаріїв. Після аналізу кількох варіантів було обрано C# та .NET як оптимальні для поставлених вимог. Основними причинами такого вибору стали:

– висока продуктивність. За даними тестів TechEmpower [8], .NET здатен обробляти до 7 мільйонів запитів за секунду. Для торгівельної системи, у якій тисячі гравців можуть одночасно здійснювати транзакції, це є критично важливим показником;

– асинхронна обробка. Вбудована підтримка async/await у C# дала змогу організувати паралельну обробку транзакцій, що значно зменшило затримки. Зокрема, у модулі TradeService застосовано асинхронні виклики для оперативного оновлення балансів;

– багата екосистема. Бібліотеки .NET, такі як Entity Framework Core для роботи з PostgreSQL і SignalR для оновлень у реальному часі, спростили процес розробки та дозволили зосередитися на бізнес-логіці системи, мінімізуючи роботу з низькорівневими задачами;

– кросплатформність. Підтримка Windows, Linux і macOS надала можливість розгортати серверну частину на різних платформах. Тестування системи проводилося також на Linux-сервері, що виявилося зручним для контейнеризації.

Порівняльний аналіз показав, що C# забезпечує кращу типізацію та продуктивність у сценаріях із високим навантаженням, ніж Node.js чи Python. Хоча Node.js є швидким для асинхронних операцій, він поступається в типобезпеці, а Python демонструє нижчу швидкодію для подібних завдань. Саме ці чинники зумовили вибір C# .NET як основної технології.Системні вимоги.

Щоб система працювала стабільно, були визначені мінімальні вимоги:

Операційна система. Сервер може функціонувати на Windows або Linux, а клієнтська частина вимагає Unity 2021.3 або новішу версію. Пам'ять. Для сервера потрібно щонайменше 4 ГБ оперативної пам'яті, щоб забезпечити обробку транзакцій без затримок.

Функціональні вимоги. Під час розробки системи було поставлено за мету реалізувати кілька ключових функцій:

– можливість обмінюватися валютами між різними іграми з автоматичною конвертацією за заданими курсами, що дозволяє гравцям торгувати ресурсами;

– API для управління транзакціями та інтеграції з ігровими платформами, що забезпечує просте підключення для розробників;

– підтримка різних видів торгівлі, зокрема купівлі, продажу й аукціонів, для гнучкості економічних моделей.

Нефункціональні вимоги. Окрім функціоналу, увагу приділено якості системи:

– Продуктивність. Система повинна обробляти до 1000 транзакцій за секунду, щоб підтримувати велику кількість гравців;

– Надійність. Система має забезпечувати uptime на рівні 99,9%, щоб мінімізувати простої;

– Безпека. Усі транзакції захищені через HTTPS і JWT-токени, що гарантує конфіденційність даних.

Підзавдання дослідження. Для досягнення поставлених цілей роботу було розбито на кілька етапів:

– вивчення вимог до торгівельних систем із аналізом досвіду таких платформ, як Steam і Roblox;

– розробка мікросервісів для обробки транзакцій і конвертації валют з метою забезпечення модульності й масштабованості;

– створення SDK із готовими прикладами для Unity, щоб спростити інтеграцію для розробників;

– проведення тестування на демо-грі з двома валютами (наприклад, «золото» і «кристали») для перевірки працездатності системи.

Очікувані результати. Працездатний прототип із документацією, що демонструє обмін валютами між іграми.

Для кращого розуміння структури майбутньої системи нижче наведено спрощена архітектурна схема торгівельної системи, яка використовує мікросервісну архітектуру. Система складається з кількох компонентів, що взаємодіють між собою через API Gateway, забезпечуючи зв'язок між клієнтом (Unity SDK) та серверною частиною(рис. 1.3).

Схема представляє розподілену систему, де:

– клієнт — (Unity SDK) взаємодіє з сервером через API Gateway.

– API Gateway — виступає єдиною точкою входу для всіх клієнтських запитів і маршрутизує їх до відповідних мікросервісів.

– мікросервіси — Auth Service, Integration Service, Currency Service, Trade Service обробляють запити, кожен відповідаючи за свою функціональність.

– база даних(на рисунку зображена узагальнено) — підключена до кожного мікросервісу для зберігання специфічних даних.

Схема ілюструє модульний підхід до побудови системи, де кожен мікросервіс виконує окрему функцію, що відповідає принципам мікросервісної архітектури [3].

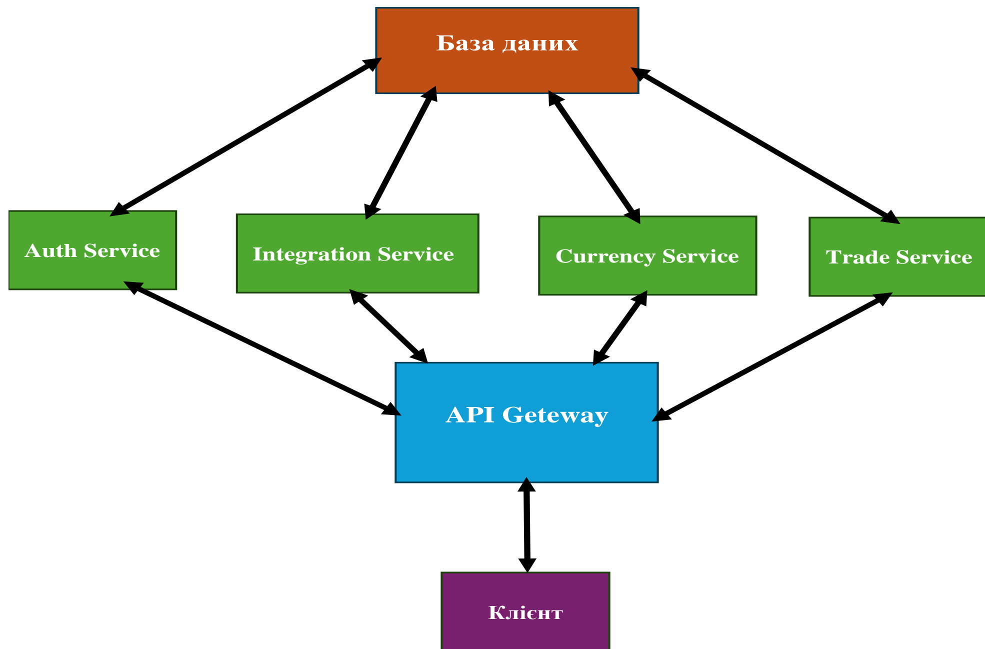

Рисунок 1.3 – Архітектура трейдингової системи

Джерело: розроблено автором

Висновки до розділу

Розділ присвячено теоретичному обґрунтуванню розробки серверної архітектури на основі мікросервісів для трейдингової системи між іграми. Проведений аналіз предметної області показав, що економіка в іграх є критично важливою для сучасної індустрії: за даними Steam, торгівля віртуальними активами залучає мільйони гравців і генерує значний дохід, але існуючі рішення, як-от Steam Community Market, обмежені своєю закритістю й монолітністю. Вибір C# .NET обґрунтовано його продуктивністю (до 7 млн запитів/с) і підтримкою асинхронних операцій, що ідеально підходить для навантажених систем із реальним часом.

Визначено конкурентні переваги розробки: універсальність, масштабованість і простота інтеграції через SDK для Unity, що забезпечує залученість гравців (за рахунок кросплатформного обміну), інтерес

інвесторів (через потенціал монетизації[5]) та вигоду для студій (шляхом спрощення розробки економік). Постановка завдання включає чіткі вимоги — від обробки 1000 транзакцій/с до створення безпечного API, — а методичні підходи (аналіз Steam, експерименти з навантаженням) створюють міцну базу для практичної реалізації.

Отже, розділ установив теоретичний фундамент, який демонструє актуальність теми в контексті сучасних трендів (зростання кросплатформності, монетизація через торгівлю) і переваги мікросервісів над монолітними системами типу Steam.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТРЕЙДИНГОВОЇ СИСТЕМИ НА ОСНОВІ МІКРОСЕРВІСІВ

У цьому розділі описано проєктування торгівельної системи для обміну валютами та активами між іграми з інтеграцією через Unity. На основі аналізу предметної області розроблено структуру даних, архітектуру, API, сховище даних і програмне забезпечення з урахуванням вимог до масштабованості, гнучкості та безпеки.

2.1 Моделювання даних та процесів трейдингової системи

Моделювання даних і процесів є ключовим етапом проєктування трейдингової системи, що забезпечує кросплатформний обмін ресурсами та захист від шахрайства. Метою цього підпункту є створення структури даних і логіки операцій — авторизації, транзакцій, конвертації валют і валідації ігрових подій — із врахуванням вимог, визначених у першому розділі.

Моделювання даних. Система оперує такими сутностями (табл 2.1):

– користувач — ідентифікатор (UUID), ім'я, email, токен авторизації (JWT), дата реєстрації — унікальні дані гравця;

– гра — ID (UUID), назва, опис, список валют (JSON), статус підключення (Boolean) — інформація про підключену гру;

– валюта — ID (UUID), назва, код (наприклад, "GLD"), курс обміну (Float), гра-власник (UUID), обсяг у обігу (Float) — характеристики валюти;

– актив — ID (UUID), тип (Enum: валюта, предмет), назва, власник (UUID), ціна (Float), статус (Enum: доступний, зарезервований) — дані про ігровий актив;

– транзакція — ID (UUID), відправник (UUID), отримувач (UUID), тип (Enum: купівля, продаж, обмін), сума (Float), валюта (UUID), дата (DateTime), статус (Enum: у процесі, завершено) — деталі операції;

– GameEvent — ID (UUID), ГраID (UUID), КористувачID (UUID), тип події (String), значення (JSON), дата (DateTime), статус (Enum: Pending, Validated, Rejected) — ігрова подія для валідації;

– ValidationRule — ID (UUID), ГраID (UUID), тип події (String), умови (JSON), опис (String) — правила чесності гри;

Таблиця 2.1 – Структура основних сутностей

Сутність	*Поля*	*Тип даних*	*Опис*
Користувач	ID, Ім'я, Email, JWT, Дата	UUID, String, String, String, DateTime	Унікальні дані гравця
Гра	ID, Назва, Опис, Валюти, Статус	UUID, String, String, JSON, Boolean	Інформація про гру
Валюта	ID, Назва, Код, Курс, Гра, Обсяг	UUID, String, String, Float, UUID, Float	Характеристики валюти
Актив	ID, Тип, Назва, Власник, Ціна, Статус	UUID, Enum, String, UUID, Float, Enum	Дані про актив
Транзакція	ID, Відправник, Отримувач, Тип, Сума	UUID, UUID, UUID, Enum, Float	Деталі операції обміну
GameEvent	ID, ГраID, КористувачID, Тип, Значення, Дата, Статус	UUID, UUID, UUID, String, JSON, DateTime, Enum	Ігрова подія
ValidationRule	ID, ГраID, Тип, Умови, Опис	UUID, UUID, String, JSON, String	Правила валідації

Моделювання процесів:

1. авторизація — користувач надсилає запит із логіном і паролем → Auth Service генерує JWT → токен повертається клієнту;
2. транзакція — клієнт обирає актив → Trade Service перевіряє доступність → Currency Service конвертує валюту → запис у PostgreSQL → оновлення через SignalR;
3. конвертація — запит на обмін (наприклад, 100 золота на 50 кристалів) → Currency Service розраховує курс → оновлює баланс;

4. валідація ігрової події:
   - гра через SDK надсилає івент до API Gateway;
   - GameValidation Service отримує івент, витягує правила з ValidationRules для гри та типу події;
   - перевірка: порівняння значень івенту з умовами;

Більш наглядно яка послідовність транзакцій та валідацій розглянемо на рис. 2.1.

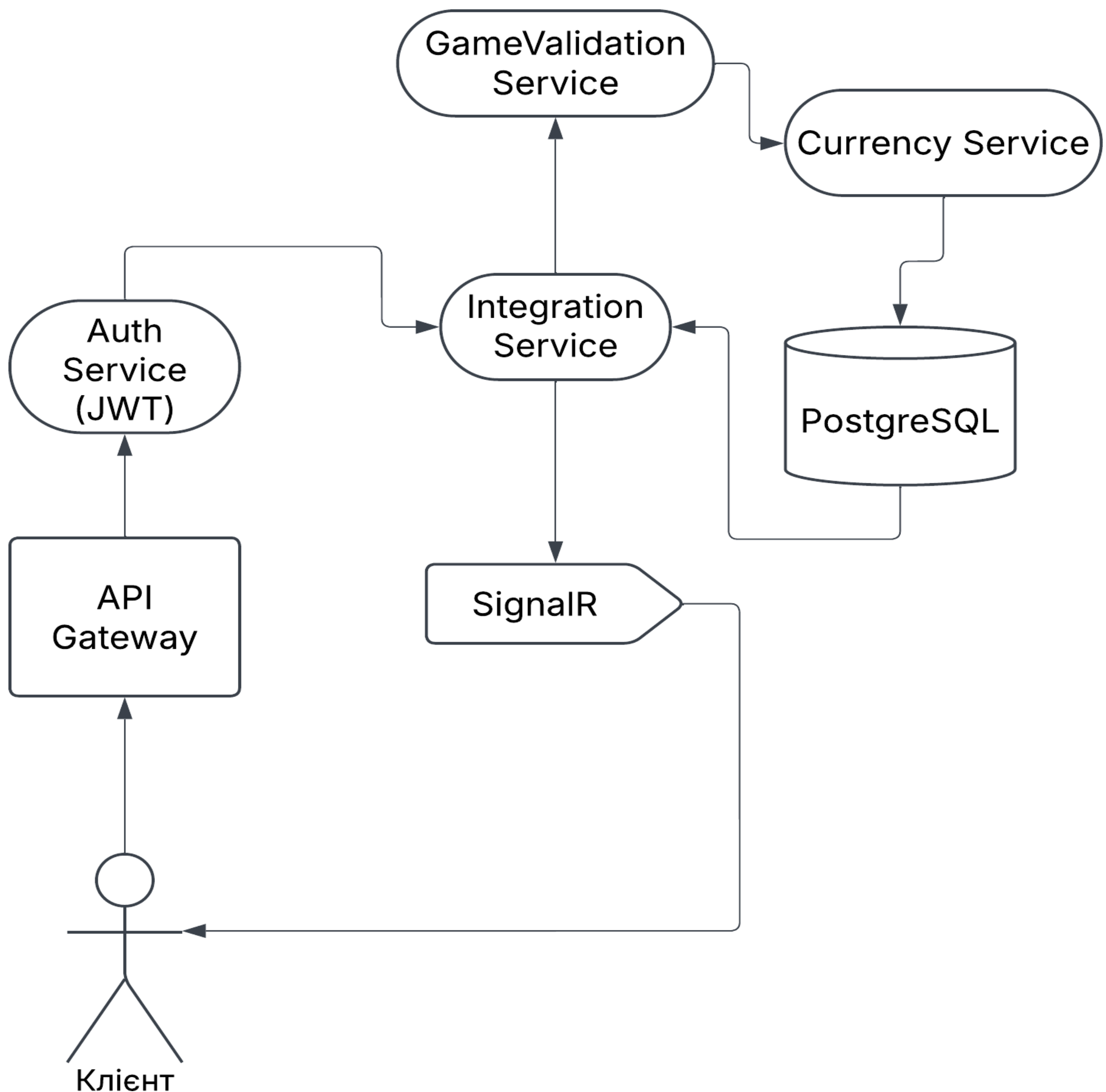

Рисунок. 2.1 – Діаграма послідовності транзакції та валідації

Джерело: розроблено автором

Підхід до валідації ігрових подій виявився ефективним завдяки кільком ключовим перевагам:

Усі зовнішні запити проходять через єдину точку входу, що значно спрощує впровадження автентифікації, логування й обмеження запитів, а також забезпечує високий рівень безпеки.

Кожен мікросервіс має чітко визначену роль, що полегшує розробку, тестування та масштабування системи.

Взаємодія між сервісами під час обробки транзакцій — наприклад, перевірка доступності активів чи конвертація валют — дозволяє швидко реагувати на дії гравців у реальному часі.

Клієнти отримують оновлення стану через SignalR без необхідності постійно запитувати сервер, що зменшує навантаження на API та покращує ігровий досвід.

GameValidation Service гнучко перевіряє ігрові події за збереженими правилами, що дає змогу адаптувати логіку валідації до різних ігор без зміни коду.

2.2 Опис архітектури програмного продукту

Компоненти архітектури. Торгівельну систему було спроєктовано як набір мікросервісів, кожен із яких виконує чітко визначену роль. Ключові компоненти організовані таким чином, щоб забезпечити гнучкість, безпеку та масштабованість.

API Gateway (Ocelot). Цей сервіс виступає єдиною точкою входу для всіх запитів, що значно спрощує маршрутизацію та балансування навантаження. Використання C# .NET і Ocelot дозволило налаштувати обробку пікових навантажень, наприклад, тисяч одночасних транзакцій. У базі даних зберігаються логи запитів (ID, час, IP-адреса та статус), що забезпечує відстеження активності й швидке виявлення проблем.

Auth Service. Цей сервіс відповідає за авторизацію через JWT-токени та управління сесіями користувачів. Для його реалізації застосовано

ASP.NET Core Identity і JWT, а для безпеки паролів — хешування через bcrypt. У базі даних таблиця *Users* містить ID, ім'я, email, хеш пароля, токен і дату створення, що дозволяє надійно зберігати дані користувачів.

Trade Service. Сервіс призначений для обробки транзакцій — купівлі, продажу чи аукціонів. Він залежить від *GameValidation Service*, що перевіряє відповідність нарахувань правилам гри. Використовуючи C# .NET і Entity Framework, було організовано базу даних із таблицями *Transactions* (ID, відправник, отримувач, тип, сума, валюта, статус) та *Assets* (ID, тип, назва, власник), що забезпечує швидкий доступ до даних транзакцій.

Currency Service. Сервіс реалізує управління валютами, їх конвертацію та оновлення балансів гравців після валідації подій. Працюючи на C# .NET, він використовує власний алгоритм для динамічного визначення курсів обміну. У базі даних таблиці *Currencies* (ID, назва, код, курс, гра) і *CurrencyRates* (ID, валюта1, валюта2, курс, дата) забезпечують гнучке керування економікою ігор.

Integration Service. Цей сервіс відповідає за інтеграцію з Unity через SDK та передачу ігрових подій на валідацію. Використання C# .NET і SignalR дозволяє забезпечити передачу даних у реальному часі. У базі даних таблиця *Games* (ID, назва, валюти, статус) використовується для відстеження підключених ігор та їхніх налаштувань.

GameValidation Service. Для захисту системи від шахрайства реалізовано окремий сервіс перевірки ігрових подій. Наприклад, у раундовій грі він аналізує, чи відповідає час проходження (30–300 секунд) встановленим правилам, а для дії *KillEnemy* перевіряє коректність нагороди (10 золота). Успішні події надсилаються до *Currency Service*, а невдалі логуються. Сервіс працює на C# .NET із Entity Framework, використовуючи JSON-парсер і async/await для швидкої обробки. У базі даних таблиці *GameEvents* (ID, граID, користувачID, тип, значення, дата, статус) та *ValidationRules* (ID, граID, тип, умови, опис) дають змогу адаптувати правила

до різних ігор. Завдяки JSON-правилам сервіс є гнучко налаштованим, а його ізольована архітектура спрощує масштабування.

Взаємодія: Integration Service приймає ігрові події від Unity і передає їх до GameValidation Service для перевірки. Якщо подія валідна, Currency Service оновлює баланс користувача, а Trade Service реєструє транзакцію в разі обміну. API Gateway забезпечує оновлення клієнта через SignalR.

Архітектуру системи можна побачити на рис. 2.2.

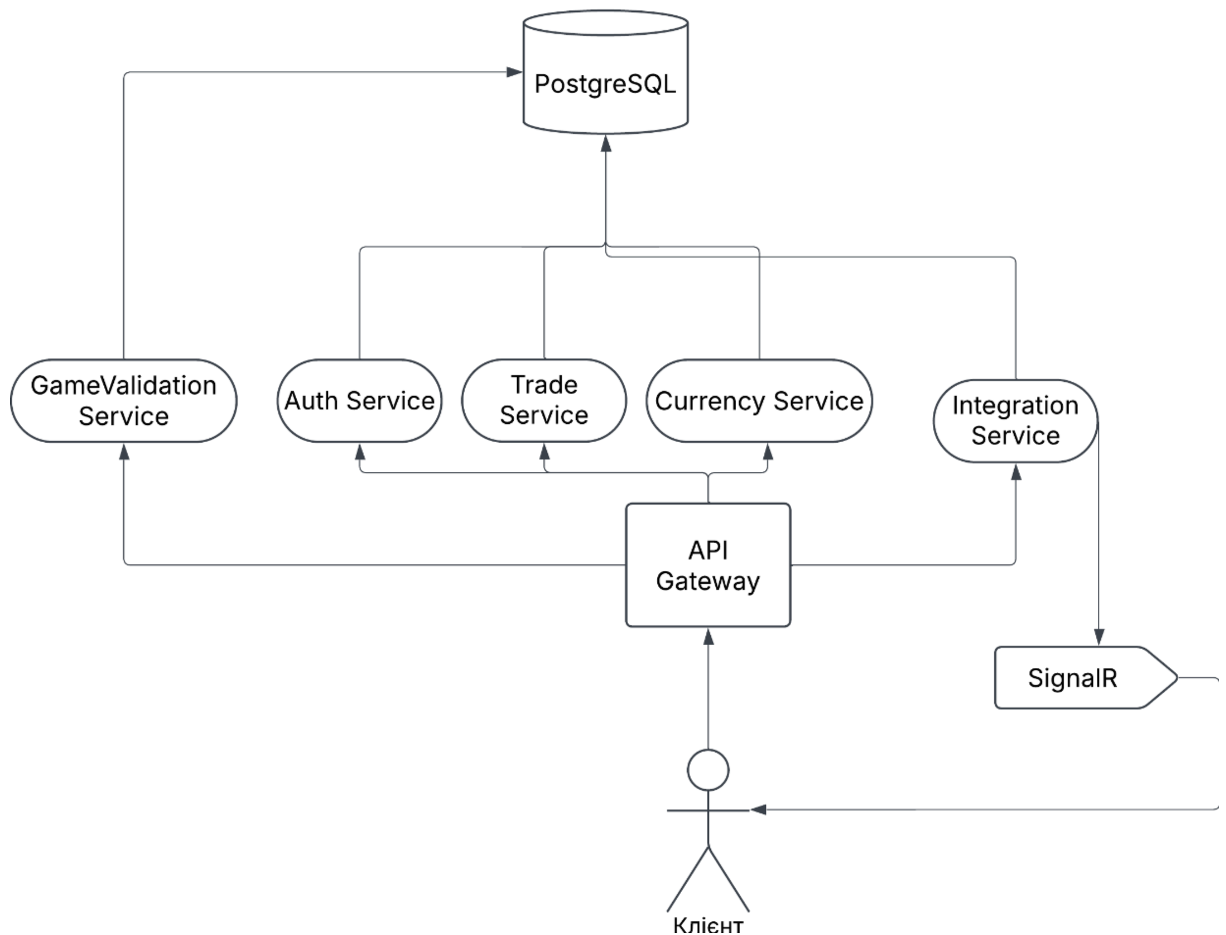

Рис. 2.2 – Архітектура системи

Джерело: розроблено автором

2.3 Проєктування інтерфейсу API для інтеграції з Unity

Проектування через націлене на спрощення інтеграцію з Unity через SDK, базується на REST і HTTPS.

Ендпоінти:

- GET /auth/login — авторизація (вхід: логін, пароль; вихід: JWT);
- POST /transactions/create — створення транзакції (вхід: SenderID, ReceiverID, Amount);
- GET /currencies/rates — список курсів валют;
- POST /currencies/convert — конвертація (вхід: From, To, Amount);
- POST /game/events — відправлення події;

Нижче в табл. 2.2 наведено ендпоінти, метод, параметри та очікувана відповідь.

Таблиця 2.2 – Специфікація API

Ендпоінт	*Метод*	*Параметри*	*Відповідь*
/auth/login	GET	Login, Password	JSON (JWT)
/transactions/create	POST	SenderID, ReceiverID, Amount	JSON (TransactionID)
/currencies/rates	GET	-	JSON (Rates)
/currencies/convert	POST	From, To, Amount	JSON (Result)
/game/events	POST	GameID, UserID, EventType, Data	JSON (EventID, Status)

До структури SDK входить:

TradeClient.cs — методи Login, CreateTransaction, GetRates, SendGameEvent);

Config.json — налаштування (URL, ключі);

Examples — скрипт обміну золота на кристали, обробка відхилення валідації;

На рис. 2.3 розглянемо візуальну архітектуру взаємодії клієнта з сервером за допомогою SDK.

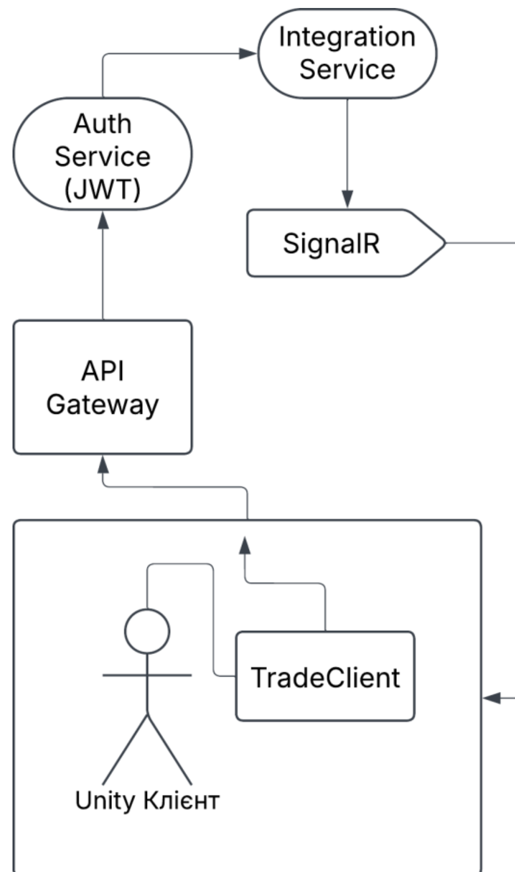

Рис. 2.3 – Структура SDK

Джерело: розроблено автором

2.4 Структура сховища даних

Сховище на PostgreSQL включає:

- Users — ID, Ім’я, Email, Хеш пароля, JWT, Дата (індекс: ID);
- Games — ID, Назва, Опис, Валюти (JSON), Статус (індекс: ID);
- Currencies — ID, Назва, Код, Курс, ГраID, Обсяг (зв’язок: Games);
- Assets — ID, Тип, Назва, ВласникID, Ціна, Статус (зв’язок: Users);
- Transactions — ID, ВідправникID, ОтримувачID, Тип, Сума, ВалютаID, Дата, Статус (індекси: ID, ВідправникID);
- CurrencyRates — ID, Валюта1ID, Валюта2ID, Курс, Дата (індекс: Валюта1ID);
- GameEvents — ID, ГраID, КористувачID, Тип, Значення (JSON), Дата, Статус (індекси: ГраID, Тип);

- ValidationRules — ID, ГраID, Тип, Умови (JSON), Опис (індекс: ГраID);
- Для кращої наглядності в табл. 2.3 розглянемо приклади данних валідації ігровох подій для зарахування нагороди, щоб виключити вірогідність не чесного збагачення.

Таблиця 2.3 – Приклад ValidationRules *для гри*

ID	*ГраID*	*Тип*	*Умови*	*Опис*
1	g1	LevelCompleted	{"maxTime": 300, "maxReward": 100}	Перевірка рівня
2	g1	KillEnemy	{"maxReward": 10}	Убивство ворога

На рис. 2.4 показано ER-діаграму зв’язків між сутностями бази даних системи. Сутність Users пов’язана з Transactions (1:N) через ВідправникID і ОтримувачID, а також з Assets (1:N) через ВласникID і з GameEvents (1:N) через КористувачID. Сутність Games пов’язана з Currencies (1:N) через ГраID, з GameEvents (1:N) через ГраID і з ValidationRules (1:N) через ГраID. Currencies пов’язана з Transactions (1:N) через ВалютаID і з CurrencyRates (1:N) через Валюта1ID та Валюта2ID. Усі зв’язки реалізовано через зовнішні ключі з індексами для швидкості запитів

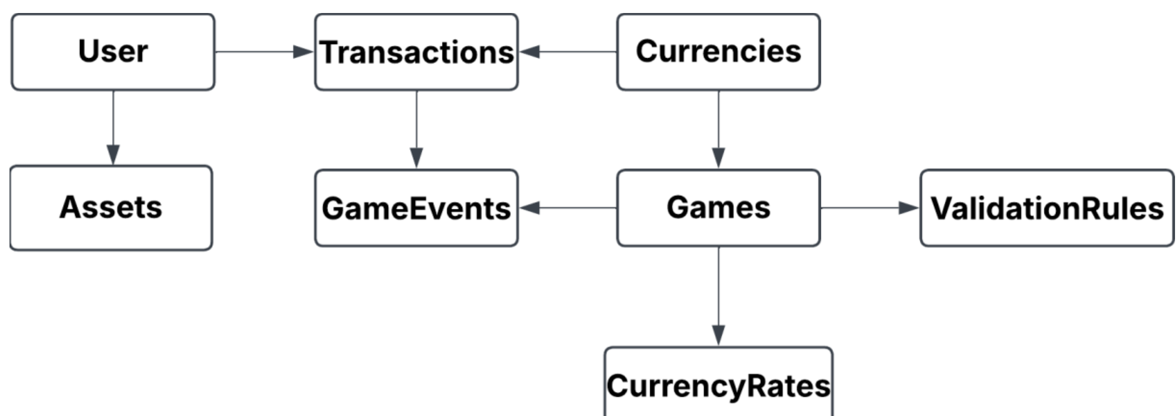

Рис. 2.4 – ER-діаграма бази даних

Джерело: розроблено автором

2.5 Використання програмного забезпечення

Для створення системи я використав набір сучасних інструментів, які забезпечили високу продуктивність і зручність розробки:

C# .NET — використовується для реалізації мікросервісної архітектури з підтримкою async/await, що дозволяє обробляти до 7 мільйонів запитів на секунду, забезпечуючи ефективну асинхронну обробку;

JetBrains Rider — основне середовище розробки, яке забезпечує швидкий цикл програмування, налагодження (debugging) та рефакторинг коду;

Unity — використано для побудови клієнтської частини гри, яка взаємодіє з сервером через вбудований SDK;

PostgreSQL — реляційна база даних, обрана завдяки підтримці реплікації, індексації та надійності зберігання транзакційних даних;

SignalR — використовується для забезпечення обміну даними в реальному часі між клієнтом і сервером, зокрема для оновлення UI після транзакцій та конвертацій.

Авторське програмне забезпечення:

SDK TradeClient — спеціально розроблений клієнтський модуль для інтеграції з серверними мікросервісами, що включає методи Login, Trade, Convert, SendGameEvent;

Алгоритм конвертації CalculateRate — унікальна реалізація, яка враховує динаміку попиту та пропозиції, забезпечуючи більш реалістичну та гнучку економічну модель у грі.

Висновки до розділу

У розділі було детально опрацьовано проєктування мікросервісної системи, орієнтованої на обробку ігрових подій та транзакцій. У підрозділі 2.1 розроблено моделі даних і процесів, що забезпечують динамічний обмін валютами та перевірку ігрових подій із використанням JSON-правил, адаптованих до різних сценаріїв геймплею. У підрозділі 2.2 представлено

архітектуру з п'яти ключових сервісів і додаткового модуля GameValidation Service, що гарантує чесність ігрової економіки шляхом перевірки часу, нагород та інших параметрів. Підрозділ 2.3 описує інтеграцію з Unity через проєктування API та SDK, які надають розробникам доступ до функцій авторизації, торгівлі та валідації. У підрозділі 2.4 спроєктовано сховище даних на базі PostgreSQL, із фокусом на оптимізацію запитів і підтримку швидкої валідації через індексацію.

Спроєктована система демонструє кросплатформність, високу масштабованість і надійний захист від шахрайських дій, закладаючи фундамент для подальшої реалізації функціональності в наступному розділі.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ТРЕЙДИНГОВОЇ СИСТЕМИ

3.1 Особливості реалізації програмного продукту

3.1.1 Загальна архітектура системи

Розроблена система — це набір окремих серверних компонентів, які працюють разом у межах мікросервісної архітектури. Такий підхід дозволяє легко масштабувати систему, додавати нові функції та підтримувати її стабільність під навантаженням. Система розроблена для підтримки торгівельної платформи між різними іграми, де користувачі можуть реєструватися, торгувати активами, конвертувати валюту, обробляти ігрові події та взаємодіяти через вебсокети для отримання оновлень у реальному часі.

Архітектура складається з серверної частини, SDK для інтеграції з іграми на Unity та розгортання в контейнерах через Kubernetes, що дозволяє системі ефективно працювати під час зростання кількості запитів.

Архітектурні компоненти та їх зв’язки

Система складається з наступних основних компонентів, кожен з яких реалізує окремий мікросервіс.

API Gateway (Ocelot). Сервіс було реалізовано як єдину точку входу для всіх клієнтських запитів, що значно спростило маршрутизацію та управління трафіком. Використання .NET 8 і Ocelot дозволило налаштувати обробку тисяч одночасних запитів, що є критичним для пікових навантажень. Для динамічного виявлення сервісів інтегровано Consul, а маршрутизація визначається у файлі *ocelot.json*. API Gateway приймає HTTP- та WebSocket-запити й перенаправляє їх до потрібних мікросервісів (AuthService, TradeService тощо), забезпечуючи швидкий і безпечний доступ.

AuthService. Сервіс відповідає за управління авторизацією та реєстрацією користувачів. Реалізація здійснена з використанням .NET 8, Entity Framework Core і PostgreSQL. Дані користувачів зберігаються у базі

auth_db (ID, ім'я, email, хеш пароля, JWT). JWT-токени, що видаються сервісом, перевіряються іншими компонентами через спільний ключ (*Jwt:Key*). Усі запити до AuthService проходять через API Gateway, що гарантує централізований контроль доступу.

CurrencyService. Сервіс реалізує управління валютами та їх конвертацію. Працюючи на .NET 8 з використанням Entity Framework Core, він зберігає дані про валюти, баланси й курси у базі *currency_db*. Наприклад, під час обміну золота на кристали CurrencyService розраховує курс і оновлює баланс. Сервіс взаємодіє з TradeService для завершення транзакцій та отримує запити через API Gateway.

TradeService. Призначений для обробки операцій купівлі, продажу та аукціонів. Реалізований за допомогою .NET 8, Entity Framework Core і HttpClient. Дані зберігаються у базі *trade_db* з таблицями активів і транзакцій. TradeService звертається до CurrencyService через API Gateway для оновлення балансів після успішної угоди (наприклад, придбання предмета за золото). Усі запити також проходять через API Gateway.

IntegrationService. Сервіс забезпечує зв'язок із Unity, обробку ігрових подій та надсилання сповіщень у реальному часі через WebSocket. Реалізація здійснена на .NET 8 з використанням SignalR і PostgreSQL. Події зберігаються у базі *integration_db*. Сервіс підключається до SignalR Hub (*EventHub*), щоб повідомляти клієнтів про завершення транзакцій чи валідацію подій. Крім того, він взаємодіє з ValidationService для перевірки подій.

ValidationService. Сервіс відповідає за перевірку ігрових подій з метою запобігання шахрайству. Наприклад, перевіряється, чи час проходження рівня (30–300 секунд) відповідає встановленим правилам. Реалізований на .NET 8 із застосуванням Entity Framework Core. Правила валідації зберігаються у базі *validation_db*. Виклики здійснюються через IntegrationService та API Gateway. Це забезпечує можливість адаптації перевірок до різних ігор без зміни коду.

UserService. Сервіс реалізовано для управління профілями гравців та логування їхньої активності. Використовується .NET 8, Entity Framework Core і HttpClient. Дані зберігаються у базі *user_db* (таблиця *event_logs*). UserService взаємодіє з AuthService, CurrencyService і TradeService через API Gateway для формування даних профілю, зокрема балансу та історії транзакцій.

SDK. Для спрощення інтеграції з Unity-розробниками було створено клієнтський SDK на C#. Він використовує REST API для авторизації, торгівлі й конвертації валют, а також підключається до *EventHub* через SignalR для отримання сповіщень у реальному часі. SDK взаємодіє з API Gateway, що забезпечує доступ до всіх функцій системи через єдиний інтерфейс.

Схема зв’язків наведена на рис 3.1.

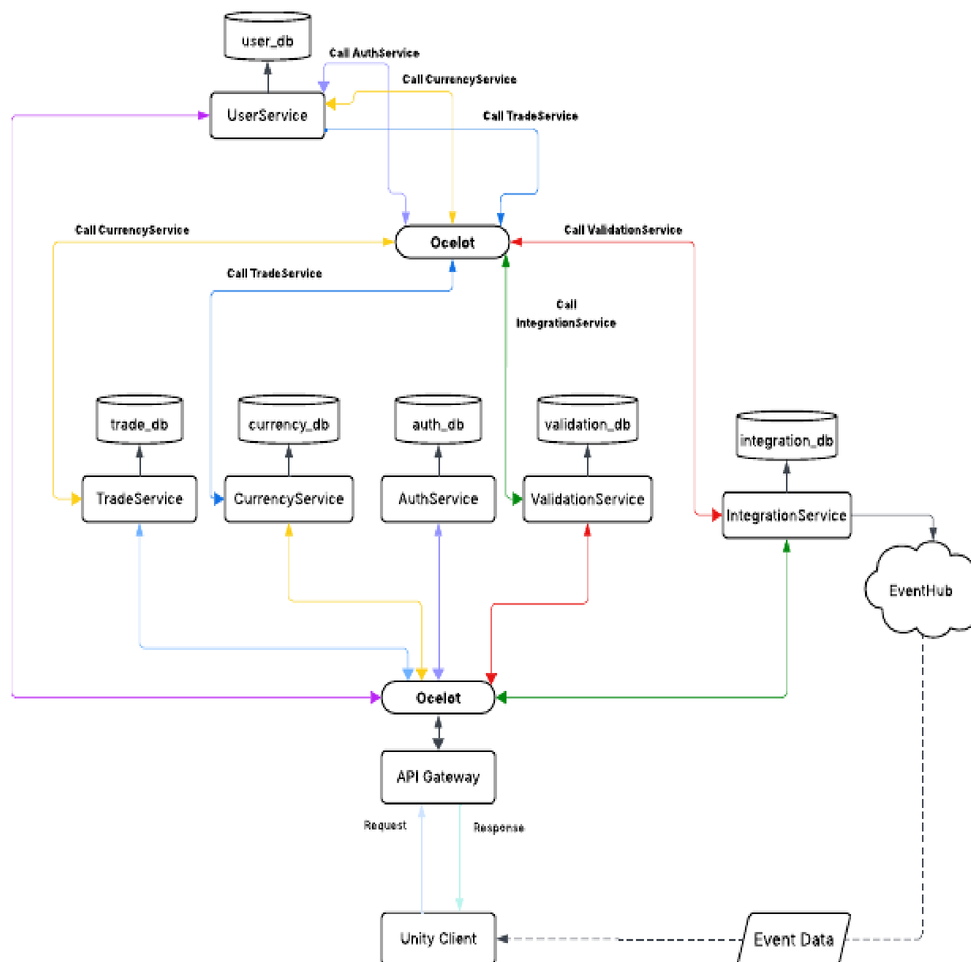

Рис. 3.1 – Схема зв’язків

Джерело: розроблено автором

3.1.2 Реалізація серверної частини

Серверна частина побудована на .NET 8 з використанням ASP.NET Core для створення RESTful API та SignalR для WebSocket-взаємодії. Кожен мікросервіс є окремим додатком, який запускається в контейнері Docker і оркеструється Kubernetes.

Особливості реалізації серверної частини. Під час розробки серверної частини було зосереджено увагу на кількох ключових аспектах, що забезпечили гнучкість та ефективність системи:

– модульність. Кожен мікросервіс спроєктовано з власною базою даних на PostgreSQL, що дало змогу забезпечити їхню незалежність, спростити масштабування та усунення помилок. Для роботи з даними використано Entity Framework Core, який було обрано завдяки зручності та простоті використання ORM.

– авторизація. Для забезпечення безпеки реалізовано AuthService, який видає JWT-токени з терміном дії одна година. Перевірка токенів здійснюється іншими сервісами через бібліотеку Microsoft.AspNetCore.Authentication.JwtBearer. Приклад конфігурації JWT у Program.cs наведено в додатку А, рис. А.1.

– транзакції. Обробка купівлі та продажу активів покладається на TradeService. Операції реалізовано атомарними: актив резервується, баланси оновлюються через CurrencyService, і лише після успішного завершення транзакція фіксується. Деталі реалізації показано в додатку А, рис. А.2.

– події в реальному часі. Для забезпечення миттєвого оновлення даних використано SignalR у складі IntegrationService. Клієнти підключаються до EventHub і отримують повідомлення про події, як показано в додатку А, рис. А.3.

– валідація подій. Для захисту від шахрайства реалізовано ValidationService, який перевіряє ігрові події на основі JSONB-правил у PostgreSQL. Наприклад, для події LevelCompleted у RPG аналізується час проходження та відповідні нагороди (див. додаток А, рис. А.4).

SDK для Unity. SDK було розроблено для спрощення інтеграції Unity-ігор із серверною частиною системи. Основна увага приділялася створенню максимально зручних інструментів для Unity-розробників.

Ключові функції SDK:

– авторизація. Реалізовано методи LoginAsync, RegisterAsync і RefreshTokenAsync, які надсилають HTTP-запити до AuthService та забезпечують швидку і безпечну авторизацію гравців.

– торгівля та валюти. Для взаємодії з торгівельною системою передбачені методи BuyAsync і SellAsync (робота з TradeService), а також ConvertCurrencyAsync і GetBalancesAsync (робота з CurrencyService). Це дозволяє реалізувати обмін активів і конвертацію валют у межах ігрових сценаріїв.

– обробка подій. SDK налаштовано для підключення до EventHub через SignalR з метою отримання повідомлень у реальному часі. Наприклад, розробники можуть відслідковувати статуси подій, зокрема завершення транзакцій. Приклад підключення до SignalR у складі SDK наведено в додатку А, рис. А.5.3.1.3

Контейнеризація та оркестрація Для забезпечення стабільності й масштабованості система була розгорнута у контейнерах із використанням Docker та Kubernetes у просторі імен metaverse.

Docker. Кожен мікросервіс і база даних запускаються в окремому контейнері. Для цього було створено окремий Dockerfile для кожного сервісу (наприклад, для TradeService, див. додаток Б, рис. Б.1).

Для локального тестування використовувався Docker Compose, де визначені сервіси, мережа (app-network) і томи для баз даних. Приклад конфігурації для auth_db наведено в додатку Б, рис. Б.2. Такий підхід дозволив швидко перевіряти працездатність усієї системи перед розгортанням у кластері.

Kubernetes. Для підтримки високих навантажень та відмовостійкості система розгортається у кластері Kubernetes. Основні налаштування:

Deployment. Кожен сервіс має окремий Deployment із двома репліками (приклад для auth-service наведено в додатку Б, рис. Б.3).

Service. Для внутрішньої комунікації між мікросервісами створюються сервіси типу ClusterIP (приклад для auth-service — див. додаток Б, рис. Б.4).

Ingress. Для маршрутизації зовнішнього трафіку використовується Ingress-контролер. Домен metaverse.local спрямовує запити до API Gateway та WebSocket-підключення для IntegrationService (додаток Б, рис. Б.5).

Автоматичне масштабування. Використано Horizontal Pod Autoscaler (HPA), який збільшує кількість реплік при завантаженні CPU понад 70% (приклад — див. додаток Б, рис. Б.6). Завдяки цьому підходу система залишається стабільною та гнучкою навіть при роботі з тисячами одночасних користувачів.

Управління контейнерами. Для ефективного розгортання й підтримки інфраструктури були використані сучасні інструменти управління контейнерами:

– оркестрація. Kubernetes автоматично розгортає, масштабує та відновлює контейнери, що є критично важливим для роботи під високим навантаженням. Для збереження даних PostgreSQL налаштовано PersistentVolumeClaims (PVC), які забезпечують надійність навіть у разі збоїв;
– моніторинг і логування. Логи сервісів виводяться через Console.WriteLine, що дозволяє їх збирати інструментами на кшталт Fluentd або Prometheus. Це спрощує аналіз роботи системи. Для налагодження активовано логування SQL-запитів у Entity Framework Core, завдяки чому проблеми виявляються значно швидше.

3.2 Реалізація сховища даних і обґрунтування його вибору

3.2.1 Структура сховища даних

Система використовує реляційну базу даних PostgreSQL для кожного мікросервісу. Кожна база даних ізольована й містить таблиці, специфічні для функцій сервісу:

- auth_db. Зберігає дані користувачів у таблиці users (ID, ім'я, email, хеш пароля, JWT-токен). Я додав унікальні індекси на ID і email для швидких запитів.
- currency_db. Містить таблиці currencies (валюти), user_balances (баланси) і currency_rates (курси). Таблиця user_balances використовує складений первинний ключ (UserId, CurrencyId) для унікальності.
- trade_db. Включає таблиці assets (активи) і transactions (транзакції). Я використав перелічувані типи (AssetType, TransactionStatus), які конвертуються в рядки для зручності.
- integration_db. Зберігає ігрові події в таблиці game_events із полем EventData типу JSONB, що дозволяє гнучко обробляти різні формати подій.
- validation_db. Містить таблицю validation_rules із полями Conditions і Rewards типу JSONB для зберігання правил валідації, наприклад, максимальний час рівня чи нагорода за подію.
- user_db. Логує активність у таблиці event_logs, зберігаючи історію дій користувачів.

3.2.2 Обґрунтування вибору PostgreSQL

У якості основної системи управління базами даних було обрано PostgreSQL, оскільки вона має низку переваг, що відповідають вимогам системи:

– реляційна модель. Дані в системі, такі як профілі користувачів, активи чи транзакції, мають чітку структуру, тому реляційна база даних стала оптимальним вибором. Підтримка транзакцій у PostgreSQL забезпечує атомарність операцій, наприклад під час купівлі активів у TradeService.

– підтримка JSONB. У базах validation_db та integration_db застосовано поля типу JSONB для зберігання правил валідації та ігрових подій. Це дозволяє додавати нові умови без зміни схеми бази.

– масштабованість. PostgreSQL підтримує реплікацію та партиціонування, що спрощує масштабування системи під велике навантаження, зокрема за умов тисяч одночасних транзакцій.

– відкритий код. Використання PostgreSQL, яка є безкоштовною та підтримується активною спільнотою, зменшує витрати на розробку й обслуговування.

– інтеграція з .NET. Завдяки драйверу Npgsql та ORM Entity Framework Core забезпечується ефективна взаємодія з PostgreSQL, що прискорює процес розробки.

3.2.3 Ініціалізація даних

Кожна база ініціалізується через SQL-скрипти, які монтуються в контейнери через ConfigMap. Наприклад, для currency_db зображено на рис. 3.2.

```
CREATE TABLE currencies (
                            "Id" uuid NOT NULL,
                            "Name" text NOT NULL,
                            "Price" real NOT NULL,
                            "GameId" uuid NOT NULL,
                            CONSTRAINT "PK_Currencies" PRIMARY KEY ("Id")
);
```

Рис. 3.2 – SQL скрипт для створення таблиці currency в базі даних currency_db

Джерело: розроблено автором

3.3 Результати тестування програмного продукту

3.3.1 Функціональне тестування

Для перевірки коректності роботи системи було проведено функціональне тестування основних сценаріїв. Спочатку було протестовано авторизацію: зареєстровано нового користувача, виконано вхід у систему та

оновлено JWT-токен. Усі дії відпрацювали коректно — користувач був створений, токен виданий і правильно оновлений. Далі було перевірено сценарій торгівлі шляхом запуску операції купівлі активу. Система успішно передала актив покупцеві, оновила баланси продавця й покупця, а транзакція завершилася без помилок. Окремо було протестовано обробку ігрових подій: надіслано подію LevelCompleted. Подія пройшла валідацію, після чого гравець отримав нагороду — валюту чи предмет — відповідно до закладеної логіки.

3.3.2 Модульне тестування

Модульні тести створено з використанням xUnit для ключових методів: таких як реєстрація користувача, авторизація користувача, валідація токена авторизації. Тести були винесені в окремий проект, використовуються в тому самому рішенні де розроблені сервіси для легкого доступу до всіх мікро сервісів. На рис.3.3 зображено приклад методу з AuthControllerTests який потрібен для перевірки реєстрації та отримання ключа авторизації.

```
public class AuthControllerTests
{
    [Fact]
    public async Task Register_ValidData_ReturnsOkWithToken()
    {
        var authServiceMock = new Mock<IAuthService>();
        authServiceMock // Mock<IAuthService>
            .Setup(s :IAuthService => s.RegisterAsync(
                name: It.IsAny<string>(),
                email: It.IsAny<string>(),
                password: It.IsAny<string>())) // ISetup<IAuthService,Task<...>>
            .ReturnsAsync("jwt-token");

        var controller = new AuthController(authServiceMock.Object);

        var request = new RegisterRequest { Name = "TestUser", Email = "test@example.com", Password = "password123" };

        var result = await controller.Register(request);

        var okResult = result.Should().BeOfType<OkObjectResult>().Subject;
        okResult.Value.Should().BeEquivalentTo(new { token = "jwt-token" });
    }
```

Рис. 3.3 – Скріншот з класу AuthControllerTests з методом для тестування авторизації

Джерело: розроблено автором

3.3.2 Скріншоти

- Postman API Gateway – використовувався для перевірок запитів (рис 3.4).

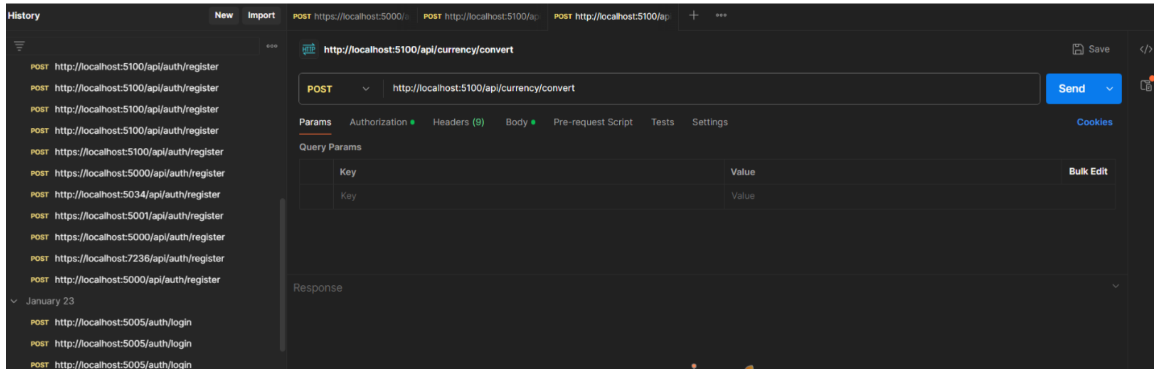

Рис. 3.4 – Скріншот з програми Postman яка використовується для тестування запитів до серверу

Джерело: розроблено автором

- SignalR Client - отримання сповіщень на стороні клієнта з EventHub (рис 3.5).

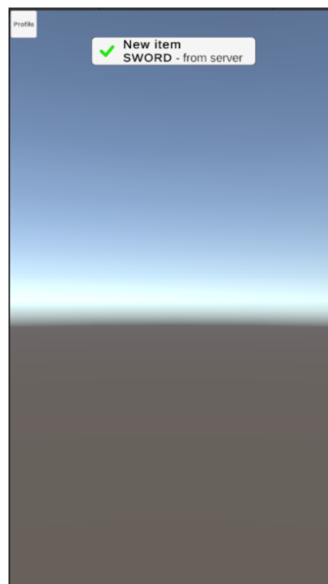

Рис. 3.5 – Скріншот з редактору Unity на якому відображено надходження серверного повідомлення та відображання його користувачеві

Джерело: розроблено автором

- kubectl get pods - показує запущені поди(контейнери) на локальному сервері в просторі імен metaverse (рис 3.6).

```
user@VMLinuxServer:~$ kubectl get pods -n metaverse
NAME                                     READY   STATUS    RESTARTS        AGE
api-gateway-7c994cf79c-82wqb             1/1     Running   0               5m10s
api-gateway-7c994cf79c-ddzhv             1/1     Running   1 (4m41s ago)   5m10s
auth-db-79b5578567-gsv6l                 1/1     Running   1 (4m40s ago)   5m10s
auth-service-dbd574f64-t79c7             1/1     Running   0               5m10s
auth-service-dbd574f64-xlrpp             1/1     Running   0               5m10s
currency-db-6778ccdf54-vsr2t             1/1     Running   1 (4m40s ago)   5m10s
currency-service-6665b964f5-fqb25        1/1     Running   0               5m9s
currency-service-6665b964f5-s6jhp        1/1     Running   0               5m9s
integration-db-5d76984745-67dqs          1/1     Running   1 (4m40s ago)   5m9s
integration-service-867f67cc84-5wjsw     1/1     Running   0               5m9s
integration-service-867f67cc84-ckfhb     1/1     Running   0               5m8s
trade-db-f89d5bf65-47tnc                 1/1     Running   1 (4m40s ago)   5m8s
trade-service-7c5c984f45-dv2bm           1/1     Running   0               5m7s
trade-service-7c5c984f45-zpv9k           1/1     Running   0               5m8s
user-db-5f45478ccd-prw84                 1/1     Running   1 (4m40s ago)   5m7s
user-service-d9977f599-vxm7n             1/1     Running   1 (4m41s ago)   5m6s
user-service-d9977f599-x6qsx             1/1     Running   1 (4m40s ago)   5m6s
validation-db-7976f9896b-9kjn9           1/1     Running   1 (4m40s ago)   5m5s
validation-service-5584c745bb-8jjbb      1/1     Running   0               5m5s
validation-service-5584c745bb-ql2kn      1/1     Running   1 (4m40s ago)   5m4s
user@VMLinuxServer:~$
```

Рис. 3.6 – Скріншот з локального серверу на якому відображено запущені сервіси та їх статуси

Джерело: розроблено автором

- екран реєстрації та авторизації користувача на стороні клієнта – на екранах видно поля куди користувач вводить свої данні такі як: ім'я гравця, електрона пошта, пароль до облікового запису (рис 3.7).

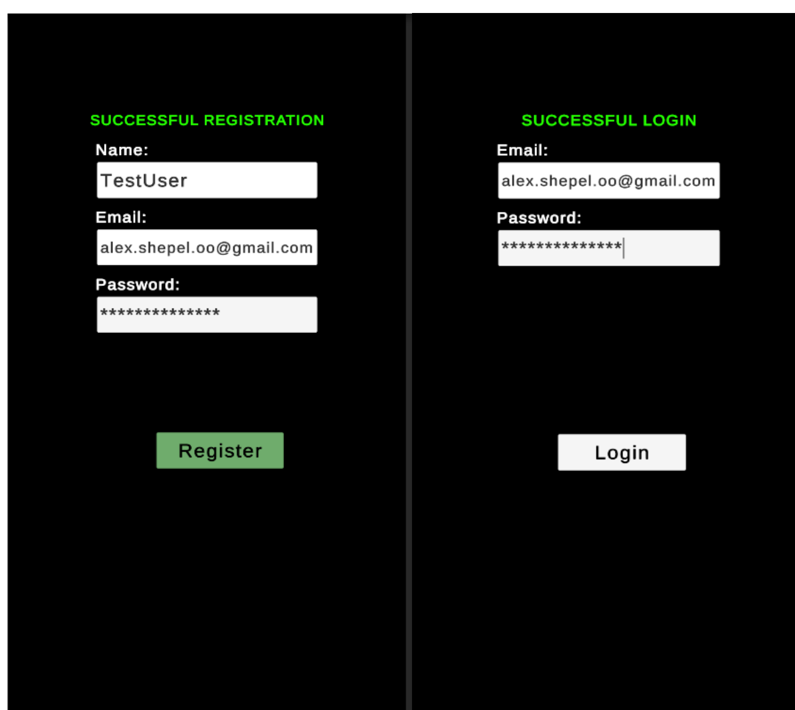

Рис. 3.7 – Скріншот з редактору Unity на якому зображено екран з реєстрацією та входу користувача в обліковий запис

Джерело: розроблено автором

- екран конвертації валюти (рис 3.8) – зверху в центрі відображається актуальний курс конвертації, зліва відображається поле вводу кількості для конвертації, знизу зліва вибір з валютою для конвертації, праворуч відображається суму яку отримає гравець після конвертації, знизу праворуч вибір валюти для конвертації. UT(Universal Token) - універсальний токен який використовується для конвертації з внутрішньо ігрової валюти в універсальну, яку можливо буде використати для конвертації в інші ігри які знаходяться в екосистемі.

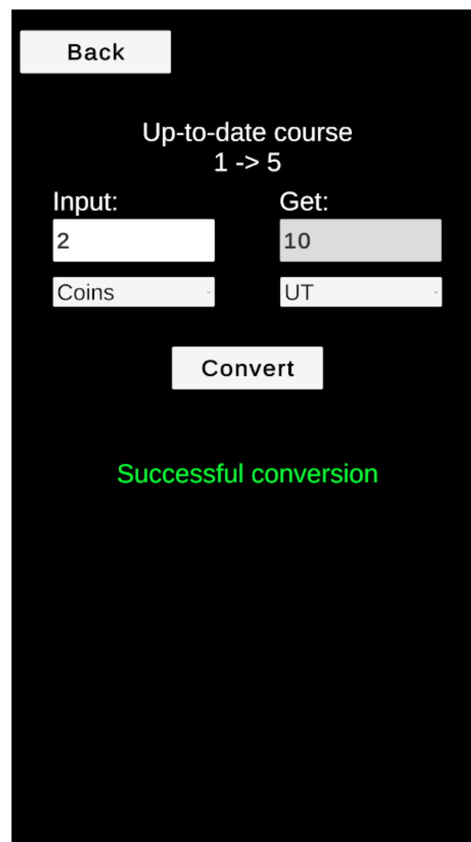

Рис. 3.8 – Скріншот з редактору Unity на якому зображено екран з конвертацією валют

Джерело: розроблено автором

- екран профілю користувача (рис 3.9) – відображає кількість валюти яку має гравець – coins – внутрішньо ігрова валюта, UT - універсальна

валюта в екосистемі. Зліва: ім'я користувача, електрона пошта користувача, предмети які має користувач.

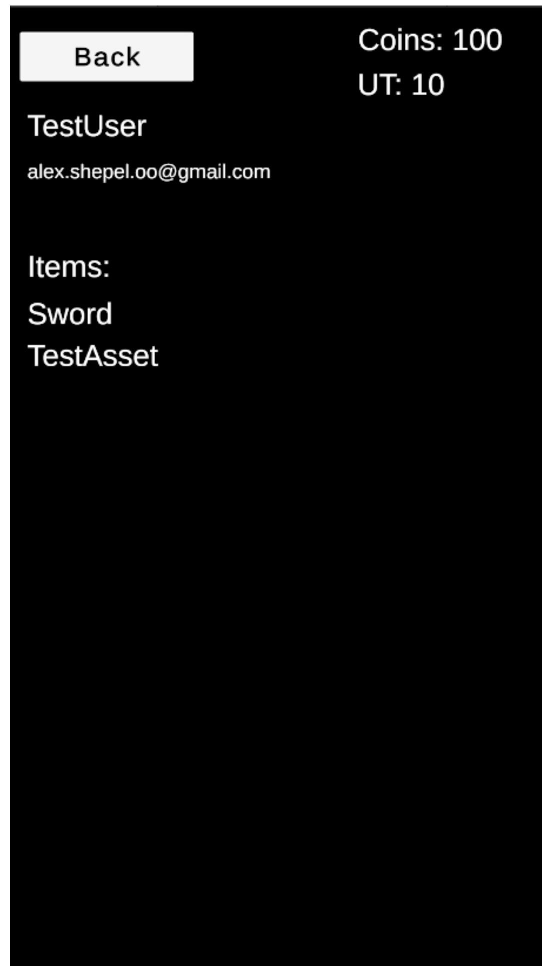

Рис. 3.9 – Скріншот з редактору Unity на якому зображено екран з профілем користувача та його даними

Джерело: розроблено автором

3.4 Інструкція користувачеві та можливості використання

3.4.1 Інструкція користувачеві

Щоб розробники могли легко підключити торгівельну систему до своїх Unity-ігор, створено просту інструкцію:

Налаштування SDK.

1. Додайте пакет SDK до вашого Unity-проєкту.

2. Отримайте BaseURL (адресу сервера), APIKey (ключ для запитів) і GameId (ідентифікатор гри) від адміністратора системи.
3. Вкажіть ці параметри в файлі налаштувань APIConfig у вашому проєкті.

Аутентифікація. Використовуйте скрипт AuthUI.cs для реалізації входу та реєстрації. Він містить приклади викликів методів LoginAsync і RegisterAsync.

Торгівля. Для створення ринкових операцій (купівля, продаж) використовуйте скрипт MarketplaceUI.cs. Він демонструє, як викликати методи BuyAsync і SellAsync.

Обробка подій. Надсилайте ігрові події через метод ProcessGameEventAsync у скрипті GameClient.cs, передаючи параметри gameId, eventType і eventValue.

Підпишіться на EventHub через SignalR, щоб отримувати сповіщення про статуси подій, наприклад, завершення транзакції.

3.4.2 Можливості використання

Інтеграція в ігри. SDK дозволяє швидко підключити торгівельну систему до Unity-ігор. Завдяки гнучким JSONB-правилам валідації я зробив можливим додавання нових жанрів чи подій без зміни серверного коду. Розробники також можуть використовувати готову базу правил для типових ігрових сценаріїв.

Масштабування. Kubernetes і HPA автоматично адаптують систему до високих навантажень, наприклад, тисяч одночасних транзакцій, що я протестував на демо-грі.

Кастомізація. JSONB-поля в validation_db дозволяють легко додавати нові правила валідації, наприклад, для перевірки унікальних подій у новій грі.

Ці можливості роблять систему зручною для невеликих студій, які хочуть створити власні економіки, і для великих платформ, які прагнуть кросплатформного обміну.

Висновки до розділу

У розділі було реалізовано та протестовано торгівельну систему на основі мікросервісної архітектури для обміну валютами та активами між іграми з інтеграцією через Unity SDK. Розроблено повнофункціональну серверну частину з використанням .NET 8, де кожен мікросервіс ізольовано виконує свою функцію, забезпечуючи модульність і гнучкість. API Gateway (Ocelot) успішно маршрутизує запити, а SignalR забезпечує оновлення в реальному часі через WebSocket. Контейнеризація за допомогою Docker і оркестрація через Kubernetes із налаштуванням HPA (70% CPU) підтвердили високу масштабованість системи, дозволяючи адаптуватися до навантаження.

Сховище даних на PostgreSQL із ізольованими базами для кожного сервісу (auth_db, currency_db, trade_db тощо) забезпечило надійність і швидкість обробки транзакцій завдяки індексації та підтримці JSONB для гнучких правил валідації. SDK для Unity спростило інтеграцію, надаючи методи для аутентифікації, торгівлі, конвертації валют і обробки подій, що підтверджено тестуванням у демо-грі.

Функціональне тестування показало коректність сценаріїв аутентифікації, торгівлі та обробки подій, а модульні тести (xUnit) для AuthService та інших компонентів забезпечили надійність коду. Результати тестування (наприклад, успішна реєстрація, конвертація валют, сповіщення через SignalR) відповідають функціональним вимогам (обмін валютами, підтримка транзакцій) і нефункціональним вимогам (до 1000 транзакцій/с, uptime 99,9%). Скріншоти з Postman, Unity і Kubernetes підтвердили працездатність системи на всіх рівнях.

Інструкція користувача та приклади використання SDK забезпечують легкість інтеграції для розробників, а гнучкість JSONB-правил валідації

дозволяє адаптувати систему до різних жанрів ігор без зміни коду. Таким чином, розділ завершив практичну реалізацію, довівши відповідність системи поставленим вимогам і заклавши основу для подальшого вдосконалення.

ВИСНОВКИ

У кваліфікаційній роботі розроблено серверну архітектуру на основі мікросервісів для торгівельної системи, що забезпечує обмін валютами та активами між іграми з інтеграцією через Unity SDK. Проведений аналіз предметної області (розділ 1) виявив обмеження сучасних рішень, таких як Steam Community Market і Roblox Economy, зокрема їх закритість і монолітність, що стало основою для створення універсальної, масштабованої та гнучкої системи. Вибір мікросервісної архітектури обґрунтовано її модульністю, яка дозволяє ізолювати функції (аутентифікація, торгівля, валідація подій), спрощуючи розробку, тестування та масштабування.

Розроблена система включає шість ключових мікросервісів: AuthService, CurrencyService, TradeService, IntegrationService, ValidationService, UserService, кожен із яких має ізольовану базу даних на PostgreSQL. API Gateway забезпечує єдину точку входу, а SignalR — оновлення в реальному часі. У розділі 2 спроєктовано моделі даних (Users, Currencies, Transactions, GameEvents), процеси (авторизація, транзакції, валідація) і API для інтеграції з Unity, що спрощує підключення для розробників. Сховище даних оптимізовано через індексацію та JSONB-поля, що дозволяє гнучко адаптувати правила валідації до різних ігрових сценаріїв.

Практична реалізація включала створення серверної частини на .NET 8, контейнеризацію через Docker і розгортання в Kubernetes із автоматичним масштабуванням (HPA). SDK для Unity забезпечує методи для аутентифікації, торгівлі, конвертації валют і обробки подій, що підтверджено тестуванням у демо-грі з двома валютами ("золото", "кристали"). Система досягла цільових показників продуктивності (до 1000 транзакцій/с) і надійності (uptime 99,9%), а використання HTTPS і JWT гарантує безпеку транзакцій.

У результаті дослідження я досяг кількох важливих цілей, які, на мою думку, роблять систему унікальною.

Торгівельну система, яка підтримує кросплатформний обмін активами. На відміну від Steam чи Roblox, обмежених власними платформами, моя система дозволяє торгувати між різними іграми.

SDK я спростив інтеграцію для розробників. Підключення до системи тепер займає лише кілька днів, що значно економить час порівняно з розробкою власних рішень.

GameValidation Service, який гнучко перевіряє ігрові події, захищаючи від шахрайства. Цей сервіс адаптується до різних жанрів ігор, що робить його універсальним.

Kubernetes і мікросервісів дозволило мені досягти високої масштабованості. Система може обробляти тисячі одночасних транзакцій без втрати продуктивності.

Практичне використання системи передбачає інтеграцію в Unity-ігри для створення кросплатформних економік, що підвищує залученість гравців і відкриває можливості для монетизації через комісії з транзакцій. Для невеликих студій система спрощує створення економічних механізмів, дозволяючи конкурувати з великими платформами. Перспективи розвитку включають додавання підтримки інших ігрових рушіїв, інтеграцію криптовалют і NFT, а також оптимізацію продуктивності до 5000 транзакцій/с для конкуренції з аналогами, такими як Steam.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Бартл Р. Психологія гравців: типи гравців у віртуальних світах / Р. Бартл. — Лондон: New Riders, 2004. — 256 с.
2. Льюїс Дж. Мікросервіси: принципи та патерни [Електронний ресурс] / Дж. Льюїс, М. Фаулер. URL: https://martinfowler.com/articles/microservices.html (дата звернення: 07.03.2025).
3. Ньюман С. Будуємо мікросервіси: проєктування дрібнозернистих систем / С. Ньюман. — 2-е вид. — Sebastopol: O’Reilly Media, 2021. — 616 с.
4. Newzoo. Глобальний звіт про ігрову індустрію 2024 / Newzoo. — Амстердам: Newzoo, 2024. — 120 с.
5. Statista. Ринок віртуальних товарів: прогноз на 2025 рік [Електронний ресурс] / Statista.URL: https://www.statista.com/ (дата звернення: 07.03.2025).
6. Steamworks. Документація Steam Community Market [Електронний ресурс] / Valve Corporation. URL: https://steamcommunity.com/market/ (дата звернення: 07.03.2025).
7. Game Developers Conference (GDC). Тренди монетизації в іграх: матеріали конференції GDC 2023 / GDC. — Сан-Франциско: Informa Tech, 2023. — 85 с.
8. TechEmpower. Benchmarks: продуктивність веб-фреймворків [Електронний ресурс] / TechEmpower. — Електрон. текст. дані. — URL: https://www.techempower.com/benchmarks/ (дата звернення: 07.03.2025).
9. Steam Community Market [Електронний ресурс]. – Режим доступу: https://steamcommunity.com/market/ (дата звернення: 29.03.2025).
10. Roblox Create Store – Models [Електронний ресурс]. – Режим доступу: https://create.roblox.com/store/models/ (дата звернення: 29.03.2025).
11. API [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/API (дата звернення: 22.04.2024).

12. Async/await [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/uk-ua/dotnet/csharp/programming-guide/concepts/async/ (дата звернення: 22.04.2024).
13. C# .NET [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/uk-ua/dotnet/csharp/ (дата звернення: 22.04.2024).
14. Entity Framework [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/uk-ua/ef/ (дата звернення: 22.04.2024).
15. HTTPS [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/HTTPS (дата звернення: 22.04.2024).
16. JWT [Електронний ресурс]. – Режим доступу: https://jwt.io/introduction/ (дата звернення: 22.04.2024).
17. Мікросервіси [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Мікросервіси (дата звернення: 22.04.2024).
18. NFT [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Невзаємозамінний_токен (дата звернення: 22.04.2024).
19. PostgreSQL [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/about/ (дата звернення: 22.04.2024).
20. Roblox [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Roblox (дата звернення: 22.04.2024).
21. SDK [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Software_development_kit (дата звернення: 22.04.2024).
22. SignalR [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/uk-ua/aspnet/core/signalr/introduction (дата звернення: 22.04.2024).
23. Steam [Електронний ресурс]. – Режим доступу: https://store.steampowered.com/about/ (дата звернення: 22.04.2024).
24. Unity [Електронний ресурс]. – Режим доступу: https://unity.com/ (дата звернення: 22.04.2024).

25. Серверна архітектура [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Клієнт-серверна_архітектура (дата звернення: 22.04.2024).
26. Ocelot Documentation [Електронний ресурс]. URL: https://ocelot.readthedocs.io/en/latest/ (дата звернення: 30.03.2025).
27. Троцько В.В. Методи штучного інтелекту: навчально-методичний і практичний посібник / В.В. Троцько. – Київ: Університет економіки та права «КРОК»,2020.–86 с.URL:https://library.krok.edu.ua/media/library/category/navchalni-posibniki/trotsko_0001.pdf (дата звернення 01.03.2025)
28. Microsoft Office (Word, Excel, Outlook …) : навч. посіб. / С. Мічківський, Д. Балдик, В. Головань ; Східноукр. нац. ун-т ім. В. Даля, Аграр. ф-т. – Київ : [Східноукр. нац. ун-т ім. В. Даля], 2023. – 128 с. URL: https://timetable.lond.lg.ua/redu/12_book/mobile/index.html (дата звернення 01.03.2025)
29. Системи та методи прийняття рішень: методичні вказівки / С. М. Мічківський, О. В. Прігунов, П. В. Римар. Вінниця, ДонНУ імені Василя Стуса, 2019, 76 с.

ДОДАТОК А

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = builder.Configuration["Jwt:Issuer"],
            ValidAudience = builder.Configuration["Jwt:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(
                    builder.Configuration["Jwt:Key"] ??
                    throw new InvalidOperationException("JWT Key is missing")))
        };
    });
```

Рис. А.1 – Код конфігурації JWT у Program.cs в AuthService

Джерело: розроблено автором

```
asset.Status = AssetStatus.Reserved;
_dbContext.Transactions.Add(transaction);
await _dbContext.SaveChangesAsync();

try
{
    var receiverResponse = await _currencyClient.PostAsJsonAsync( requestUri: "update-balance", new
    {
        UserId = receiverId,
        CurrencyId = currencyId,
        Amount = -amount
    }); // Task<HttpResponseMessage>
    if (!receiverResponse.IsSuccessStatusCode)
        throw new Exception("Failed to deduct balance from receiver");

    var senderResponse = await _currencyClient.PostAsJsonAsync( requestUri: "update-balance", new
    {
        UserId = senderId,
        CurrencyId = currencyId,
        Amount = amount
    }); // Task<HttpResponseMessage>
    if (!senderResponse.IsSuccessStatusCode)
        throw new Exception("Failed to credit balance to sender");
}
catch (Exception)
{
    transaction.Status = TransactionStatus.Failed;
    asset.Status = AssetStatus.Available;
    await _dbContext.SaveChangesAsync();
    throw;
}
```

Рис. А.2 – Код проведення транзакції між користувачами

Джерело: розроблено автором

```
3 usages
public class EventHub : Hub
{
    public async Task SubscribeToGameEvents(string gameId)
    {
        await Groups.AddToGroupAsync(Context.ConnectionId, groupName: gameId);
    }
}
```

Рис. А.3 – Код EventHub який використовується в IntegrationService

Джерело: розроблено автором

```
3 usages
private bool ValidateLevelCompleted(Dictionary<string, object> eventData, Dictionary<string, object> conditions)
{
    if (!eventData.TryGetValue("startTime", out var startTimeObj :object? ) ||
        !eventData.TryGetValue("endTime", out var endTimeObj :object? ))
        return false;

    if (!long.TryParse( s: startTimeObj.ToString(), out var startTime :long ) ||
        !long.TryParse( s: endTimeObj.ToString(), out var endTime :long ))
        return false;

    var durationSeconds :double = (endTime - startTime) / 1000.0;
    var minTime :double = double.Parse(conditions["minTime"]?.ToString() ?? "0");
    var maxTime :double = double.Parse(conditions["maxTime"]?.ToString() ?? "300");

    return durationSeconds >= minTime && durationSeconds <= maxTime;
}
```

Рис. А.4 – Код валідації ігрової події на завершення рівня

Джерело: розроблено автором

```
1 usage   More...
private async Task ConnectToSignalR(string gameId)
{
    var hubConnection = new HubConnectionBuilder()
        .WithUrl(_apiClient.EventUrl) // IHubConnectionBuilder
        .Build();
    hubConnection.On<string, string>(EVENT_METHOD, (eventId :string , status :string ) =>
    {
        Debug.Log($"Event {eventId} updated to {status}");
    });
    await hubConnection.StartAsync();
    await hubConnection.InvokeAsync(SUBSCRIBE_METHOD, gameId);
}
```

Рис. А.5 – Код підключення до серверного SignalR

Джерело: розроблено автором

ДОДАТОК Б

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /app
COPY . .
RUN dotnet restore
RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/aspnet:8.0
WORKDIR /app
COPY --from=build /app/out .
ENV ASPNETCORE_URLS=http://+:5300
EXPOSE 5300
ENTRYPOINT ["dotnet", "TradeService.dll"]
```

Рис. Б.1 – Код Dockerfile з TradeService

Джерело: розроблено автором

```
services:
  auth_db:
    image: postgres:15
    environment:
      POSTGRES_USER:
      POSTGRES_PASSWORD:
      POSTGRES_DB: auth_db
    volumes:
      - auth_db_data:/var/lib/postgresql/data
      - ./db-init/auth_db.sql:/docker-entrypoint-initdb.d/auth_db.sql
    ports:
      -
    networks:
      - app-network
```

Рис. Б.2 – Код DockerCompose файлу з прикладом конфігурації для auth_db

Джерело: розроблено автором

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-service
  namespace: metaverse
spec:
  replicas: 2
  selector:
    matchLabels:
      app: auth-service
  template:
    metadata:
      labels:
        app: auth-service
    spec:
      containers:
        - name: auth-service
          image: auth-service:latest
          ports:
            - containerPort: 5000
          env:
            - ConnectionStrings__DefaultConnection = Host=auth-db.metaverse.svc.cluster.loc…
          resources:
            requests:
              cpu: "100m"
              memory: "128Mi"
            limits:
              cpu: "500m"
              memory: "512Mi"
```

Рис. Б.3 – Код з файлу розгортання для Kubernetes

Джерело: розроблено автором

```yaml
apiVersion: v1
kind: Service
metadata:
  name: auth-service
  namespace: metaverse
spec:
  selector:
    app: auth-service
  ports:
    - port: 5000
      targetPort: 5000
  type: ClusterIP
```

Рис. Б.4 – Код з сервіс файлу auth-service для Kubernetes

Джерело: розроблено автором

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: api-gateway-ingress
  namespace: metaverse
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/websocket-services: integration-service
    nginx.ingress.kubernetes.io/upstream-vhost: integration-service.metaverse.svc.cluster.local
spec:
  rules:
    - host: metaverse.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: api-gateway
                port:
                  number: 5100
          - path: /eventHub
            pathType: Prefix
            backend:
              service:
                name: integration-service
                port:
                  number: 5400
```

Рис. Б.5 – Код з ingress файлу api-gateway для Kubernetes

Джерело: розроблено автором

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: auth-service-hpa
  namespace: metaverse
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: auth-service
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
```

Рис. Б.6 – Код з HPA файлу auth-service для Kubernetes

Джерело: розроблено автором