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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____20 22 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

на тему: Серверна програмна система мобільної багатокористувацької асиметричної VR-гри

Виконав : студент __4_ курсу, групи ___ІП-83
(шифр групи)
_____Бойко Андрій Олександрович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник ____проф., д.т.н. Михайло НОВОТАРСЬКИЙ_____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант(нормоконтроль) проф., д.т.н. Симоненко В.П. _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент __м.н.с, к.т.н. Сергій СУШКО_____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

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

Київ – 2022 р.

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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2022 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Бойка Андрій Олександровича

1. Тема проєкту *Серверна програмна система мобільної багатокористувацької асиметричної VR-гри*
керівник проєкту Новотарський М. А., проф., д.т.н.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 20.12.2021

2. Термін здачі студентом закінченого проєкту *10 червня 2022 року*

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

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

Розділ 1. Аналіз предметної області

Розділ 2. Проєктування серверної системи

Розділ 3. Реалізація серверної системи мобільної багатокористувацької асиметричної vr гри

Розділ 4. Огляд створеної серверної системи мобільної багатокористувацької асиметричної VR гри

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, функціональна схема (діаграма класів), алгоритм дій програмного забезпечення.

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

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Симоненко В. П		

7. Дата видачі завдання «30» серпня 2021 р.

Календарний план

№ П/П	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	08.02.2022	
2.	*Вивчення та аналіз завдання*	02.05.22 – 08.05.22	
3.	*Розробка архітектури та загальної структури системи*	09.05.22 – 15.05.22	
4.	*Розробка структур окремих Підсистем*	16.05.22 – 22.05.22	
5.	*Програмна реалізація системи*	23.05.22 – 29.05.22	
6.	*Оформлення пояснювальної записки*	29.05.22 – 10.06.2022	
7.	*Захист програмного продукту*	13.05.22	
8.	*Передзахист*	17.05.22	
9.	*Захист*	24.06.2021	

Студент-дипломник Андрій Бойко

(підпис)

Керівник проєкту Михайло Новотарський

(підпис)

АНОТАЦІЯ

У даній роботі було детально розглянуто мобільні багатокористувацькі асиметричні VR ігри, а також принципи їх роботи. Були проаналізовані їх слабкі та сильні сторони. На основі аналізу було вибрано дві кращі гри, які лягли в основу створення власної серверної системи мобільної багатокористувацької асиметричної VR гри. Розроблена серверна система дає можливість встановлювати зв'язок між гравцями у режимі реального часу, оброблювати ігрові обчислення, переглядати інформацію про гравців та сесії. Програмний продукт був розроблений на мові Javascript.

Ключові слова: ігровий сервер, віртуальна реальність, взаємодія у реальному часі, мова програмування.

ANNOTATION

In this project for a Bachelor's Degree, mobile multiplayer asymmetric VR games, as well as the principles of their operation were discussed in detail. Their strengths and weaknesses were analyzed. Based on the analysis, the two best games were selected, which formed the basis for creating your own server system for mobile multiplayer asymmetric VR games. The developed server system allows you to establish a connection between players in real time, process game calculations, view information about players and sessions. The software product was developed in Javascript.

Keywords: game server, virtual reality, real-time interaction, programming language.

Справки	Формат	Значення	Найменування	Кіл. листів	№ екземпля	Додаток
			Документація загальна			
			Знову розроблена			
	А4	*ІАЛЦ.467200.002 ТЗ*	Серверна програмна система	3		
			мобільної			
			багатокористувацької			
			асиметричної VR-гри			
			Технічне завдання			
	А4	*ІАЛЦ.467200.003 ПЗ*	Серверна програмна система	60		
			мобільної			
			багатокористувацької			
			асиметричної VR-гри			
			Пояснювальна записка			
	А4	*ІАЛЦ.467200.004 Д1*	Серверна програмна система	1		
			мобільної			
			багатокористувацької			
			асиметричної VR-гри			
			Схема структурна			
	А4	*ІАЛЦ.4672008.005 Д2*	Серверна програмна система	1		
			мобільної			
			багатокористувацької			
			асиметричної VR-гри			
			Діаграма класів			
	А4	*ІАЛЦ.467200.006 Д3*	Серверна програмна система	1		
			мобільної			
			багатокористувацької			
			асиметричної VR-гри			
			Принципова схема			
	А4	*ІАЛЦ.467200.007 Д4*	Серверна програмна система	60		
			мобільної			
			багатокористувацької			
			асиметричної VR-гри			
			Лістинг програми			

Зм	*Лист*	*№ докум.*	*Підп*	*Дата*	

ІАЛЦ.467200.001 ОА

Розроб		Бойко А. О.			*Серверна прогрмна система мобільної багатокористувацької асиметричної VR-гри* Опис альбому	Літ.	Аркуш	Аркушів
Перев		Новотарський М. А.					1	1
						НТУУ "КПІ" ФІОТ ІП-83		

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Серверна програмна система мобільної багатокористувацької асиметричної VR-гри*»

Київ – 2022

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку серверної програмної системи мобільної багатокористувацької асиметричної VR гри, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування даної системи є багатокористувацькі асиметричні ігри у віртуальній реальності у режимі реального часу.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

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

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка серверної програмної системи мобільної багатокористувацької асиметричної VR гри.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблювальна система має виконувати такі вимоги:

- Взаємодіяти з клієнтами у режимі реального часу
- Оброблювати ігрову логіку
- Бути легко розширюваною та масштабованою
- Надавати користувачам доступ до власної інформації та інформації системи

5.2. Вимоги до програмного забезпечення

- Windows, MacOS or Linux
- Node.js версії 16.0.0 або більше

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Pentium® G5400.
- ROM не менше ніж 500 МБ.
- RAM не менше ніж 1 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	
Вивчення та аналіз завдання	
Розробка архітектури та загальної структури системи	
Розробка структур окремих частин системи	
Програмна реалізація системи	
Виправлення помилок	
Оформлення пояснювальної записки	

ПОЯСНЮВАЛЬНА ЗАПИСКА

ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Серверна програмна система мобільної багатокористувацької асиметричної VR-гри*»

Київ – 2022

ЗМІСТ

ВСТУП

В наш час розважальна сфера не стоїть на місці. На зміну застарілим способам відпочинку з'являються нові альтернативи, які все більше покладаються на науковий прогрес та спираються на сучасні технології. Замість звичного століттями телебачення та кумедних телешоу прийшли відео у телефоні. На зміну гральним автоматам у спеціальних клубах були винайдені настільні персональні комп'ютери з іграми доступними кожному через мережу Інтернет. Але і цього виявилось у свій час замало, тож новим витком технологій стала віртуальна реальність. Завдяки ній у людини виникає відчуття ілюзії дійсності у іншому вигаданому просторі, який зазвичай обмежується тільки фантазією розробника і апаратними ресурсами. Це дозволяє отримати переваги над існуючими рішеннями багатьох проблем у різних сферах життєдіяльності, таких як: навчання, наука, дизайн, розваги та інші. Саме можливість зануритися у тур віртуальних екскурсій світом не купуючи квиток до іншого кутку світу, або перспектива відчути себе космонавтом не проходячи виснажливий курс підготовки, який не кожний витримає, робить цей напрямок технологій привабливим та перспективним.

Одним із найвагоміших переваг розваг за допомогою ігор без перебільшення є можливість взаємодії між різними гравцями задля досягнення командного результату або найкращого результату серед протиборчих сторін. З метою досягнення даної взаємодії між гравцями, які знаходяться у різних локаціях, використовується різни види підключень, серед яких найпростішим варіантом є пряме підключення між гравцями, яке має досить велику кількість недоліків. Іншим рішенням є виділення спеціального хмарного серверу, який буде брати на себе задачі по з'єднанню та синхронізації між гравцями. Саме цей варіант є найбільш популярним та універсальним щодо більш складних ігрових систем.

Проте серверних рішень для багатокористувацьких ігор, через тільки-но зростаючу популярність, все ще невелика кількість. Ці рішення, по-перше, не є системами з відкритим вихідним кодом, що не дозволяє вільно використовувати їх у особистих цілях, покращувати та масштабувати. По-друге, через брак оптимізації та контролю над ресурсами, системи не є максимально оптимізованими навіть при невеликих навантаженнях. Через це постає проблема: завчасно обирати витриваліші хмарні ресурси, але при малих кількостях клієнтів марно витрачати гроші на утримання невикористаних потужностей, або постійно слідкувати за критичним рівнем навантаження і безперервно стикатися з затримками та проблема зі сторони користувачів. Саме тому була розроблена серверна програмна система мобільної багатокористувацької асиметричної VR-гри представлена в даній дипломній роботі. Основною відмінністю серверної програми є висока швидкість взаємодії з клієнтами, можливість масштабуватися в залежності від кількісті підключень, оптимальний вибір локації для зменшення затримки з'єднання та оптимізація використання серверних ресурсів.

Метою дослідження є розробка серверної програмної системи мобільної багатокористувацької асиметричної VR гри.

Завдання дослідження:

1. Зробити аналіз предметної області.
2. Визначити вимоги до розроблювальної системи
3. Обрати мережеву архітектуру
4. Вибрати технології розробки
5. Описати розробку системи
6. Дослідити та проаналізувати розроблену систему

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд асиметричного ігрового процесу в іграх

У багатьох іграх існує конкуренція між гравцями, пов'язаними однаковими правилами, з однаковими або збалансованими ігровими здібностями. Ці ігри є симетричними[1]. Асиметричний ігровий процес відбувається в ігровому середовищі, де у різних гравців різні правила, здібності або цілі. Класичним прикладом асиметричного ігрового процесу є гра 1970 року "Mastermind", в якому один гравець повинен вибрати приховану послідовність кольорів, а інший повинен вивести послідовність за обмежену кількість ходів [2]. Асиметричний ігровий процес поширений у всіх цифрових іграх і, як було показано, сприяє командній роботі, критичному мислення і навичкам ведення переговорів більш ефективно, ніж симетрична гра[3].

Асиметричний ігровий процес поширений як у кооперативних іграх, так і в змагальних. У багатьох онлайн іграх асиметрія може існувати всередині команд у вигляді різних "героїв" або "класів", які можуть мати різні стилі гри. В онлайн-іграх стало нормою, щоб гравці в команді володіли різними взаємодоповнюючими здібностями, оскільки така настройка за своєю суттю сприяє командній роботі . Наприклад, у багатьох популярних онлайн-іграх, таких як Overwatch і League of Legends, є персонажі "підтримки", ролі яких в першу чергу націлені для зцілення або захисту головного гравця команди. Набагато простіше мати окремого гравця для зцілення і окремого гравця для нанесення шкоди, задля покращення командної роботи. Асиметричний дизайн гри також може дозволити гравцям з різним рівнем майстерності грати разом, наприклад, один гравець може використовувати сенсорний інтерфейс, щоб внести додаткову користь в ігровий процес. Або ж додати інклюзивності, коли гравці можуть знайти ролі, найбільш підходящі для їх унікального стилю, який допомагаю здобувати перевагу та розкривати можливості.

Недавнє відродження ігор віртуальної реальності (VR) надало безліч нових можливостей для дослідження і унікального ігрового дизайну. Незважаючи на те, що є безліч цікавих однокористувацьких ігор, недавній звіт Greenlight Insights показує, що 77% користувачів віртуальної реальності хочуть більше соціальної активності [4]. Хоча існує велика кількість багатокористувацьких VR-ігор, більшість з них онлайн, що означає, що гравці не знаходяться в одному і тому ж загальному фізичному простір. Крім того, для цих ігор потрібна гарнітура віртуальної реальності на кожну людини. Гарнітури швидко стають все більш потужними і портативними, наприклад, недавно випущений бездротової Oculus Quest [5]. Додаткова мобільність цих і майбутніх пристроїв призведе до збільшення ситуацій, в яких віртуальна може поширюватись в межах одного фізичного простору, оскільки тепер її можна переносити набагато простіше, ніж до появи цих бездротових рішення. Будь то вдома або в ігровому залі, соціальна динаміка у фізичному середовищі - це та, якою нехтують в існуючій літературі і в комерційних проектах.

Щодо популярності VR індустрії, можна навести статистичні дані щодо існуючих ігрових шоломів, та аналітично визначити як багато їх буде в найближчому майбутньому. За поточними даними на 2022 рік, шоломами віртуальної реальності володіють приблизно 35 мільйонів людей, що майже вдвічі більше аніж за аналогічний період 2019 року. Дані зображені на рис. 1.1

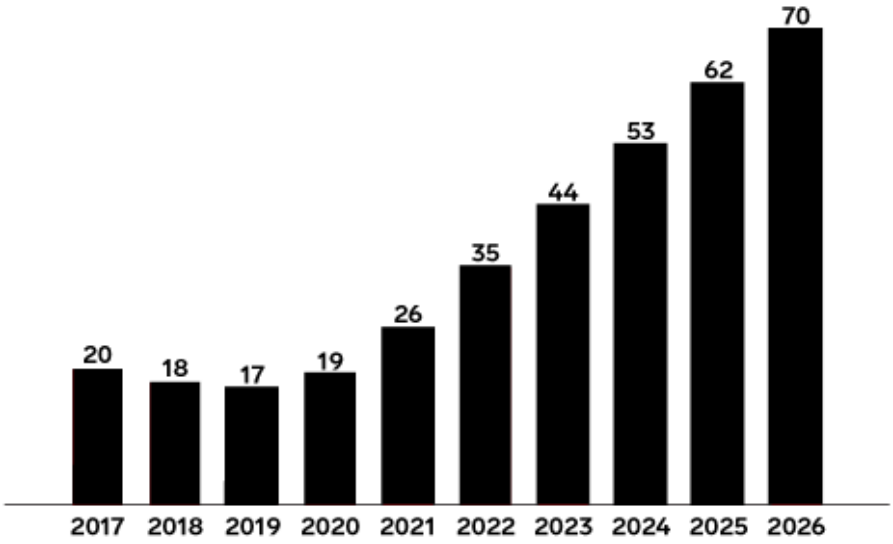


Рис. 1.1 Кількість віртуальних шоломів у світі у мільйонах

Отже можна побачити, що кількість шоломів буде збільшуватись лінійно з кожним роком. Але ці цифри все ще невеликі порівняно з «класичним» видом ігрових розваг (рис 1.2), який еволюціонував починаючи з 1962 року, що робить VR індустрію відносно молодою та ще тільки-но розвиваючою.

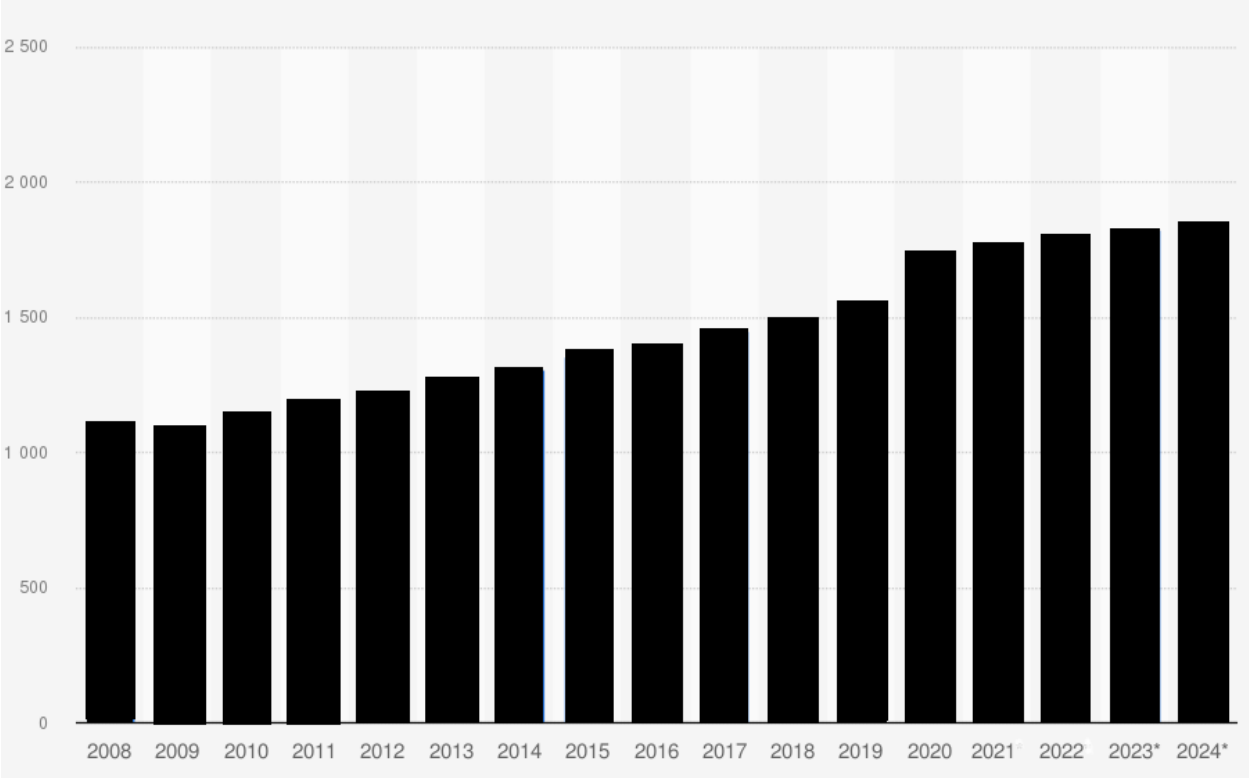


Рис 1.2 Кількість загальних ігрових користувачів на комп'ютерній платформі у мільйонах

Основними перешкодами є замалий вибір контенту, який не буде швидко набридати та викличе постійний інтерес до шолому, незадоволення у самому використані шолому, через незвичність людини знаходитись у іншому просторі, що викликає неприємні відчуття, а також все ще невелику доступність в плані ціни. Статистичні дані по відсотками наведені у рис. 1.3.

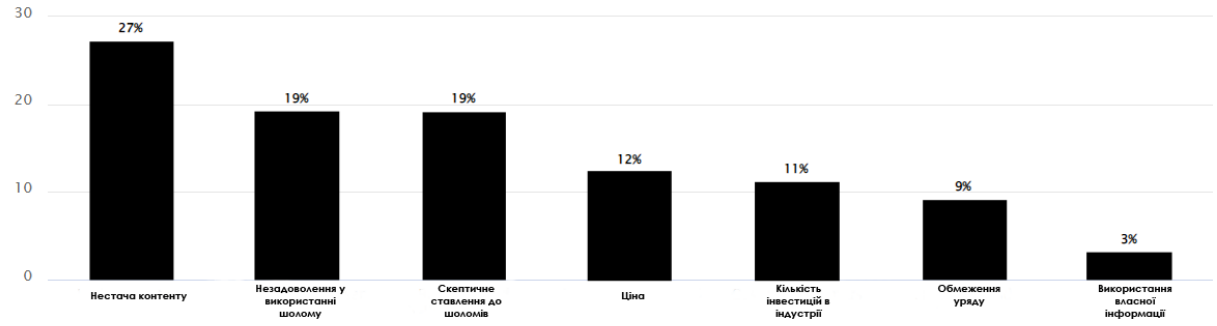


Рис. 1.3 Основні перешкоди у користуванні шоломів віртуальної реальності

Більшість вражень від віртуальної реальності отримуються тільки від того, що відбувається всередині гарнітури, залишаючи без уваги не менш важливі компоненти досвіду поза шоломом або пристроєм. У той час як деякі люди можуть грати наодинці, багато хто буде насолоджуватись віртуальною реальністю з друзями вдома або в ігрових залах. У цих сценаріях є люди, які очікують своєї черги до гарнітури задля отримання відповідних вражень. Оскільки ігровий дизайн, найчастіше, строго те, що відбувається всередині гарнітури, користувацький досвід людей, що тільки-но очікують своєї черги не враховується. Але не треба забувати, що користувальницький досвід - це щось більше, ніж просто сама гра, він швидше включає в себе контекст, що оточує гру: яка навколишнє середа до того, як ми надягаємо гарнітуру, і після того, як ми її знімаємо. Кожен елемент може вплинути на те, як ми ставимося до гри, а також на наше її сприйняття.

Саме через сукупність факторів незадоволенням тільки грою у VR, без додаткових взаємодій с навколишнім світом та невдоволення гравців, які з деяких причин не мають можливості опанувати віртуальні технології на даний момент, асиметричний VR стає частковим вирішення проблеми.

1.2 Теоретичні відомості

1.2.1 Опис асиметричності у VR індустрії

Асиметрична VR гра - підклас багатокористувацької віртуальної реальності, який пропонує не всім учасникам однакові можливості взаємодії з

віртуальною сценою. У той час як один користувач може бути занурений за допомогою VR-дисплея, встановленого на голові (HMD), інший користувач може випробувати віртуальну реальність за допомогою звичайного настільного ПК, мобільної платформи або з будь-якої консолі[6]. Така взаємодія має на увазі розподілення відповідних ролей з унікальним можливості у кожної із сторін, яка створює повну картину та занурює у процес кожного гравця, усуваючи недоліки можливої відсутності VR шолому. Оскільки у сучасному світі майже у кожної людини із собою є смартфон, з достатньою обчислюваною потужністю задля встановлення та запуску оптимізованого мобільного додатку, більш доцільним є спрямовувати розробку саме на мобільним напрямок. Це дозволяє розшити можливості використання та популярності гри через саме мобільність ігрових систем, бо задля гри в компанії у одному фізичному просторі, тягнути важкі та громіздкі настільні комп'ютери не є доцільним. Хоча, це трохи обмежує можливості розробника, можливі переваги перевершують усі обмеження.

Повноцінна гра складається з 3 компонентів – клієнтської частини у віртуальній реальності, клієнтської частини мобільного системи та серверного застосунку, який пов'язує усі клієнтські частини між собою. Оскільки темою даної роботи є саме серверний застосунок, розглянемо обов'язкові властивості притаманні типовим рішенням.

1.2.2 Взаємодія у реальному часі

Багатокористувацька гра, в якій гравці взаємодіють в режимі реального часу, тобто, коли гравець виконує будь-яку дію, інші гравці дізнаються про наслідки цієї дії протягом встановленого терміну. Якщо один гравець робить рух у напрямку X, інші гравці підключені до ігрової сесії мають побачити цей рух з мінімальною затримкою, аби мати можливість побачити та обробити цю дію. Оскільки розбіжність у сприйнятті гравців може призвести до небажаних результатів, такі ігри тягнуть за собою підвищені обмеження на оперативність і послідовність, а дизайн серверної частини має бути побудованим саме під одночасні оновлення, мінімізуючи затримку. Основною можливою

проблемою є інтернет з'єднання та невдале мережеве архітектурне рішення. Задля уникнення можливих проблем цього плану, дані питання були у частини проектування.

1.2.3 Симуляція ігрової логіки

Перше і головне правило розробки сервера: клієнт знаходиться в руках ворога. Клієнт, зазвичай, доволі сильно захищений, але чисто теоретично (а інколи і практично) і його можуть хакнути - розшифрувати клієнт-серверний протокол, знайти вразливі місця, отримати доступ до чутливої інформації. Злом клієнта може привести до обходу ігрових правил, нечесній грі, використанні ком'ютера замість гравця та інші. Такі речі руйнують концепт гри для всіх і виставляє розробника некомпетентним. Щоб цього не сталося, ми повинні емулювати весь ігровий світ з усіма ігровими правилами у себе на сервері, а клієнт використовувати тільки для відображення красивої картинки. Використовуючи сервер як агрегат для обчислення ігрової логіки зменшує можливість шахраїв знайти відповідні вразливості і додає захисту, який ніколи не буває зайвим. Крім того, клієнт має постійно перевірятися на злом, відстежуючи підозрілу поведінку, або нетипові речу для звичайного гравця, через постійну еволюцію хакерських атак.

1.3 Огляд існуючих аналогів

Оскільки серверна система не може існувати окремо від повноцінної гри, був проведений комплексний аналіз повноцінних VR ігор.

1.3.1 Do you copy?

Do you copy – багатокористувацька асиметрична VR гра, яка пропонує кооперативних ігровий процес задля досягнення спільної мети. Суть гри полягає у досягненні спільної мети – пограбування банку, через постійну голосову взаємодію гравців, які знаходяться у одній кімнаті. Людина яка використовує шолом, знаходиться у приміщенні банку, який має успішно пограбувати. Вона має розв'язувати загадки, знаходити правильний шлях до місце перебування грошей та уникати охоронців. Вирішити ці завдання їй

допомагає інший гравець, який використовує комп'ютерну систему, з якої є доступ до карти приміщення у реальному часі та відповіді на усі загадки. Ключ до успішного пограбування – комунікація між гравцями та гарне володіння своєю роллю. Із основних позитивних особливостей гри можна виділити:

- Асиметричний ігровий процес
- Локальний мультиплеєр
- Ранній доступ
- Зручний користувацький інтерфейс
- Реграбельність
- Широка підтримка шоломів

Із основних недоліків можна помітити:

- Відсутність гри через мережу
- Низьку швидкість оновлення кадрів на секунду
- Висока ціна
- Слабку оптимізацію
- Обмежена швидкість оновлення даних від серверу
- Високі системні потреби

1.3.2 Panoptic

Panoptic – локальна багатокористувацька змагальна асиметрична VR гра, в якій два гравця, граючи або за гігантського наглядача, або за крихітне створіння, мають протистояти один одного. Завдання наглядача, який використовує шолом віртуальної реальності, захистити свої головні споруди від маленького створіння – гравця з комп'ютера. Кожна роль має свої унікальні можливості, що дозволяє додати різноманітності у ігровий процес.

Основними перевагами цієї гри є:

- Асиметричний ігровий процес

- Локальний мультиплеєр
- Оптимізація затримки
- Широка підтримка шоломів
- Безкоштовний комп’ютерний клієнт
- Висока оптимізація

Серед недоліків можна виділити:

- Відсутність гри через мережу
- Незрозумілий користувацький інтерфейс
- Відсутність підтримки мобільної платформи
- Низька реграбельність
- Обмежена швидкість оновлення даних від серверу
- Обмеження у кількості гравців

ВИСНОВКИ ДО РОЗДІЛУ 1

У першому розділі був проведений аналіз предметної області асиметричних ігор, розглянуто визначення асиметричного ігрового процесу та його переваги над класичним симетричним підходом при розробці багатокористувацьких ігор, такими як:

- Сприяння командній роботі
- Розвиток критичного мислення
- Покращення навичок ведення переговорів.

Був проведений огляд симетричного підходу до проектування багатокористувацьких ігор у сфері віртуальної реальності, який визначив основні недоліки, такі як:

- Недостатня кількість цікавих багатокористувацьких ігор
- Незручність використання шолому у багатьох людей одночасно через незвичні відчуття
- Висока ціна на обладнання

Дивлячись на вище згадані недоліки, було визначено, що асиметричний підхід до проектування дає можливість частково вирішити проблеми, через впровадження високої реграбельності, зменшення кількості шоломів та можливості змінювати ролі під час самої гри.

Були визначені і розглянуті основні властивості асиметричної серверної системи для віртуальної реальністю, такі як:

- Взаємодія у реальному часі
- Симуляція ігрової логіки

Проаналізувавши основні наявні аналоги багатокористувацьких асиметричних VR ігор були визначені такі недоліки:

- Відсутність підтримки мережевого з’єднання
- Недостатня підтримка великої кількості гравців

- Обмеження у швидкості оновлення інформації від серверу
- Нестача підтримки мобільних платформ

Зважаючи на перелічені вище недоліки, були визначені основні функціональні можливості багатокористувацької серверної системи для асиметричної VR гри:

- Можливість онлайн з’єднання
- Висока швидкість надходження оновлень стану гри
- Підтримка великої кількості гравців
- Оптимізація мережевого рішення
- Можливість швидко масштабуватися в залежності від навантаження

РОЗДІЛ 2
ПРОЄКТУВАННЯ СЕРВЕРНОЇ СИСТЕМИ

2.1 Постанова задачі

Задля успішної реалізації нашої системи, на початку проектування мають буди визначені кінцеві функції застосунку. Система складається з двох різних модулів, які є незалежними частинами, але можуть взаємодіяти через протокол HTTP. Перший модуль програми обробляє так звані «статичні» операції, тобто слідує класичному підходу запит-відповідь задля надання клієнту інформації, яка не змінюється в реальному часі. Другий модуль програми зосереджений тільки на взаємодії з клієнтами у режимі реального часу і не має доступу до статичних ресурсів.

Розглянемо статичний модуль програми більш детально:

- Авторизація через обліковий запис Facebook – через необхідність впровадження базових засобів безпеки і надання кожному клієнту свій унікальний ідентифікатор, система має надавати можливість авторизації за допомогою OAuth 2.0 – відкритого стандарту авторизації, який дозволяє надавати доступ до приватних даних, що зберігаються на іншому сайті без необхідності вводу імені користувача та паролю[7].
- Перегляд та редагування власного профілю – система має надавати інформацію про власний профіль користувача і також давати можливість його змінити, або за бажанням видалити.
- Перегляд поточних і минулих сесій – застосунок має зберігати усі розпочаті та закінчені сесії кожного гравця з відповідним результатом, та надавати можливість їх переглядати.

Розглянемо модулі програми, які мають виконуватись у режимі реального часу:

- Встановлення зв’язку між гравцям – система має прийняти та обробити підключення від кожного клієнта. Також має бути визначена можлива затримка між кожним гравцем, та прийняті рішення з можливої її оптимізації.
- Перегляд відкритих до підключення ігрових кімнат з повною інформацію про вже підключених клієнтів.
- Розподіл ролей та старт гри – при отриманні сигналу готовності від кожного клієнту, система має відповідно до типу підключення (VR-шолом або мобільний пристрій) визначити роль та надати клієнту повноваження до запуску ігрової сесії, а також стати відкритою до прийняття та обробки подій зміну стану.
- Обробка ігрових обчислень – система має реагувати на вхідний потік даних від кожного клієнта, робити відповідні обчислення та відповідати кожному гравцю про можливу зміну ігрового стану.
- Повторне підключення – при умові втрати з’єднання між клієнтом та сервером, мають бути вжиті усі можливі заходи зі сторони системи для відновлення підключення та примусового оновлення стану відключеного клієнту до поточного.

Щодо ігрових обчислень, то цей пункт треба розглянути окрему. Визначимо, які саме обчислення має проводити наша серверна система, аби визначити для яких клієнтів підходить наш застосунок. Отже серверна система оброблює 3-D переміщення кожного клієнта у просторі, зважаючи на VR гравця та мобільних клієнтів. Також система реєструє нанесення шкоди кожному із гравців, а значить вона підходить під будь-яку асиметричну багатокористувацьку змагальну VR-гру у реальному часі.

2.2 Вибір мережевої архітектури

В наш час були розроблені різні види підключень між клієнтами під час онлайн-гри. Серед них найбільш популярними є:

- Client-Server Architecture

- Peer-to-Peer Architecture
- Client-Multi-Server Architecture

Розглянемо кожен із цих архітектурних рішень детально, аби знайти найбільш відповідний нашій серверній системі. Але перед цим, обираючи спосіб спілкування між гравцями, які мають обробляти дані у режими реального часу, декілька важливих архітектурних моментів мають бути враховані.

Першим моментом є тип гри, під яку будується мережева архітектура. В залежності від того, як швидко потрібно реагувати на зміни, які відбуваються між гравцями, мають бути визначені критерії для максимально можливої затримки. Наприклад, за даними дослідження, було виявлено, що при затримці у 100 мілісекунд у гоночній грі, вплив на керування авто майже непомітний. Але при умовах виникнення затримки більш аніж на 200 мілісекунд, керування віртуальним автомобілем стає майже неможливим [8]. В той час як при наявності будь-якої затримки більшою за 80 мілісекунд у шутері, де реакція і швидкість передачі даних грають чи не найважливішу роль, можливість гравця видавати гідний результат суттєво понижується. В той самий час, багато ігор пропонують таких ігровий процес, який дозволяє грати з великими затримками без будь-яких проблем. У стратегічних і рольових іграх час реакції не є фактором. Навіть ігри, схожі на екшн може маскувати затримки, реалізуючи нечіткі або повільні дії, в той час як інші клієнти або сервер обробляють інформацію.

Другим моментом є максимально можлива мінімізація періоду десинхронізації. У іграх з великою щільністю дій на секунду, постає проблема вчасно повідомляти підключених клієнтів про постійні зміни. Ціллю є обмежити можливі проблеми з синхронізацією та наперед визначити шляхи вирішення ймовірних проблем.

Третім моментом є пропускна здатність. Вимоги до пропускної здатності також дуже важливі. В архітектурі клієнт-сервер ми не можемо

ігнорувати вимоги до пропускної здатності на сервері. Її потреби швидко зростають зі збільшенням кількості гравців, тому канал доступу може стати вузьким місцем в грі. Обмеження пропускної здатності також впливає на клієнтів; гра для восьми гравців споживає 50 Кбіт/с на стороні клієнта, що ускладнює відтворення через модем. Що ще гірше, при умові що сервер споживає 400 Кбіт / с, кожен з підключених клієнтів повинен мати хороший зв'язок [9]. Тож можливість зменшення пропускної здатності зі сторони клієнта, за допомогою вибраної архітектури підключення, є критичною.

Щодо порівняння пропускної здібності між можливими підходами, спочатку ми маємо визначити як вона буде обраховуватись через загальну формулу. У типовій багатокористувацькій грі гравець знаходиться у певному циклі оновлень протягом всієї гри. Цей цикл на кожній ітерації включає в себе читання вхідних даних локального гравця, отримання оновлень від віддалених гравців, обчислення нового локального стану і, спираючись на оновлений локальний стан, візуалізацію графічного представлення світу гравця. Тривалість цього циклу називається періодом оновлення гравця, який ми позначимо як T_u. Хоча він може варіюватися в залежності від апаратного забезпечення та стабільності підключення до мережі, для простоти припустимо, що всі гравці мають той же період оновлення. Кожен гравець відправляє повідомлення про оновлення на кожній ітерації циклу всім іншим гравцям, або на центральний сервер. Розмір повідомлень про оновлення може варіюватися в залежності від активності гравця під час даної ітерації циклу. Щоб спростити обчислення вимог до пропускної здатності, ми також припустимо, що всі оновлення мають однаковий розмір, а саме L_u байт. Таким чином, вимоги до пропускної здатності гравця або сервера визначаються кількістю повідомлень про оновлення, відправлених і отриманих в кожен період оновлення. Це число кількість повідомлень залежить від базової архітектури гри, а також від способу виявлення і усунення десинхронізації. Наприклад, якщо клієнт надсилає та отримує N-кількість оновлень при кожній

ітерації ігрового циклу, пропускна здатність може бути представлена формулою $N\frac{L_u}{T_u}$ [10].

Четвертий пункт являється сукупністю чинників адміністративного контролю та заходів безпеки. Використовуючи підключення, які базуються на використанні виділеного серверу, розробник отримує доступ до важливих функцій покращення стабільності гри, таких як:

1. Доступ до всіх гравців та можливість контролю над ними та/або збір потрібної інформації для покращення продукту
2. Можливість швидких покращень або виправлення помилок через випуск оновлень тільки для серверу, без участі клієнту
3. Боротьба с цифровим піратством за допомогою ідентифікації гравців

Останнім основним пунктом для вибору потрібного рішення є перспектива масштабування та оптимізація ресурсів. Якщо логіка гри передбачає можливу зміну навантаження, наприклад, від кількості гравців, архітектурне рішення повинно має практичну відповідь на усі потреби. Також при умові можливих складних обчислень, які не кожен клієнт може обробляти зі швидкістю потрібною для темпу гри, є доцільним обрати більш слушне рішення.

Отже, опишемо кожний підхід відповідно до наших вимог

Client-Server Architecture

Найбільш поширеною архітектурою для багатокористувацьких ігор є клієнт-серверна модель. В цій архітектурі один хост (іноді сам гравець) позначається як сервер, до якого підключаються інші гравці. Кожен клієнт відправляє свої дії на сервер, в той час як сервер використовує отриману інформацію для симуляції ігрового процесу і відправляє оновлений стан назад до гравця, для оновлення відповідного відображення на стороні клієнта. Ця архітектура може бути зображена на рис 2.1

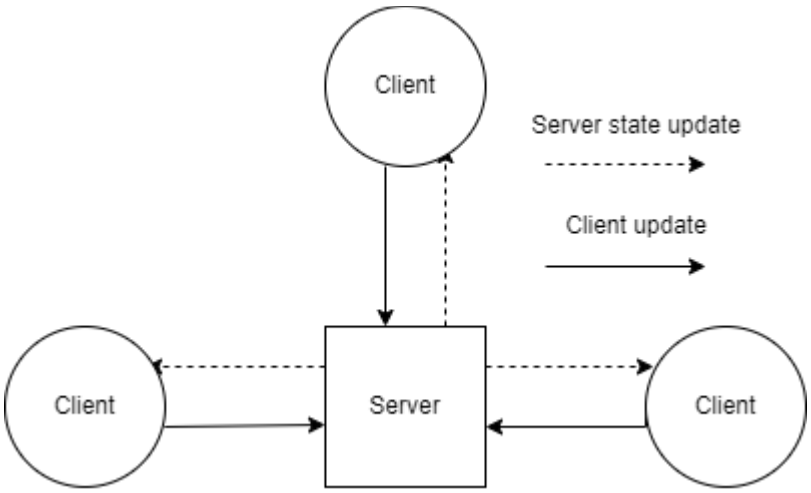


Рис 2.1 Клієнт-серверна архітектура

Визначимо вимоги до пропускної здатності.

В першу чергу, ми розглянемо вимоги до пропускної здатності на стороні клієнта. Клієнт буде відправляти оновлення на сервер кожну ітерацію циклу оновлень, і тому вимога до пропускної здатності висхідного потоку клієнта становить $\frac{L_u}{T_u}$. Коли сервер отримує оновлення гравця, він обчислює новий стан для кожного клієнта і відправляє назад повідомлення про поточний стан гри, який представляє новий стан кожного з N гравців. Розмір повідомлення про поточний стан гри дорівнює $N*L_u$ байт. Отже, пропускну здатність вхідного потоку клієнта можна позначити як $N\frac{L_u}{T_u}$. В результаті наших обчислень, загальна потреба в пропускній здатності у клієнта становить як $(N+1)\frac{L_u}{T_u}$.

Далі, ми маємо врахувати вимоги до пропускної здатності на сервері. У висхідному потоці сервер відправляє повідомлення про поточний стан гри всім гравцям, тому вимога до пропускної здатності низхідного потоку дорівнює $N^2\frac{L_u}{T_u}$. У висхідному потоці сервер отримає повідомлення про оновлення від N клієнтів , тому вимога до пропускної здатності висхідного потоку дорівнює $N\frac{L_u}{T_u}$. Загальна потреба в пропускній здатності на сервері становить: $N(N+1)\frac{L_u}{T_u}$

Існує декілька причин популярності клієнт-серверної архітектури. Це відносно проста у реалізації архітектура, оскільки потрібно турбуватися тільки про одне з'єднання – від клієнта до серверу. Крім того, легко підтримувати синхронізацію між всіма клієнтами через те, що сервер є централізованим тримачем актуального стану, та може слідкувати за станом клієнтів. Також за допомогою оптимізації розміщення ігрового серверу можна суттєво скоротити затримку між клієнтами. Зрештою, централізований сервер дає усі адміністративні можливості, та при правильному налаштуванні надійний захист від нечесних гравців та інтернет піратства.

Розглянемо недоліки даного підходу. По-перше, сервер являє собою єдину точку відмови. По-друге, в той час як вимоги до пропускної здатності у гравців мінімальні, вимоги до пропускної здатності на сервері можуть бути доволі громіздкі, що призводить до багатьох проблем та збільшує потреби до оптимізації. Відзначимо, що існує кілька методів, які можуть бути використані в будь-якій архітектурі для зниження вимог до пропускної здатності, такі як: метод числення координат, метод оновлення даних відносно до інтересу клієнтів, multicast та інші.

Peer-to-Peer Architecture

Більшість ігор, які додатково підтримують багатокористувацькі функції для обмеженого числа гравців використовують однорангову систему підключень. В рамках цієї архітектури кожен клієнт взаємодіє з усіма іншими системами, передаючи їм всі локальні зміни стану по мережі. В однорангової мережі немає традиційного сервера; один клієнт бере на себе роль інформаційного сервера, який відповідає за приєднання нових гравців, зберігаючи поточну інформацію про всіх клієнтів. Архітектури однорангових ігор потребують, щоб копія всього стану гри зберігалася на кожному клієнті. В результаті від гравців вимагається постійна форма синхронізації, аби гарантувати, що кожна копія стану гри однакова. Без синхронізації, через затримку в мережі та інших факторів, ігрові стани клієнтів з часом будуть

розходитися, що призведе до десинхронізації. Ця архітектура може бути зображена на рис 2.2

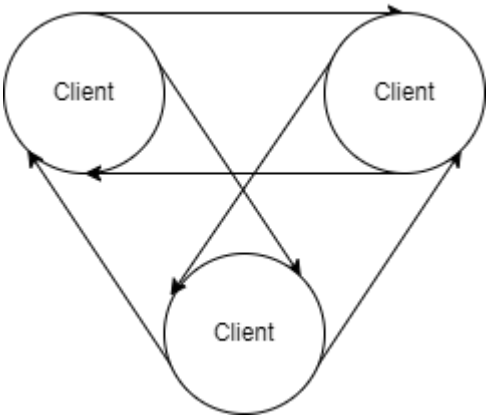


Рис. 2.2 Однорангова система підключень

Щодо вимог до пропускної здібності, оскільки гравець відправляю свою інформацію кожному іншому гравцю, та таким же чином отримує потрібну для оновлення інформацію, загальна потреба до пропускної здібності буде вказана в формулі $2(N-1)\frac{L_u}{T_u}$. Відносно клієнт-серверної архітектури, можна побачити на рис 2.3, що коли кількість гравців менше 4, за умови однакового періоду оновлення гравця та кількості відправлених байт, однорангова мережа справляється краще. Але як тільки кількість гравців стає більшою за 4, клієнт-сервер у пропускній здібності переважає.

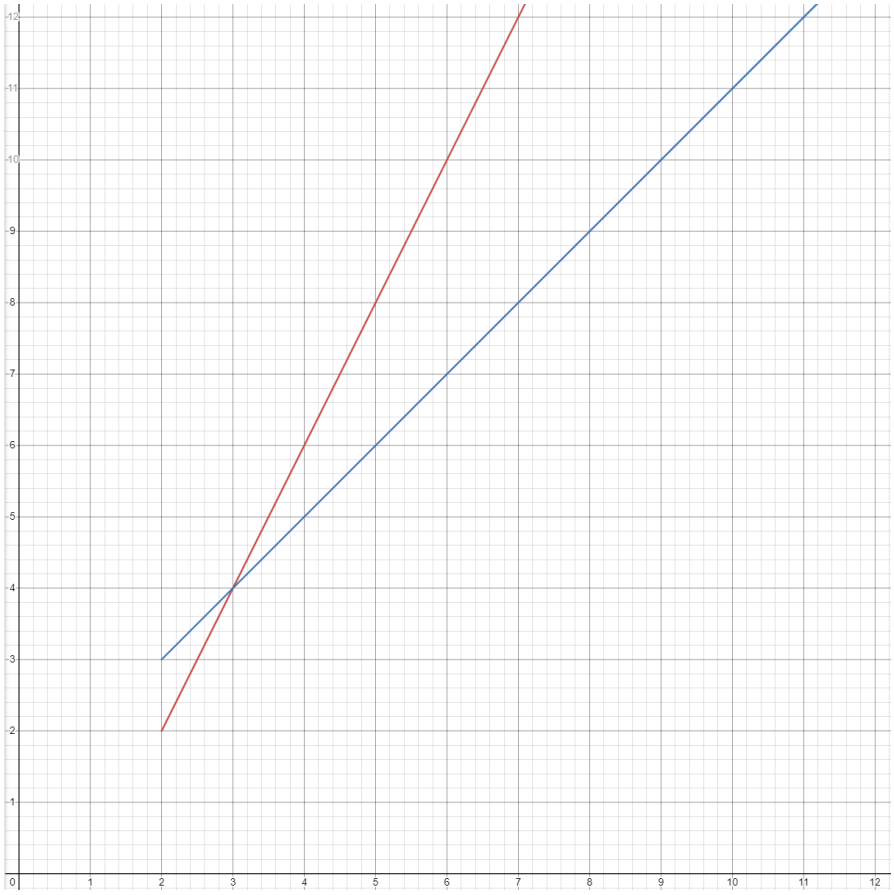


Рис. 2.3 Залежність вимоги пропускної здібності від кількості гравців для клієнт серверної та однорангової архітектур

Відсутність централізованого сервера дає ряд переваг однорангової архітектурі, таких як зменшення затримки повідомлень між клієнтами та усунення єдиної точки відмови. Вона не має можливої проблеми із апаратним забезпеченням серверу, а також ідеально підходить для передачі інформації між невеликою групою гравців.

Щодо недоліків, то кожен гравець окрім симуляції клієнта, має відповідати на можливі попередні обчислення перед оновленням свого стану. Це може спричинити збільшення вимог до апаратного забезпечення клієнта. Оскільки немає сервера для виявлення та усунення десинхронізації між гравцями, будь-які невідповідності повинні бути виявлені гравцями з використанням певного алгоритму чи протоколу, такого як trailing state synchronization[11] або bucket synchronization [12]. Це робить імплементацію та налаштування гри більш складним, та не виключає велику кількість

помилок. Також відсутність централізованого нагляду за гравцями зі сторони серверу, робить гру більш вразливою до захисту та можливого піратства.

Client-Multi-Server Architecture

Поєднання клієнт-серверної та однорангової архітектури являє собою клієнт-мультисерверну архітектуру. У клієнт-багатосерверній архітектурі існує кілька серверів. Кожен сервер відповідає за певну частину ігрового світу, або за незалежні його складові. Як і в архітектурі клієнт-сервер, декілька серверів знаходяться під адміністративним управлінням, а отже отримуються всі його переваги. На жаль, багатосерверна система все одно повинна справлятися з декількома копіями стану гри, що є одним із мінусів підходу однорангової архітектури. У клієнтсько-багатосерверній архітектурі клієнти відправляють повідомлення про оновлення на сервер, до якого вони в даний момент підключені. Сервер обчислює свою власну копію стану гри і надсилає оновлення стану своїм підключеним клієнтам. Потім клієнти оновлюють відповідне відображення. При необхідності кожен сервер може взаємодіяти один з одним. Наприклад, коли клієнти змінюють свій регіон, серверам може знадобитися обмін їх поточною інформацією, щоб оновити їх ігровий стан. Ця архітектура може бути зображена на рис 2.4

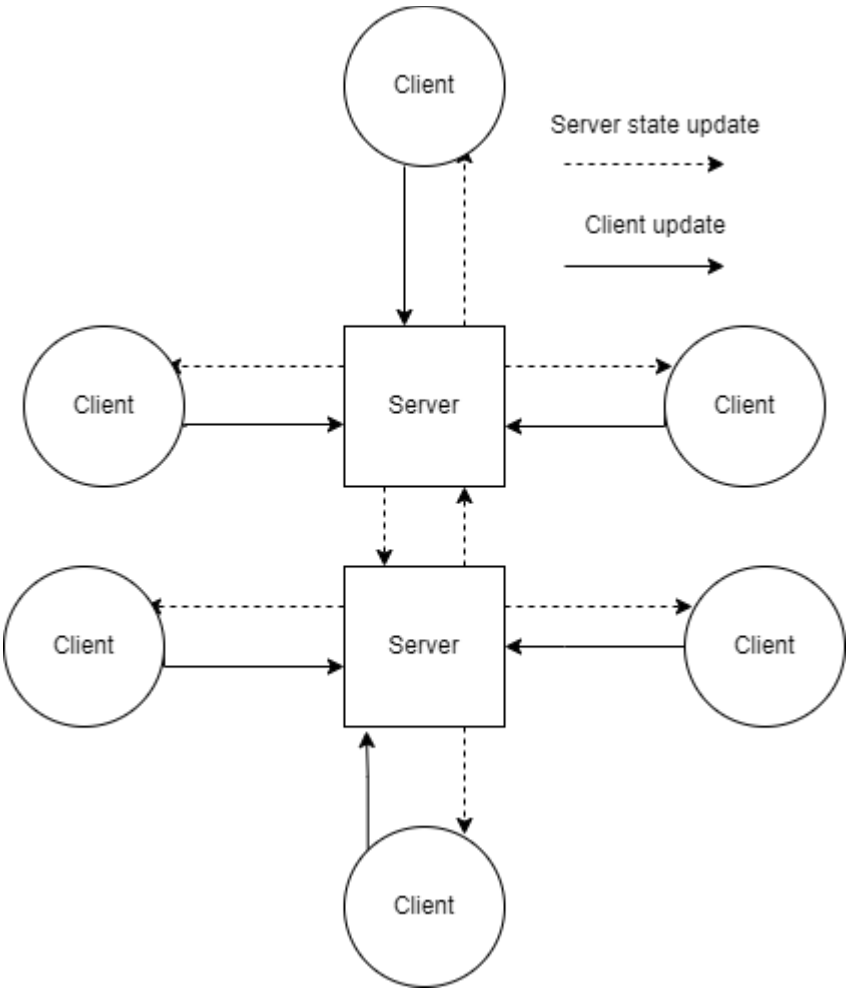


Рис. 2.4 Клієнт-мультисерверна архітектура

Вимога до пропускної здатності для клієнтів в архітектурі клієнт-мультисервер може бути розраховано аналогічно архітектурі клієнт-сервер, яка становить $(N+1)\frac{L_u}{T_u}$. Якщо до кожного серверу підключити однакову кількість клієнтів, вимога до пропускної здатності на загальному сервері буде дорівнювати вимозі до пропускної здатності в архітектурі клієнт-сервер, поділеному на кількість серверів M. Отже розраховуватись за формулою : $\frac{N(N+1)\frac{L_u}{T_u}}{M}$

Переваги архітектури клієнт-мультисервер аналогічні перевагам архітектури клієнт-сервер з додатковою функцією зниження вимог до пропускної здатності на сервері, що усуває проблему вузького місця в пропускній здатності і дозволяє уникнути єдиної точки відмови.

Отже проаналізувавши усі можливі варіанти та спираючись на тип нашої гри, була обрана клієнт-мультисерверна архітектура, через можливість оптимальну затримку між клієнтами та знайденого серверу, достатню пропускну здатність для більше аніж 5 гравців, повний контроль серверу над ресурсами та усунення десинхронізації, адміністративний контроль та засоби безпеки.

2.3 Вибір інструментів розробки

2.3.1 NodeJS

Node.js – це платформа, побудована на середовищі виконання Chrome JavaScript V8 для легкого створення швидких і масштабованих мережевих програм[13]. Node.js використовує керовану подіями, неблокуючу модель вводу-виводу, що робить платформу легкою та ефективною, ідеальною для додатків реального часу з інтенсивним використанням даних, які працюють на розподілених пристроях. NodeJS використовує мову програмування JavaScript, яка розпочинала свій шлях як звичайна скриптова мова для анімацій у браузері, але за десять років успішного розвитку, набору популярності та удосконалення, вона вийшла на enterprise рівень саме за допомоги серверної платформи. За цей час навколо цієї платформи зібралось доволі обширна спільнота розробників і саме через це майже на кожну проблему завжди знайдеться відповідь на форму, або відповідний пакет, який вирішує задачу.

Цикл подій – це те, що робить Node.js унікальним середовищем для JavaScript. Цикл подій дозволяє виконувати неблокуючі операції вводу-виводу незважаючи на те, що JavaScript є однопотоковим. Це відбувається шляхом розвантаження операцій до ядра системи, коли це можливо. Оскільки більшість сучасних ядер є багатопоточними, вони можуть обробляти кілька операцій, що виконуються у фоновому режимі. Коли одна з цих операцій завершується, ядро повідомляє Node.js, щоб відповідний зворотний виклик міг бути доданий до черги опитування для виконання. Черга циклу подій складається з шести різний стадій. Кожна фаза має чергу FIFO зворотних

викликів для виконання. Хоча кожна фаза по-своєму особлива, для кожної із них можна виділити загальний алгоритм дій: коли цикл подій переходить у певну фазу, він виконує будь-які операції, характерні для цієї фази, а потім виконує зворотні виклики в черзі цієї фази, доки черга не буде вичерпана або не буде вичерпана максимальна кількість зворотних викликів. Після цього цикл переходить на наступної фази. Все це дозволяє NodeJS обробляти величезну кількість вхідної та вихідної інформації, але робить складним обробку важких обчислень. Оскільки наша система не обробляє велику кількість даних на сервері, а має швидко приймати та віддавати інформацію від усіх клієнтів, NodeJS є простим там ефективним вибором.

Оскільки наш статичний сервер має оброблювати запити використовуючи HTTP, потрібно обрати відповідний фреймворк для HTTP серверу. Наразі існує широкий вибір варіантів, але основним та перевіреним часом є express. Express – це мінімальний та гнучкий NodeJS фреймворк, який пропонує надійний набір рішень для серверний застосункі, а завдяки безлічі допоміжних методів HTTP і проміжного програмного забезпечення, створення надійного API є швидким і легким [14]. Основними перевагами express є

- Простота, мінімалістичність, гнучкість і масштабованість
- Швидкість розробка програми
- Гнучка конфігурація під будь-які потреби
- Низька крива навчання
- Проста інтеграція сторонніх служб і проміжного програмного забезпечення

Очевидно, що всі дані, які надходять мають десь зберігатися. Ідеальним варіантом для цього є SQL бази даних, через надійну структуру даних, оптимізацію та інтуїтивну роботу. Задля того, аби наша серверна система могла працювати з буль-якою SQL базою даних, було обрано популярну бібліотеку knex. Knex – це незалежний від бази даних будівельник SQL запитів за допомогою JavaScript. Він надає безліч функцій для роботу зі сховищем, які

комбінуються між собою, дозволяючи створювати складну логіку із невеличких атомів запитів[15]. Додатково до цього, knex обертає кожен запит у транзакцію, яка поверне вже внесені зміни при будь-якій помилці, надає можливість визначити пул підключень та надає зручний інтерфейс для побудови міграцій.

Для аутентифікації у застосунку було обрано відкритий стандарт JSON Web Key Sets, який дозволяє будь-якому сервісу перевірити підписаний алгоритмом RS256 токен, який був виданий авторизаційним сервером/сервісом. Для цього авторизаційний сервер надає сет публічних ключів, які можуть бути використані у перевірці валідності токена, виданого цим же сервером. Для реалізації цього нам потрібна бібліотека, яка дозволяє генерувати json файл з ключами, створювати на його основі токен, та перевіряти токен за допомогою публічних ключів. Під список наших вимог ідеально підходить бібліотека node-jose[16], яка підтримує усе вищевикладене.

2.3.2 Colyseus

Для взаємодії ігрових клієнтів у режими реального часу було обрано сучасний фреймворк для NodeJS – Colyseus[17]. Його особливість полягає у тому, що він інкапсулює логіку створення та лістингу кімнат за допомогою наслідування від базового класу Room, що дозволяє пропустити етап ручного створення та налаштування цього модуля програми, та зосередитися на ігровій логікі. Наступним величезним бонусом є частковий контроль над мережевим з'єднанням між сервером та клієнтом. Клієнт автоматично встановлює з'язок до серверу, та отримає оптимізовані оновлення ігрового стану, кожен раз коли він змінюється. При швидкій початковій розробці, цього має бути достатньо, аби написати MVP продукт, але фреймворк також надає можливість повного ручного контролю на підключенням та способом надсилання оновленого ігрового стану клієнтам.

Але головною особливістю цього фреймворку є використання найшвидшого типу підключення, який використовується у всій найсучасніших

іграх - uWebSockets.js. uWebSockets.js – реалізація сокетного підключення між клієнтами на мові програмування C++, який підтримується серверною платформою NodeJS. Це дозволяє суттєво збільшити швидкість передачі інформації між клієнтами. Для порівняння візьмемо найпопулярнішу бібліотеку для роботи із сокетами socket.io. Запустимо одночасно п’ятсот клієнтів та порівняємо кількість транзакцій на секунду. Як можна побачити на рис 2.5, нативна реалізація uWebSockets.js виграє у socket.io майже в 10 разів.

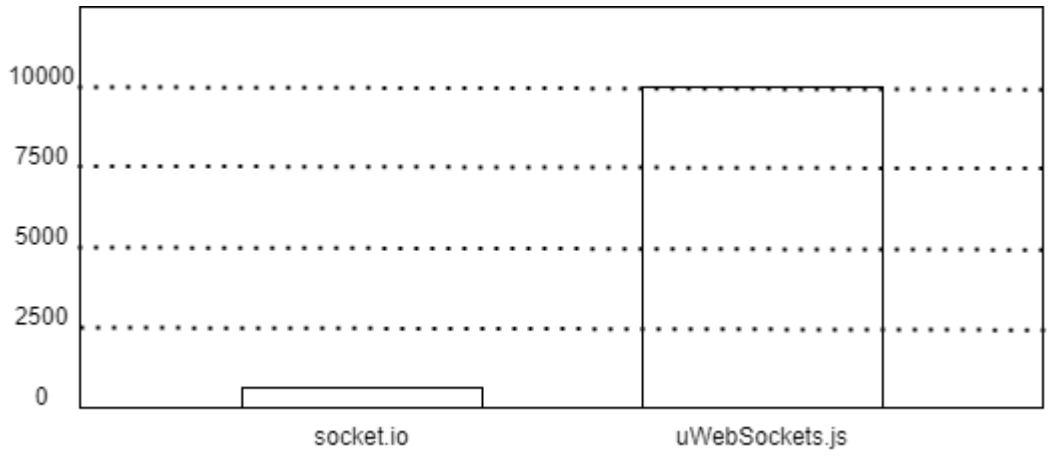


Рис 2.5

2.3.3 Docker

Docker - це програмна платформа для швидкої розробки, тестування та розгортання додатків. Він упаковує ПО в стандартизовані блоки, які називаються контейнерами. Кожен контейнер включає все необхідне для роботи програми: бібліотеки, Системні інструменти, код і середовище виконання. Завдяки Docker можна швидко розгортати і масштабувати додатки в будь-якому середовищі і зберігати впевненість в тому, що код буде працювати. Це надає розробникам і системним адміністраторам надійний і економічний спосіб збірки, доставки і запуску розподілених додатків будь-якого масштабу[18].

В основі роботи Docker лежить стандартизований спосіб виконання коду, який можна уявити як операційну систему для контейнерів. Подібно до

того як віртуальна машина створює віртуальне представлення апаратного забезпечення сервера (тобто усуває необхідність безпосередньо керувати таким), контейнери створюють віртуальне представлення серверної операційної системи. Після установки на кожен сервер Docker надає доступ до простих команд, необхідних для складання, запуску або зупинки контейнерів.

Використання цього програмного забезпечення дозволяє швидше і ефективніше доставляти або переміщати код, стандартизує виконувані додатками операції і в цілому економить кошти, оптимізуючи використання ресурсів. Завдяки Docker користувачі отримують об'єкт, який з високою надійністю можна запускати на будь-якій платформі. Простий і зрозумілий синтаксис забезпечує повний контроль над виконуваними операціями. Доставка додатків в невеликих контейнерах спрощує процес розгортання, виявлення проблем і відкату для їх усунення до минулих версій. Додатки на основі контейнерів Docker можна ефективно переносити з локальних машин, на яких ведеться розробка, в будь-які хмарні рішення, а також контейнери дозволяють виконувати більше коду на кожному сервері, підвищуючи ефективність використання ресурсів і скорочуючи витрати.

2.3.4 AWS

Amazon Web Services (AWS) – це дочірня компанія Amazon, яка надає оренду хмарних обчислень через свою платформу[19]. Послугами компанії можуть скористатися будь-які фізичні та юридичні особи за допомогою платної підписки. Все це досягається через технологію володіння віртуальним кластером розрахункових машин через мережу Інтернет. Все що знаходиться у фізичному комп'ютері притаманно і віртуальному - апаратні пристрої, відеокарта, процесор, жорсткий диск або SSD накопичував, оперативну пам'ять, тощо. На ці віртуальні машини можна встановлювати будь-які програми завдяки операційній системі, яка завчасно встановлена. Хостинг пропоную різні хаби по всьому світу задля кращого вибору локації, що в

сукупності з вищезгаданими функціями робить AWS ідеальним підходом для хмарних обчислень та розгортки.

ВИСНОВКИ ДО РОЗДІЛУ 2

В даному розділі були розглянути основні рішення під час проектування серверної програмної системи мобільної багатокористувацької асиметричної VR-гри. Були визначені основні задачі, які має реалізувати система, такі як:

- Авторизація через обліковий запис Facebook
- Перегляд та створення лобі
- Перегляд власного профілю та минулих сесій
- Встановлення зв'язку між гравцям
- Розподіл ролей та старт гри
- Обробка ігрових обчислень
- Повторне підключення

Було визначено, що система складається із двох модулів – статичного, той що відповідає за представлення статичної інформації за потреби, і модуля ігрового сервера, який взаємодіє з клієнтами у реальному часі.

Були розглянуті основні можливі мережеві рішення, базуючись на функціях розроблювального додатку. В результаті аналіз серед наявних аналогів, було вибрано клієнт-мультисерверну архітектуру, яка, не дивлячись на її недоліки, повністю задовольняє потреби розроблювального додатку.

Також були розглянуті основні інструменти розробки, якими стали:

- NodeJS – серверна платформа побудована на середовищі виконання Chrome JavaScript V8 для легкого створення швидких і масштабованих мережевих програм, які залежать на велику кількість вводу/виводу.
- Express – бібліотека для реалізації HTTP серверу, маршрутів та проміжних обробників.
- Knex – бібліотека для роботи з SQL базами даних, абстрагована від конкретної БД.
- node-jose – бібліоткека для роботи з JWKS ключами.

- Colyseus – ігровий фреймворк для NodeJS, який спрощує роботу з мережею та ігровими кімнатами, інкаплусюючи дану логіку
- Docker - програмна платформа для швидкої розробки, тестування та розгортання додатків
- AWS – сервіс хмарний обчислень для розгортання, масштабування та хостингу програм

РОЗДІЛ 3
РЕАЛІЗАЦІЯ СЕРВЕРНОЇ СИСТЕМИ МОБІЛЬНОЇ БАГАТОКОРИСТУВАЦЬКОЇ АСИМЕТРИЧНОЇ VR ГРИ

3.1 Статичний модуль серверної системи

Для роботи зі статичним модулем програми було створено пустий проект за допомогою пакетного менеджера yarn та команди yarn init. Ця команда створила файл package.json, який містить усю інформацію про наявні залежності, можливі команди та загальну інформацію про проект. Далі було додано файл tsconfig.json, який виставляє параметри для компіляції типобезбечного TypeScript у чистий Javascript переде виконанням коду програми. З додаткових інструментів було додано eslint та prettier, які дозволяють досягнути однакого вигляду коду при подальшій модифікації проекту іншими розробниками, та уникнути можливих проблем, таких як забута крапка з комою, або не закриті дужки.

Сама система була побудована за допомогою підходу MVC (Model View Controller), який дозволяє швидко та ефективно масштабувати програму, або редагувати існуючі рішення. У цьому підході ключовим є 3 компоненти: репозиторій, сервіс та контролер. Репозиторій є абстракцією, яка працює напряму з даними, які зберігаються у базі, та є повністю незалежною від інший компонентів програми. В нашій системі було створено базовий репозиторій, який агрегує основні можливі функції запитів до бази даних, але також дозволяє використати наслідування та додати будь-які складніші або специфічні за логікою запити, що зменшує кількість коду та додає гнучкості у розробці запитів (рис 3.1).

Рис 3.1 Код базового репозиторію

Сервіс – це невеличкі функції коду, які інкапсулюють у собі усю бізнес логіку, можливі обчислення та роботу з репозиторіями. Кожен сервіс має відповідати за одну конкретну дію, для більш кращої модульності коду. Сервіси отримують усі потрібні для роботи за даними репозиторії через ін'єкцію залежностей через параметри конструктора. Контролер є основним обробником HTTP запитів, які приходять на конкретний маршрут. Вони отримують усі потрібні дані з параметрів запиту, валідують їх, та делегують на обробки до потрібного сервісу. Контролер може мати будь-яку потрібну кількість сервісів для оброки даних.

Відповідно до наших вимог, система має реалізувати три основні задачі – авторизація та аутентифікація користувачів, можливість перегляду власної інформації профілю та перегляд минулих або поточних ігрових сесій. Для цього було створено три підмодуля програми – Auth, що відповідає за авторизація аутентифікацію, Users, який має доступ до приватної інформації користувача та Sessions, який зберігає та модифікує ігрові сесії. Для кращого

розуміння принципу роботи системи, далі буде розглянутий кожен з цих підмодулей.

Auth модуль реалізує відкритий стандарт авторизації (JWKs) за допомогою токенів, а також дає можливість клієнту авторизуватися через обліковий запис Facebook. За авторизацію через Facebook відповідає клас AuthController, який отримає на вхід спеціальний код від користувача, який дозволяє отримати доступ до даних, які зазначені у цьому коді. Клас AuthController викликає низьку сервісів, для отримання інформації та збереження її до бази даних, а також генерує власний токен авторизації і віддає його клієнту. Декларація цього класу зазначена на рис 3.2.

```
export class AuthController extends AuthControllerWithServices {
    async find(req: any, res: Response) {
        try {
            const { code } = req.query;

            const info = await this.authService.getFacebookInfo(code);

            const user = await this.authService.createUser(info);

            const token = await this.authService.createToken(user);

            await this.authService.saveUserToken(token.toString(), user.id);

            return res.status(200).json(token);
        } catch (e) {
            return res.status(500).send(e);
        }
    }
}
```

Рис 3.2 Декларація класу AuthController

Наступним підмодулем нашої системи є Users, який дає можливість подивитися на список користувачів, отримати повну інформацію про клієнта, який робить запит, відредагувати власну інформацію або видалити свій акаунт. Клас UserController реалізує стандартний набір операцій CRUD через відповідні методи find, get, patch та remove (рис 3.3), та потребує Authorization

Bearer токен, створений AuthController, у заголовках запиту задля безпеки та перевірки прав на редагування.

```
export class UserController extends UserControllerWithServices {
    async find(_: any, res: Response) {···
    }

    async get(req: any, res: Response) {···
    }

    async patch(req: any, res: any) {···
    }

    async remove(req: any, res: any) {···
    }
}
```

Рис 3.3 Методи класу UserController

Останнім підмодулем є Sessions. Він складається з двох контролерів – стандартного контролера, який, аналогічно UserController відповідає за надання CRUD операцій над сесіями, та SessionsStateController, який дозволяє нашій ігровій системі у реальному часі створювати та оновлювати інформацію про сесії та її гравців. SessionsStateController приймає запити тільки від відповідного хосту, який був заданий у конфігурації, а також потребує відповідного токену від іншої системи, задля покращення безпеки.

Відповідно до визначеного функціоналу, система підтримує певний набір HTTP запитів, кожен з яких відповідно до RESTful підходу має унікальний ідентифікатор у вигляді URI. Перелік запитів, потрібних параметрів та унікальних ідентифікаторів наведено у таблиці 3.1

Таблиця 3.1 – Опис можливих HTTP запитів

Метод	URI	Параметри	Опис
GET	/api/v1/auth/social	?code – унікальний код від Facebook для авторизації	Повертає токен для авторизації
GET	/api/v1/jwks/public	-	Надає публічний список токенів для перевірки токену клієнта
GET	/api/v1/sessions	-	Повертає загальний список сесій
GET	/api/v1/sessions/:id	?id – унікальний ідентифікатор шуканої сесії	Повертає сесію за унікальним ідентифікатором
POST	/api/v1/sessions/start	-	Створює сесію та повертає її унікальний ідентифікатор
POST	/api/v1/sessions/users/add	userIds – список користувачів які приймали участь у сесії sessionId -унікальний ідентифікатор сесії	Додає список користувачів для сесії
POST	/api/v1/sessions/finish	sessionId - унікальний ідентифікатор сесії wonSide – сторона що виграла	Успішно закінчує сесії та оновлює дані про неї
POST	/api/v1/sessions/abort	sessionId - унікальний ідентифікатор сесії	Закінчує сесію з помилкою
GET	/api/v1/users	-	Повертає список користувачів
GET	/api/v1/users/:id	id – унікальний ідентифікатор шуканого користувача	Повертає користувача за унікальним ідентифікатором

Продовження таблиця 3.1

Метод	URI	Параметри	Опис
PATCH	/api/v1/users/:id	id – унікальний ідентифікатор обновлюваного користувача firstName lastName email	Оновлює користувача за унікальним ідентифікатором
DELETE	/api/v1/users/:id	id – унікальний ідентифікатор користувача, що видаляється	Видаляє користувача за унікальним ідентифікатором

Кінець таблиці 3.1

3.2 Ігровий модуль серверної системи у реальному часі

Для роботи зі ігровим модулем програми також було створено пустий проект, аналогічно статичному модулю, та додані відповідні конфігураційні файли, такі як package.json, tsconfig.json, eslint та prettier разом із потрібними залежностями із пакетного хмарного сховища npm.

Система поділена на три основних компонента – обчислення стану, обробка кімнати та ігрова логіка. Відповідно до документації Colyseus було створено клас кімнати – FMTSRoom (рис 3.4), який відповідає за інкапсуляцію ігрової логіки, підключення гравців в межах однієї кімнати та обробки повідомлень клієнту. Клас кімнати автоматично підтримує з'єднання та надає інформацію про усі створенні кімнати клієнту при запиті. Серед основних методів варто відокремити onCreate, onJoin та onLeave. Кожен з цих методів викликається відповідно до запитів клієнтського SDK.

```
export class FMTSRoom extends Room<GameState> {
    private roomOptions: any;
    private serverTime = 0;
    public sessionId = 0;

    constructor() {
        super();
    }

    async onCreate(options: any) {…
    }

    onJoin(client: Client, options: any) {…
    }

    async onLeave(client: Client, consented: boolean) {…
    }
}
```

Рис 3.4 Клас ігрової кімнати

onCreate метод викликається при створені нової ігрової кімнати, та ініціалізує ігрові правила кімнати, створює сесію за допомогою виклику до статичного серверу, реєструє стан гри, який буде оновлюватися, реєструє обробники клієнтських повідомлень та запускає основний ігровий цикл. Цей метод приймає один аргумент options – особливі налаштування кімнати, які можуть бути використані сервером під час подальших обчислень.

onJoin метод викликається після виклику onCreate, або під час підключення нового гравця до кімнати. В нашій системі цей метод створює новий об’єкт стану User відповідно до типу підключення та додає його до колекції гравців для подальшого використання.

onLeave метод викликається при відключенню або втрати зв’язку між клієнтом та сервером. Цей метод перевіряє, чи клієнт відключився сам і якщо так, то видаляє його з колекції об’єктів гравців. Якщо ж клієнт не навмисно

втратив зв'язок, сервер намагається перепідключитися протягом визначеного часу і при невдачі видаляє клієнта з колекції гравців.

Стан гри буде доступний під час усього ігрового циклу, дозволяючи серверу змінювати його відповідно до інформації від клієнтів, або ж на свій розсуд зважаючи на обчислення. При будь якій зміні стану сервер звіряє нову інформацію з минулим станом, та надсилає оновлену інформацію усім клієнтам, але не частіше аніж визначена зміна PatchRate. Це дозволяє збирати оновлення у невеликі групи – батчі, та ефективніше оновлювати стан клієнту оптимізуючи використання мережевих ресурсів. Основною точкою входу у стан серверу є клас GameState, який тримає в собі інформацію про підключених мобільних клієнтів, VR-гравця, правила гри, а також поточний стан ігрового циклу, зображений на рис 3.5 Клас VRUserState відповідає за положення VR гравця у просторі, стан його підключення та стан синхронізації. Клас MobileUserState тримає в собі інформацію про позицію мобільного гравця та додаткову інформацію.

```
export class GameState extends Schema {
    @type({ map: MobileUserState }) mobileUsers =
        new MapSchema<MobileUserState>();

    @type(VRUserState) vrUser!: VRUserState;

    @type('string') gameState = GameStateEnum.WAITING;
    @type('string') previousGameState = GameStateEnum.END_ROUND;
    @type(GameRulesSchema) gameRules!: GameRulesSchema;
}
```

Рис 3.5 Клас ігрового стану

За симуляцію ігрової логіки відповідає функція gameLoop, яка запускається за допомогою методу setSimulationInterval під час створення кімнати та викликається один раз у заданих інтервал. Ігровий цикл знаходить поточний ігровий стан, який складається із п'яти можливих варіантів: NONE, START_ROUND, WAITING, SIMULATE_ROUND та END_ROUND. Відповідно до знайденого значення, викликається одна з обробних функцій,

яка отримує екземпляр класу кімнати та доступ до усій публічних методів та полів. Функція gameLoop зображена на рис 3.6. Про реалізацію кожного з обробних методів буде написано далі у цьому розділі.

```
export const gameLoop = function (
    roomRef: FMTSRoom,
    deltaTime: number,
    currentTime: number,
    roomOptions: any
) {
    // Update the game state
    switch (getGameState(roomRef)) {
        case GameStateEnum.NONE:
            break;
        case GameStateEnum.START_ROUND:
            startGameRound(roomRef);
            break;
        case GameStateEnum.WAITING:
            waitingGameLogic(roomRef, roomOptions);
            break;
        case GameStateEnum.SIMULATE_ROUND:
            simulateRoundLogic(roomRef, currentTime);
            break;
        case GameStateEnum.END_ROUND:
            endRoundLogic(roomRef);
            break;
        default:
            break;
    }
};
```

Рис 3.6 Функція ігрового циклу

waitingGameLogic – метод який очікує на підключення усіх гравців їхнє підтвердження готовності до гри. Мінімальна кількість людей для гри – двоє, один з VR частини, один з мобільною. Після того як мінімальна кількість гравців підключилася, та система отримала сигнали готовності від кожного гравця, waitingGameLogic викликає допоміжну функцію updateGameState, яка оновлює поточний стан на START_ROUND та зберігає попередній.

startGameRound – метод який агрегує інформацію про гравців та робить асинхронний HTTP до статичного серверу, надсилаючи інформацію про гравців у поточній сесії. Після цього за допомогою методу broadcast, кожен клієнт отримує повідомлення «beginRound», кімната використовуючи метод lock забороняє нові підключення (тільки повторні при втраті зв'язку) та прибирає кімнату зі списку до закінчення раунду. Кінець кінцем, викликається допоміжна функція updateGameState, і гра переходить до наступного стану.

simulateRoundLogic – основний метод, який на кожному оновленні серверу перевіряє чи продовжувати сесію, чи пора її завершати. Якщо VR гравець має більш аніж нуль очок HP, або змінна HP у хоча б одного із мобільних гравців більше нуля, то раунд продовжується, а ці умови перевіряються уже на наступному етапі. В іншому випадку викликається допоміжний метод updateGameState та ігровий цикл переходить до стану END_ROUND.

endRoundLogic – під час виклику цього методу, сервер обчислює яка сторона програла, надсилає кожному клієнту інформацію про закінчення раунду за допомогою повідомлення onRoundEnd. Після цього, викликається функція API, яка звертається до статичного серверу та надсилає йому дані про завершення сесії, а саме яка сторона виграла і коли це сталося. Коли всі дані були успішно збережені, ігровий цикл починається заново, до моменту як всі клієнти не покинуть кімнату.

3.3 Розгортання сервісів на хмарний ресурсах

Для розгортки сервісів на хмарному сервері AWS, спочатку потрібно докеризувати кожен із модулів. Для цього було створено відповідно два схожих Dockerfile і описано спосіб запуску контейнеру, який можна побачити на рис 3.7

```
FROM node:latest
WORKDIR /usr/static-server
COPY package.json .
RUN yarn
COPY . .
RUN yarn build
CMD ["node", "./build/index.js"]
```

Рис 3.7 Вміст Dockerfile

Після цього було створено хмарний VPS сервер на AWS за допомогою EC2, та отримано до нього доступ за допомогою SSH ключа та відповідно файлу сертифікату (рис 3.8). Тепер підключитися до серверу можна за допомогою команди shh -i “file.pem” vsp-adress.

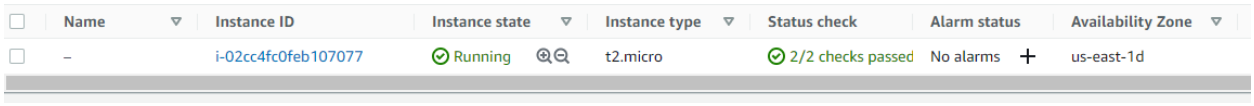


Рис 3.8 Запущений EC2 хмарний сервер

Далі для того щоб запустити наші сервери, був стягнутий вихідний код з нашого github репозиторію та запущена команда docker build для створення образу кожного із застосунка. Після цього, за допомогою команди docker run, були запущені наші контейнери, які будуть доступні на портах 8080 і 3000.

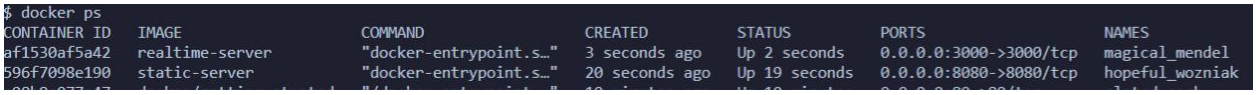


Рис 3.8 Список запущених контейнерів

3.4 Балансування навантаження на ігрових серверах

Задля покращення використання хмарних ресурсів, був розроблений балансувальний сервер, який злідкує за кількістю створених кімнат, та при виходженню за якийсь ліміт, створює екземпляр сервера у реальному часі. Балансування між екземплярами створених серверів відбувається за допомогою файлу nginx, який динамічно оновлюється під час нового старту екземпляру серверу. Щоб уникнути періоду downtime, було створено кластер nginx, який дозволяє динамічно вносити правки до файлу конфігурації головної, а потім за допомогою команди nginx-sync синхронізувати усі дочірні

і уникнути downtime. Якщо ж балансувальний сервер розуміє, що додаткові потужності вже не потрібні через невелику кількість лобі, він приймає рішення щодо згортки додаткових серверів через bash скрипт.

Щодо врегулювання ситуації, коли клієнти в середині однієї кімнати підключені до різних серверів, при створені серверу було додано параметр presence, що дозволяє увімкнути у фреймворку Colyseus механізм синхронізації даних між різними екземплярами за допомогою допоміжної база даних у реальному часі – Redis.

ВИСНОВКИ ДО РОЗДІЛУ 3

В рамках цього розділу було описано реалізацію двох модулів програми – статичних та ігровий модуль у реальному часі, розглянуто їх взаємодію. Обидва модулі були написані мовою TypeScript, з використанням низькі додаткових залежностей, таких як Knex, node-jose, express для статичного модуля та colyseus у модулі в реальному часі.

В результаті реалізації статичного модуля, клієнт отримав можливість авторизуватися та отримувати токен, переглядати та редагувати власний профіль, отримувати власні сесії та інформацію о користувачах в сесії. Модуль у реальному часі отримав можливість створювати нові сесії, додавати туди користувачів та завершати або переривати їх.

Ігровий модуль у реальному часі був імплементований способом реалізації класу кімнати FMTSRoom та основного ігрового стану класу GameState.

Важливим моментом розробки стала розгортка серверних застосунків на сервері, що дало відкритий доступ до них у мережі інтернет. Це було реалізовано за допомогою сервісу хмарних обчислень AWS та інструментом контейнеризації Docker.

Останнім, але не менш важливим став модуль балансування на ігрових серверах, який динамічно додає та прибирає створені екземпляри ігрового серверу в залежності від завантаження, а увесь трафік рівномірно розподіляється між екземплярами за допомогою nginx. Проблема клієнтів у одній кімнати підключених до різних серверів була вирішена за допомогою бази даних у реальному часі Redis, яка дозволяє синхронізуватися між різними екземплярами.

В результаті імплементації було створено легко розширювану, оптимізовану та ефективну серверну систему для мобільної

багатокористувацької асиметричної VR гри, що відповідає всім вимогам, які були поставлені вище.

РОЗДІЛ 4
ОГЛЯД СТВОРЕНОЇ СЕРВЕРНОЇ СИСТЕМИ МОБІЛЬНОЇ БАГАТОКОРИСТУВАЦЬКОЇ АСИМЕТРИЧНОЇ VR ГРИ

Для тестування отриманого API використаємо Postman[20] і розглянемо запити для статичного модулю нашої програми.

Першим розглянемо підмодуль Auth. Для отримання токену клієнтом треба зробити запит на «/api/v1/auth/social» з параметром code, який видається користувачу після входу в свій акаунт Facebook (рис 4.1).

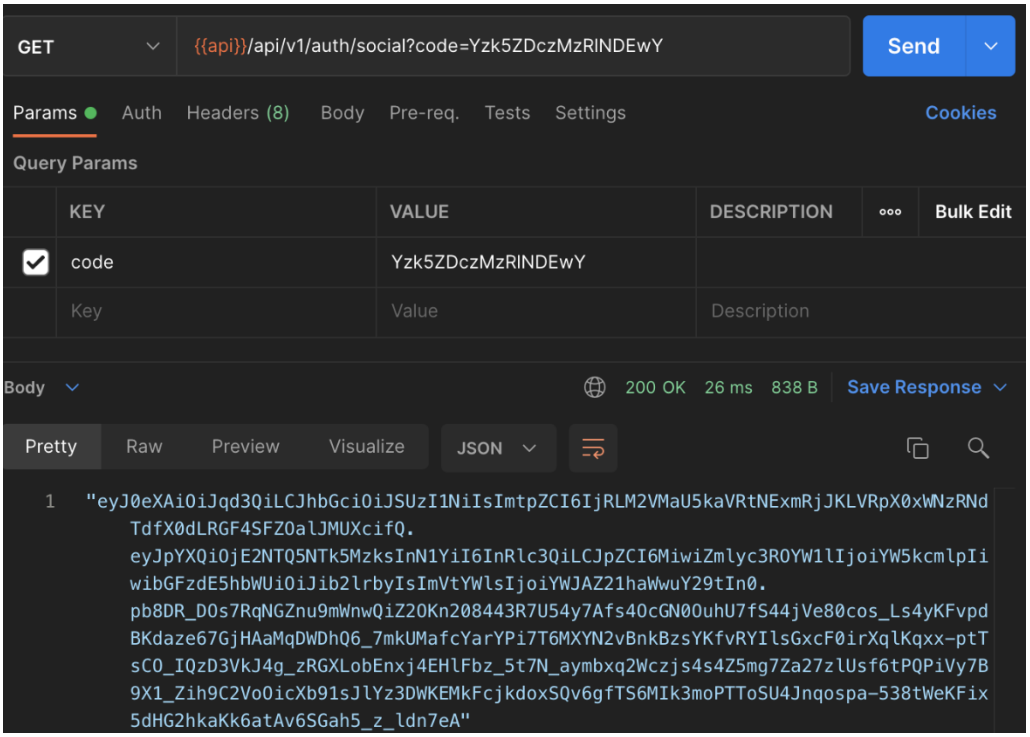


Рис 4.1 Запит на отримання токену для входу

Для інших сервісів для успішної валідації клієнтського токену, треба отримати відповідну колекцію публічних ключів, що зберігається на сервері. Для цього є ендпоінт «/api/v1/jwks/public», який є публічним та повертає потрібні ключі JWKS клієнту (рис 4.2).

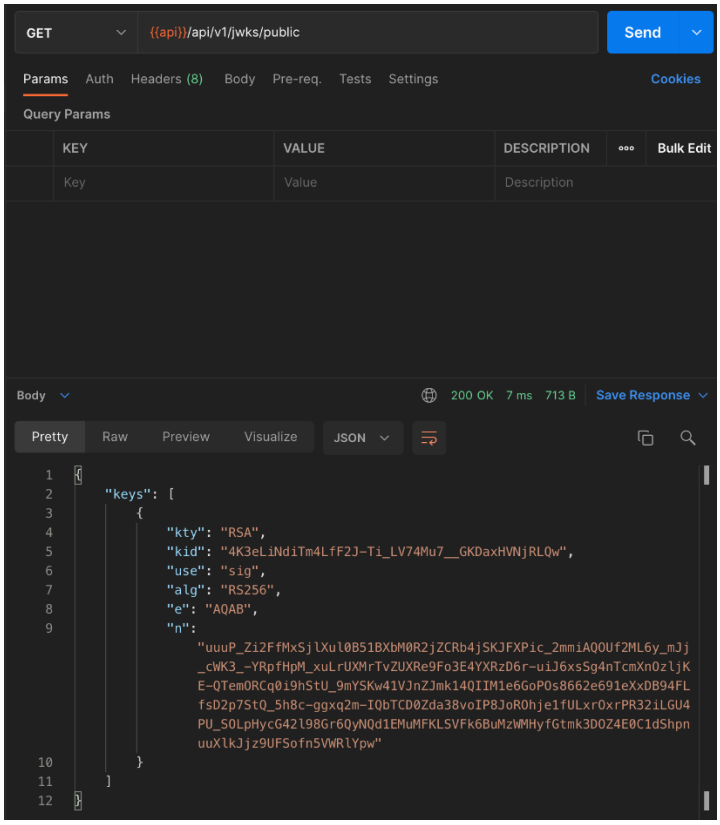


Рис 4.2 Отримання публічних ключів для перевірки токену

Далі розглянемо підмодуль Sessions. Для створення нової сесії нашим модулем ігровим модулем, можна зробити запит з відповідним токеном до «/api/v1/sessions/start», отримавши у результаті унікальним ідентифікатор сесії (рис 4.3).

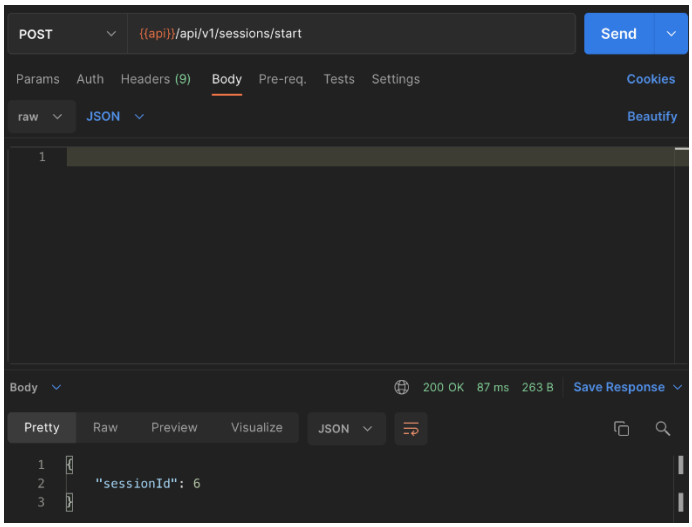


Рис 4.3 Початок сесії

Отримавши унікальний ідентифікатор сесії, можна модифікувати сесію, додаючи до неї користувачів (рис 4.4).

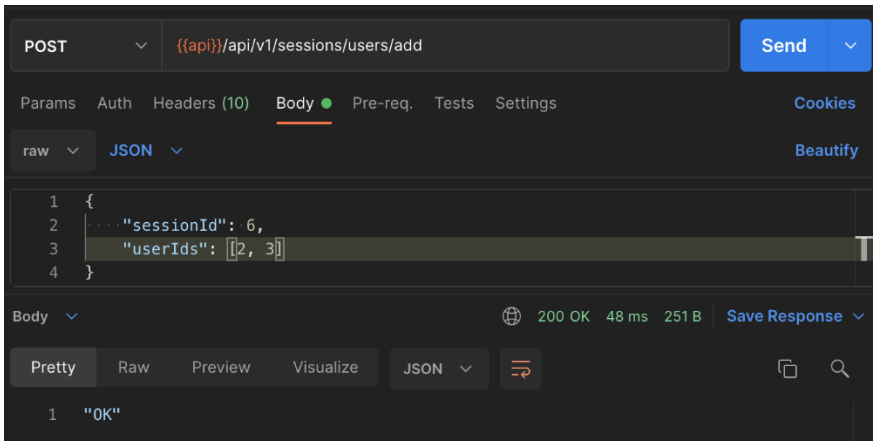


Рис 4.4 Додавання користувачів до сесії

Після завершення сесії у ігровій системи, потрібно сповістити статичну систему відповідним запитом на «/api/v1/sessions/finish», передаючи в тілі запиту унікальний ідентифікатор сесії та сторону, яка виграла (рис 4.5).

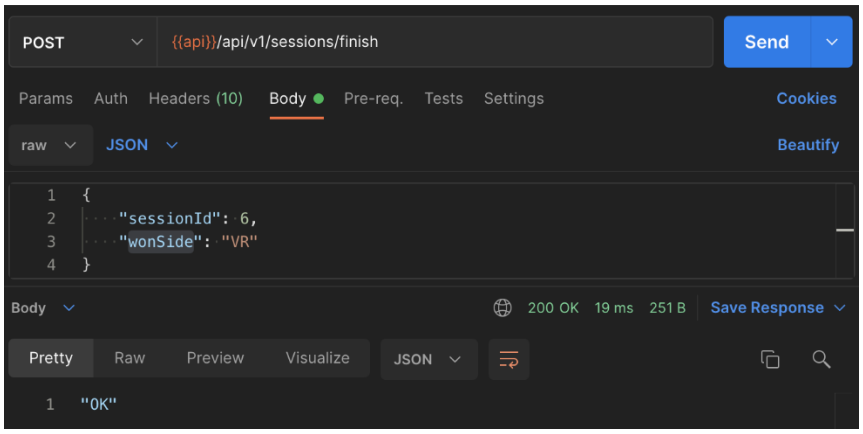


Рис 4.5 Завершення сесії по унікальному ідентифікатору з виграною стороною

Після успішного логіну клієнт може продивитися весь список сесій, зробивши запит до «/api/v1/sessions», отримавши у результаті базовий набір атрибутів (рис 4.6).

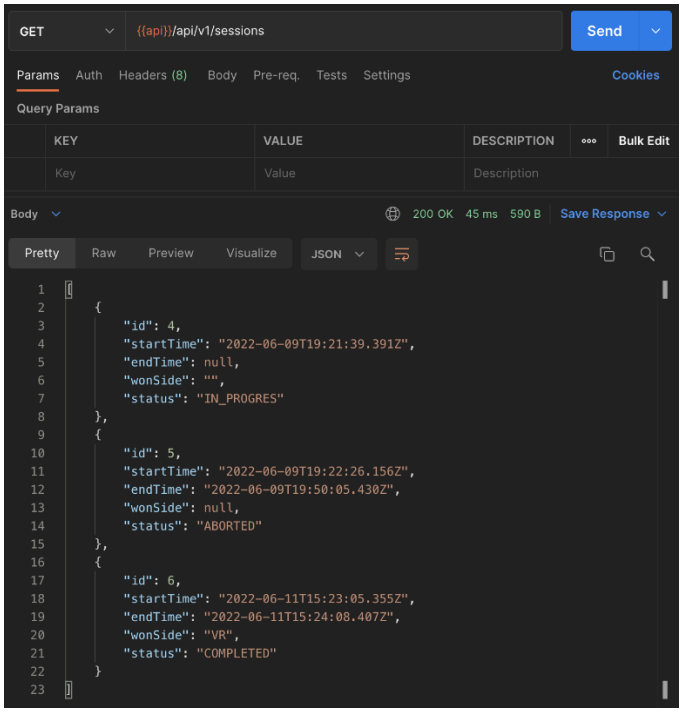


Рис 4.6 Перегляд усіх наявних сесій

Також клієнт має можливість більше детально передивитися сесію за унікальним ідентифікатором, задля отримання додаткової інформації. Для цього до запиту «/api/v1/sessions», потрібно передати унікальний ідентифікатор (рис 4.7).

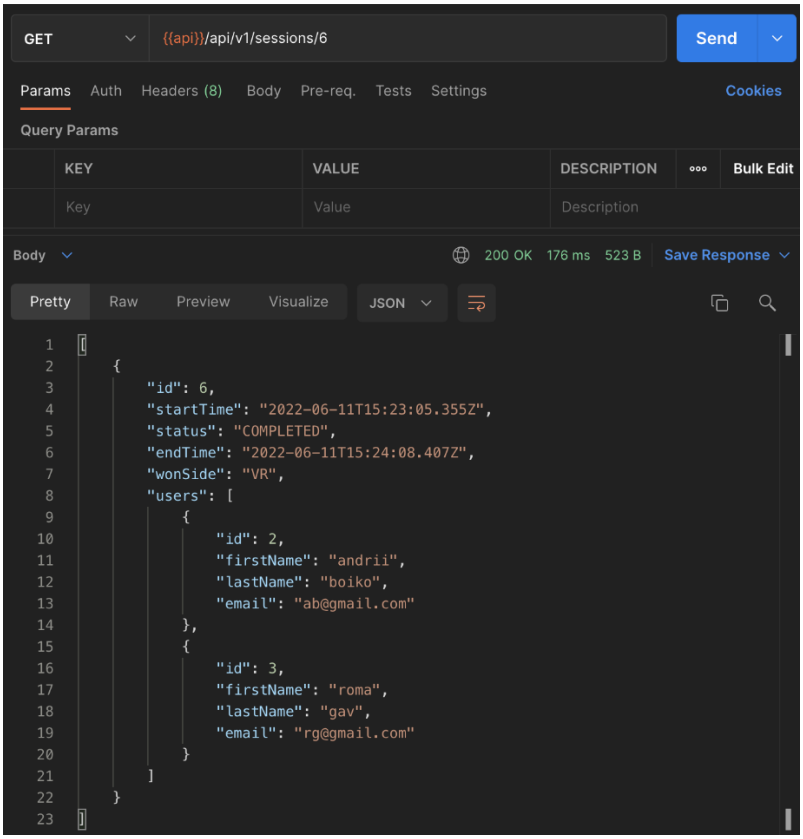


Рис 4.7 Перегляд сесії з додатковою інформацією за унікальним ідентифікатором

Останнім підмодулем нашої статичної програми є Users, у якого перегляд усіх користувачів, та перегляд конкретного користувача за унікальним ідентифікатором є аналогічним до підмодулю Sessions. Додатково до них є запит на редагування інформації у користувача у системі (рис 4.8).

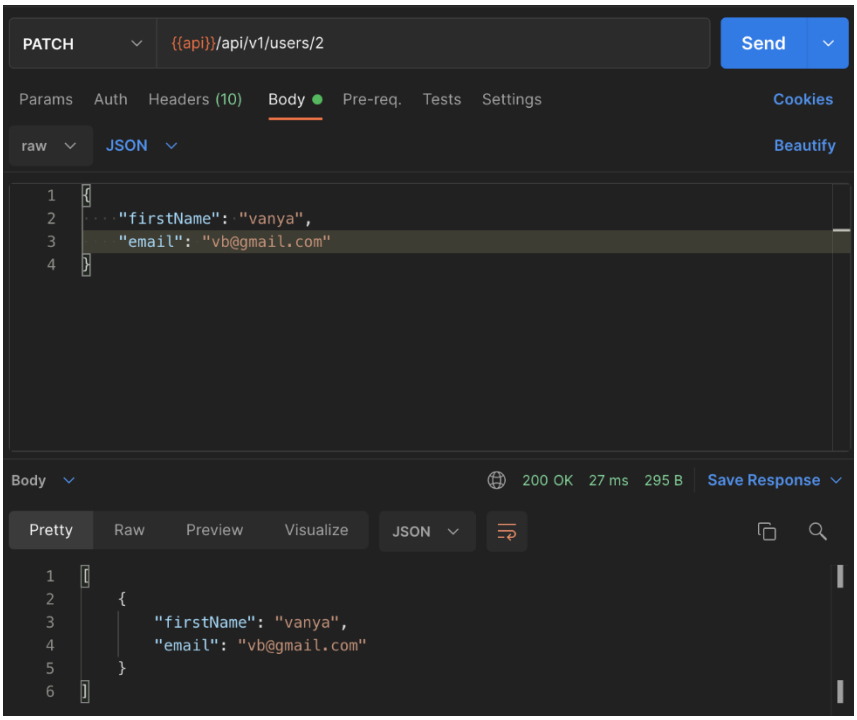


Рис 4.8 Редагування інформації користувача

Останнім можливим запитом є видалення користувача із системи, що видаляє усю інформацію, включаючи гравця у сесіях (рис 4.9).

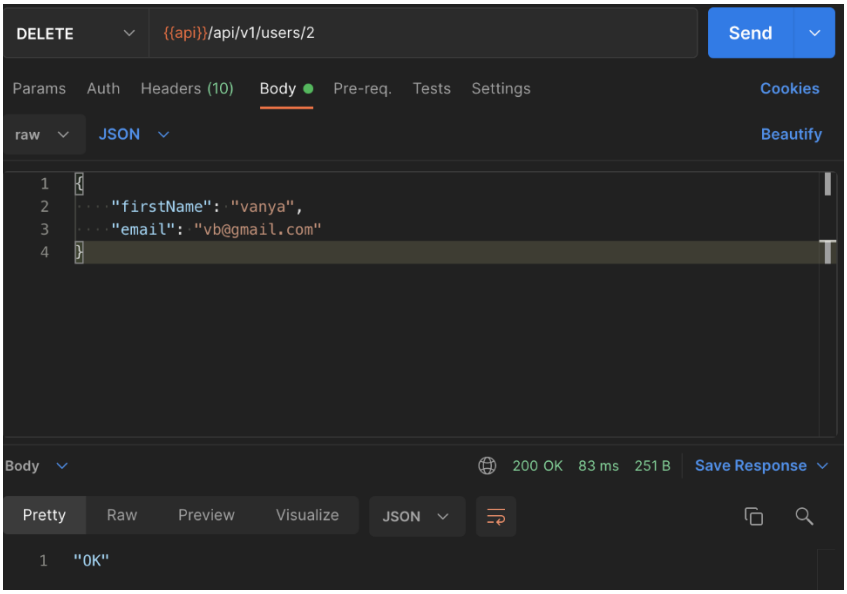


Рис 4.9 Видалення інформації про користувача

Для огляду можливостей нашого ігрового серверу, був створений невеликий тестовий застосунок, який реалізує базові функції серверу.

Після авторизації через статичний сервер, користувачу пропонується вибрати сторону, за яку він буде грати (рис 4.10).

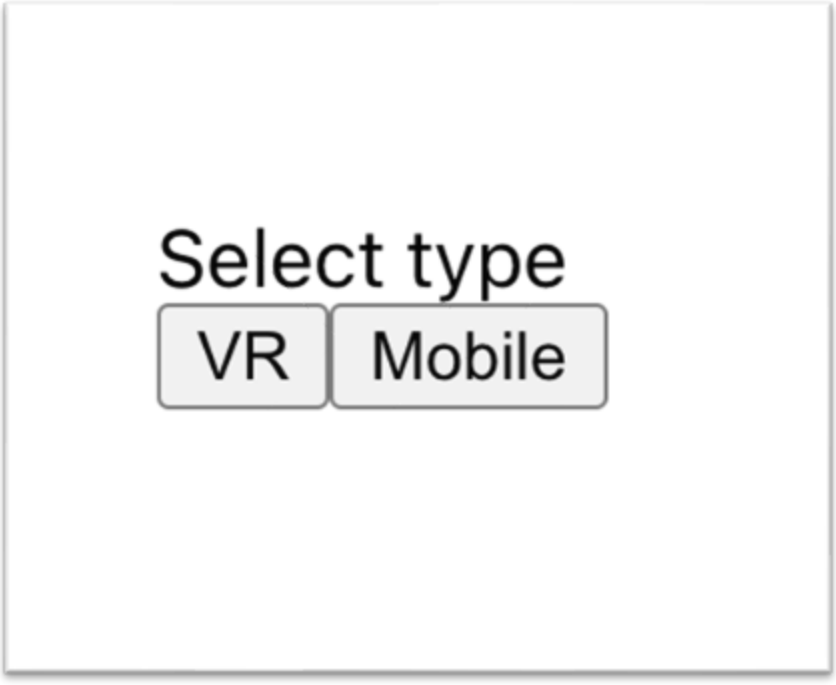


Рис 4.10 Вибір ігрової сторони

Далі, користувач може переглядати вже існуючі лобі, або створювати свої власні з автоматичним підключенням до нього (рис 4.11).

Рис 4.11 Вибір існуючого лобі та створення нового

Після створення або підключення до лобі, користувач має підтвердити свою готовність щодо участі у раунді, натиснувши на кнопку Ready (рис 4.12).

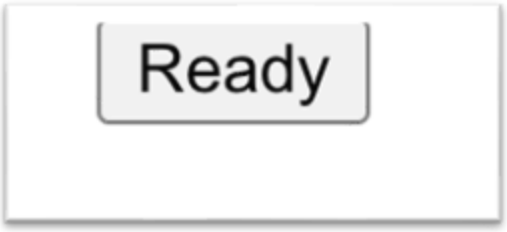


Рис 4.12 Кнопка підтвердження готовності

Коли усі гравці натиснули кнопки готовий, починається раунд гри. Кожен з гравців має поточну інформацію про стан кожного з гравців, власний стан та можливі дії (рис 4.13).

Рис 4.13 Поточний стан гри

Гравці, виконуючи дії, обмінюються інформацією, яка у режимі реального часу відправляється на сервер та передається кожному із гравців. Клієнтська частина відповідно оновлює усі дані (рис 4.14)

Рис 4.14 Стан гри після оновлень від гравців

Після вичерпання часу на раунд, або виграшу однією із сторін, система сповіщає про закінчення раунду, та повертається у початковий стан готовності кожної зі сторін (див рис 4.11).

.

ВИСНОВКИ ДО РОЗДІЛУ 4

В цьому розділі представлений огляд використання серверної систему багатокористувацької асиметричної VR гри, створеної в рамках цієї дипломної роботи. Були розглянуті усі сценарії використання статичного модулю програми з використанням програми для тестування серверних застосунків Postman, такі як: авторизація, перегляд та маніпуляція з сесіями, перегляд та редагування сесій.

Також було розглянуто основні функції використання ігрового модулю у режимі реального часу за допомогою тестового клієнту, а саме: перелік ігрових лобі, створення та підключення до лобі, готовність до матчу, старт гри, перелік підключених клієнтів та їх ігрові стани, зміну стану та оновлення інформації у режимі реального часу, закінчення сесії та запуск ігрового циклу заново.

У результаті цей розділ показує, що система реалізована у третьому розділі має весь необхідний функціонал, що був визначений, є доступною та легко масштабованою, надає швидку та зручну взаємодію.

ВИСНОВКИ

Метою даної дипломної роботи було створення серверної програмної системи мобільної багатокористувацької асиметричної VR гри.

У першому розділі було розглянуто асиметричний ігровий процес у класичних комп'ютерних іграх, визначені основні плюси та мінуси. Були визначені особливості асиметричного ігрового процесу саме у VR іграх, після чого було детально розглянуті та описані 2 найпопулярніші асиметричні ігри у віртуальній реальності. В ході аналізу було виявлено ряд недоліків та переваг. Головною ціллю розробки було уникнути існуючих недоліків та додати головні переваги із додаткових функціоналом.

В другому розділі були розглянуті основні задачі, які має забезпечувати серверна система, поділено систему на 2 окремих модуля – статичний та модуль у режимі реального часу. Було обґрунтовано вибір мережевої архітектури, а саме клієнт-мультисерверної. Також було обрано основні інструменти реалізації, якими стали NodeJS, з додатковим вибором бібліотек express, knex та node-jose, Colyseus, Docker та сервіс для хмарних обчислень AWS.

Третій розділ розкриває деталі імплементації серверної системи мобільної багатокористувацької асиметричної VR гри, а саме її статичний модуль та модуль у режимі реального часу. Було продемонстровано реалізацію швидкого розгортання до хмарного сервісу та динамічне масштабування в залежності від навантаження.

Заключний четвертий розділ демонструє принцип та результат роботи кожної частини серверної системи, основний наявний фунціонал.

Отже описана у дипломному проекті серверна програмна система мобільної багатокористувацької асиметричної VR гри реалізує поставлені у першому розділі вимоги, уникає недоліки притаманні існуючим рішенням, включає в собі додатковий функціонал. У результаті існує чимало

можливостей для подальшого розвитку та модифікації даної системи, а саме: додавання додаткової логіки, обробки додаткових даних, голосового або текстового чату.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Fullerton T. Game design workshop: a playcentric approach to creating innovative games. 2nd ed. Morgan Kaufmann, 2008. 496 p.
2. Mastermind. BoardGameGeek. URL: http://boardgamegeek.com/boardgame/2392/mastermind
3. Gandolfi E. You have got a (different) friend in me: asymmetrical roles in gaming as potential ambassadors of computational and cooperative thinking. E-Learning and digital media. 2018. Vol. 15, no. 3. P. 128—145. URL: https://doi.org/10.1177/2042753018757757
4. Koetsier J. VR needs more social: 77% of virtual reality users want more social engagement. Forbes. URL: https://www.forbes.com/sites/johnkoetsier/2018/04/30/virtual-reality-77-of-vr-users-want-more-social-engagement-67-use-weekly-28-use-daily/?sh=372d871618fc
5. Rubin P. The oculus quest finally sets VR free. Wired. URL: https://www.wired.com/review/oculus-quest/
6. The shared view paradigm in asymmetric virtual reality setups / R. Horst et al. I-com. 2020. Vol. 19, no. 2. P. 87—101. URL: https://doi.org/10.1515/icom-2020-0006
7. OAuth 2.0 — OAuth. OAuth Community Site. URL: https://oauth.net/2
8. Pantel L., Wolf L. C. On the impact of delay on real-time multiplayer games. The 12th international workshop, Miami, Florida, USA, 12—14 May 2002. New York, New York, USA, 2002. URL: https://doi.org/10.1145/507670.507674
9. Liang D. An implementation of multiplayer online game with distributed server architecture. URL: http://ro.uow.edu.au/theses/633
10. Pellegrino J. D., Dovrolis C. Bandwidth requirement and state consistency in three multiplayer game architectures. The 2nd workshop, Redwood City,

California, 22—23 May 2003. New York, New York, USA, 2003. URL: https://doi.org/10.1145/963900.963905

11. An efficient synchronization mechanism for mirrored game architectures / E. Cronin et al. Multimedia tools and applications. 2004. Vol. 23, no. 1. P. 7—30. URL: https://doi.org/10.1023/b:mtap.0000026839.31028.9f
12. Scalable distributed simulation of large dense crowds using the real-time framework (RTF) / O. Scharf et al. Euro-Par 2010 - parallel processing. Berlin, Heidelberg, 2010. P. 572—583. URL: https://doi.org/10.1007/978-3-642-15277-1_54
13. Node.js. Node.js. URL: https://nodejs.org/en/
14. Express - Node.js web application framework. Express - Node.js web application framework. URL: https://expressjs.com/
15. SQL query builder for javascript | knex.js. SQL Query Builder for Javascript | Knex.js. URL: http://knexjs.org/
16. GitHub - cisco/node-jose. GitHub. URL: https://github.com/cisco/node-jose
17. Multiplayer server | colyseus: simple & fast multiplayer game creation. Colyseus. URL: https://colyseus.io/
18. Home - docker. Docker. URL: https://www.docker.com/
19. Cloud computing services - amazon web services (AWS). Amazon Web Services, Inc. URL: https://aws.amazon.com/?nc1=h_ls
20. Postman. URL: https://www.postman.com/

ДОДАТОК 1

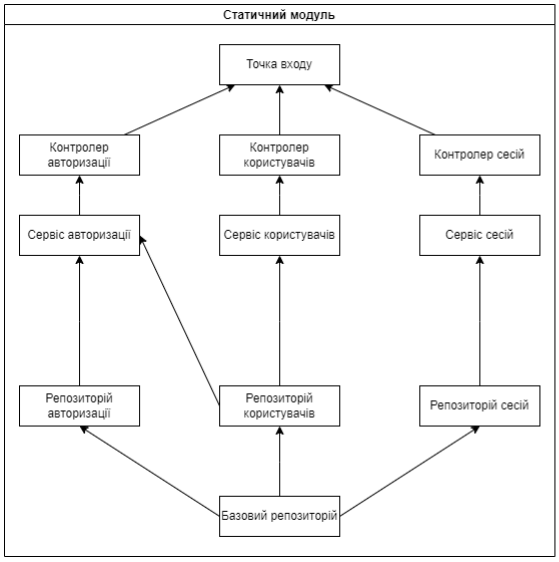
Серверна програмна система мобільної багатокористувацької асиметричної VR-гри

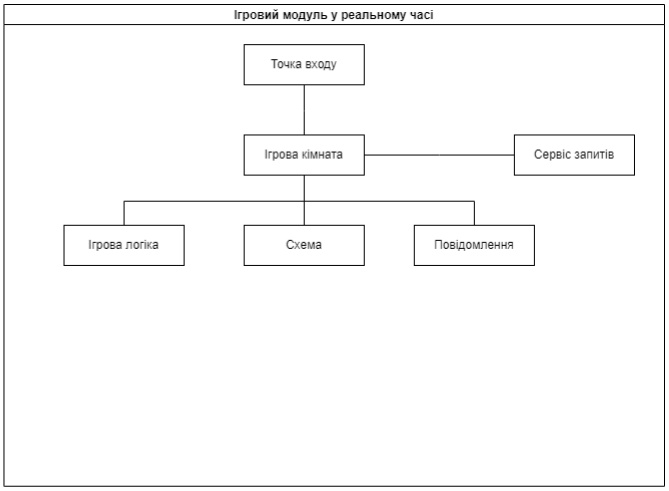
СХЕМА СТРУКТУРНА

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2022 р





					ІАЛЦ.467200.004 Д1		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Бойко А. О.			**Серверна програмна система мобільної багатокористувацької асиметричної VR-гри**	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Новотарський М. А.					
Реценз.					Схема структурна	НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-83	
Н. Контр.		Сімоненко В. П.					
Затвердив							

ДОДАТОК 2

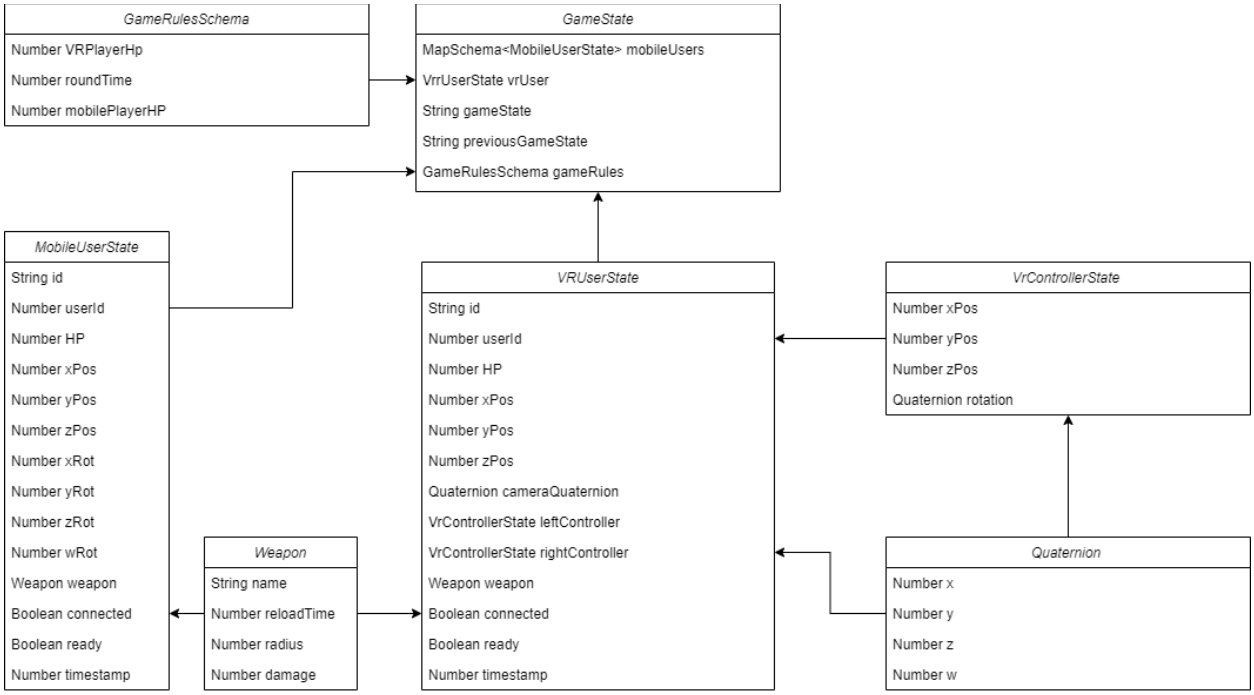
Серверна програмна система мобільної багатокористувацької асиметричної VR-гри

ДІАГРАМА КЛАСІВ

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.005 Д2		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Бойко А. О.			**Серверна програмна система мобільної багатокористувацької асиметричної VR-гри**	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Новотарський М. А.					
Реценз.						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-83	
Н. Контр.		Сімоненко В. П.			Схема функціональна		
Затвердив							

ДОДАТОК З

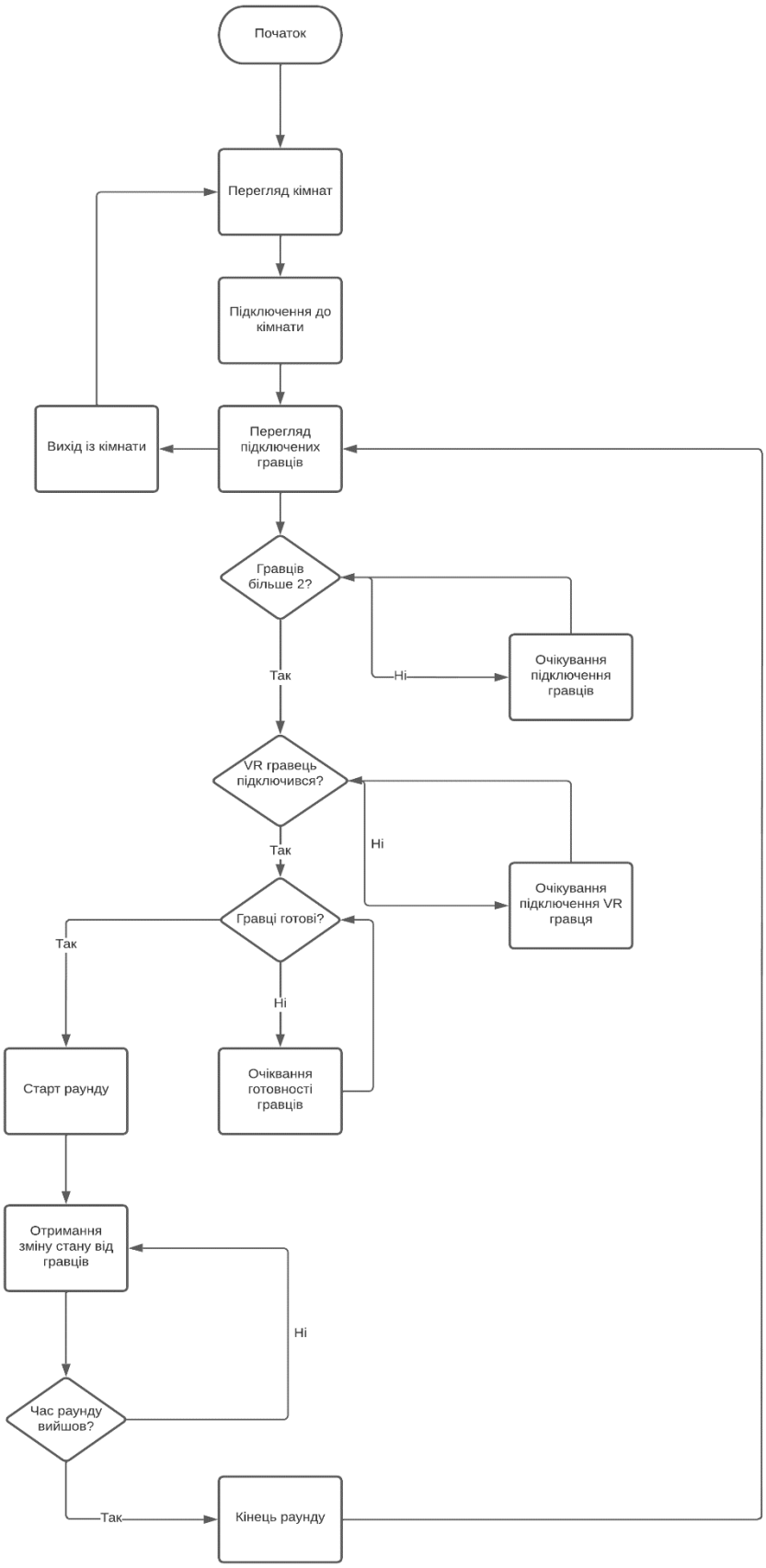
Серверна програмна система мобільної багатокористувацької асиметричної VR-гри

ПРИНЦИПОВА СХЕМА

ІАЛЦ.467200.005 ДЗ

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.005 ДЗ		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Бойко А. О.			**Серверна програмна система мобільної багатокористувацької асиметричної VR-гри**	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Новотарський М. А.					
Реценз.						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-83	
Н. Контр.		Сімоненко В. П.			Принципова схема		
Затвердив							

ДОДАТОК 4

Серверна програмна система мобільної багатокористувацької асиметричної VR-гри

ЛІСТИНГ ПРОГРАМИ

Аркушів 60

Київ 2022 р

index.ts

```
import express from 'express';

import expressify from 'uwebsockets-express';

import { Server } from '@colyseus/core';

import { uWebSocketsTransport } from '@colyseus/uwebsockets-transport';

import { monitor } from '@colyseus/monitor';

import { FMTSRoom } from './rooms/FMTSRoom';


const transport = new uWebSocketsTransport({

    /* ...options */

});

const app = expressify(transport.app);


// use existing middleware implementations!

app.use(express.json());


// register routes

app.get('/hello', (req, res) => {

    res.json({ hello: 'world!' });

});


app.use('/colyseus', monitor());


const port = Number(process.env.port) || 3000;
```

```
const gameServer = new Server({
    transport
});

gameServer.define('fmts', FMTSRoom);

gameServer.listen(port).then(() => console.info(`Listening on ws://localhost:${port}`));
```

FMTSRoom.ts

```
import { Client, Room } from 'colyseus';
import { createSession } from './api/createSession';
import { initGameLoop } from './gameLogic/initGameLogic';
import {
    MobilePlayerMoveMessage,
    PlayerDamage,
    ReadyState,
    VrPlayerMoveMessage,
} from './messages/messages';
import { GameRulesSchema } from './schema/GameRules';
import { GameState } from './schema/GameState';
import { MobileUserState } from './schema/MobileUser';
import { Quaternion } from './schema/Quaternion';
import { VrControllerState, VRUserState } from './schema/VRUser';
import { Weapon } from './schema/Weapon';

export class FMTSRoom extends Room<GameState> {
```

```
private roomOptions: any;
private serverTime  = 0;
public sessionId = 0;

constructor() {
    super();
}

async onCreate(options: any) {
    this.maxClients = 25;
    this.roomOptions = options;

    if (options['roomId'] != null) {
        this.roomId = options['roomId'];
    }

    const gameRules = new GameRulesSchema().assign({
        mobilePlayerHP: 20,
        VRPlayerHP: 100,
        roundTime: 1000,
    });

    this.setState(new GameState().assign({ gameRules }));

    const sessionId = await createSession();
```

```
this.sessionId = sessionId;


this.onMessage('ping',
(client) => {
    client.send(0, { serverTime: this.serverTime });
});


this.onMessage('vrPlayerMove',
(_, message: VrPlayerMoveMessage) => {
    this.state.vrUser.xPos = message.xPos;
    this.state.vrUser.yPos = message.yPos;
    this.state.vrUser.zPos = message.zPos;


    this.state.vrUser.cameraQuaternion = message.cameraQuaternion;
    this.state.vrUser.leftController = message.leftController;
    this.state.vrUser.rightController = message.rightController;
});


this.onMessage('mobilePlayerMove',
    (client, message: MobilePlayerMoveMessage) => {
        const player = this.state.mobileUsers.get(client.id);
        if (player) {
            player.xPos = message.xPos;
            player.yPos = message.yPos;
```

```
            player.zPos = message.zPos;
            player.xRot = message.xRot;
            player.yRot = message.yRot;
            player.zRot = message.zRot;
            player.wRot = message.wRot;
        }
    }
);

this.onMessage('mobilePlayerDamage',
    (client, message: PlayerDamage) => {
        const player = this.state.mobileUsers.get(client.id);
        if (player) {
            player.HP -= message.damage;
            player.timestamp = this.serverTime;
        }
    }
);
this.onMessage('VRPlayerDamage',
(client, message: PlayerDamage) => {
    this.state.vrUser.HP -= message.damage;
    this.state.vrUser.timestamp = this.serverTime;
});

this.onMessage('ready',
```

```
    (client, message: ReadyState) => {
        if (message.type === 'VR') {
            this.state.vrUser.ready = true;
        } else {
            const mobileUser = this.state.mobileUsers.get(client.id);
            if (mobileUser) {
                mobileUser.ready = true;
            }
        }
    });

    this.setSimulationInterval((dt) => {
        this.serverTime += dt;
        //Run Custom Logic for room if loaded
        initGameLoop(this, dt, this.serverTime, this.roomOptions);
    });
}

onJoin(client: Client, options: any) {
    const { type } = options;

    console.info(`${type} Client joined!- ${client.sessionId}`);

    if (type === 'VR') {
        const VRUser = new VRUserState().assign({
```

```
        id: client.id,
        userId: options.userId,
        HP: 100,
        cameraQuaternion: new Quaternion(),
        leftController: new VrControllerState(),
        rightController: new VrControllerState(),
        weapon: new Weapon(
            type,
            Math.random() > 0.5 ? 'SWORD' : 'SPEAR'
        ),
    });

    client.send('onJoin', VRUser);

    this.state.vrUser = VRUser;
    return;
}

const MobileUser = new MobileUserState().assign({
    id: client.id,
    userId: options.userId,
    HP: 20,
    weapon: new Weapon(type, Math.random() > 0.5 ? 'SWORD' : 'SPEAR'),
});
```

```
    this.state.mobileUsers.set(client.id, MobileUser);

    client.send('onJoin', MobileUser);
}

async onLeave(client: Client, consented: boolean) {
    const isVrUserDisconnected = this.state.vrUser.id === client.id;

    if (isVrUserDisconnected) {
        this.broadcast('pause');
        this.state.vrUser.connected = false;

        try {
            if (consented) {
                throw new Error('Consented leave!');
            }

            console.info(
                `Waiting for client ${client.sessionId} reconnection`
            );
            const newClient = await this.allowReconnection(client, 10);
            console.info('Reconnected! client: ' + newClient.id);
            this.state.vrUser.connected = true;
            this.broadcast('resume');
        } catch (e) {
```

```
        console.info('Disconnected! client: ' + client.id);
    }
    return;
}

const mobileUserDisconnected = this.state.mobileUsers.get(client.id);

if (mobileUserDisconnected) {
    mobileUserDisconnected.connected = false;

    try {
        if (consented) {
            throw new Error('Consented leave!');
        }

        console.info(
            `Waiting for client ${client.sessionId} reconnection`
        );
        const newClient = await this.allowReconnection(client, 10);
        console.info('Reconnected! client: ' + newClient.id);
        mobileUserDisconnected.connected = true;
    } catch (e) {
        console.info('Disconnected! client: ' + client.id);
    }
}
```

```
    }
}
```

Types/Players.ts

```
import { UserType } from './types';

export class Player {
    constructor(
        public readonly id: string,
        public readonly type: UserType,
        private _ready: boolean
    ) {}

    public get isReady(): boolean {
        return this._ready;
    }
    public set isReady(isReady: boolean) {
        this._ready = isReady;
    }
}
```

Types/types.ts

```
export type UserType = 'VR' | 'MOBILE';

export type WeaponType = 'SWORD' | 'SPEAR';

export enum GameStateEnum {
```

```
    WAITING = 'WAITING',
    START_ROUND = 'START_ROUND',
    SIMULATE_ROUND = 'SIMULATE_ROUND',
    END_ROUND = 'END_ROUND',
    NONE = 'NONE',
}

export type WonSide = 'VR' | 'MOBILE' | 'DRAW';
```

Schema/GameRules.ts

```
import { Schema, type } from '@colyseus/schema';

export class GameRulesSchema extends Schema {
    @type('int32') VRPlayerHP!: number;
    @type('int32') mobilePlayerHP!: number;
    @type('int32') roundTime!: number;
}
```

Schema/GameState.ts

```
import { Schema, type, MapSchema } from '@colyseus/schema';
import { GameStateEnum } from '../types/types';
import { GameRulesSchema } from './GameRules';
import { MobileUserState } from './MobileUser';
import { VRUserState } from './VRUser';

export class GameState extends Schema {
    @type({ map: MobileUserState }) mobileUsers =
```

```
        new MapSchema<MobileUserState>();

    @type(VRUserState) vrUser!: VRUserState;

    @type('string') gameState = GameStateEnum.WAITING;
    @type('string') previousGameState = GameStateEnum.END_ROUND;
    @type(GameRulesSchema) gameRules!: GameRulesSchema;
}
```

Schema/MobileUser.ts

```
import { Schema, type } from '@colyseus/schema';
import { Weapon } from './Weapon';

export class MobileUserState extends Schema {
    @type('string') id = 'ID';
    @type('number') userId!: number;
    @type('number') HP!: number;

    @type('number') xPos = 0.0;
    @type('number') yPos = 0.0;
    @type('number') zPos = 0.0;

    @type('number') xRot = 0.0;
    @type('number') yRot = 0.0;
    @type('number') zRot = 0.0;
```

```
    @type('number') wRot = 0.0;
    @type(Weapon) weapon!: Weapon;
    @type('boolean') connected = true;
    @type('boolean') ready = false;

    @type('number') timestamp = 0.0;
}
```

Schema/Quaternion.ts

```
import { Schema, type } from '@colyseus/schema';

export class Quaternion extends Schema {
    @type('number') x = 0.0;
    @type('number') y = 0.0;
    @type('number') z = 0.0;
    @type('number') w = 0.0;
}
```

Schema/VRUser.ts

```
import { Schema, type } from '@colyseus/schema';
import { Quaternion } from './Quaternion';
import { Weapon } from './Weapon';

export class VrControllerState extends Schema {
    @type('number') xPos = 0.0;
    @type('number') yPos = 0.0;
    @type('number') zPos = 0.0;
```

```
    @type(Quaternion) rotation = [0, 0, 0, 0];
}

export class VRUserState extends Schema {
    @type('string') id = 'ID';
    @type('number') userId!: number;
    @type('number') HP!: number;

    @type('number') xPos = 0.0;
    @type('number') yPos = 0.0;
    @type('number') zPos = 0.0;
    @type('boolean') connected = true;
    @type('boolean') ready = false;
    @type(Quaternion) cameraQuaternion!: Quaternion;
    @type(VrControllerState) leftController!: VrControllerState;
    @type(VrControllerState) rightController!: VrControllerState;
    @type(Weapon) weapon!: Weapon;

    @type('number') timestamp = 0.0;
}
```

Schema/Weapon.ts

```
import { Schema, type } from '@colyseus/schema';
import { weaponMap } from '../constants/constants';
import { UserType, WeaponType } from '../types/types';
```

```
export class Weapon extends Schema {
    @type('string') name!: string;
    @type('number') reloadTime!: number;
    @type('number') radius!: number;
    @type('number') damage!: number;

    constructor(userType: UserType, name: WeaponType = 'SWORD') {
        super();
        const weapon = weaponMap[userType].find((w) => w.name === name);
        this.name = name;
        if (weapon) {
            this.reloadTime = weapon?.realoadTime;
            this.radius = weapon?.radius;
            this.damage = weapon?.damage;
        }
    }
}
```

Messages/messages.ts

```
import { Quaternion } from '../schema/Quaternion';
import { VrControllerState } from '../schema/VRUser';
import { UserType } from '../types/types';

export interface ReadyState {
    isReady: boolean;
    type: UserType;
```

```
}

export const READY = {
    isReady: true,
};

export const NOT_READY = {
    isReady: false,
};

export interface VrPlayerMoveMessage  {
    xPos: number;
    yPos: number;
    zPos: number;
    cameraQuaternion: Quaternion;
    leftController: VrControllerState;
    rightController: VrControllerState;
}

export interface MobilePlayerMoveMessage  {
    xPos: number;
    yPos: number;
    zPos: number;
    xRot: number;
    yRot: number;
```

```
    zRot: number;
    wRot: number;
}

export interface PlayerDamage {
    damage: number;
}
```

gameLogic/endRoundLogic.ts

```
import { finishSession } from '../api/finishSession';
import { FMTSRoom } from '../FMTSRoom';
import { GameStateEnum } from '../types/types';
import { updateGameState } from './helpers/helpers';

export const endRoundLogic = (roomRef: FMTSRoom) => {
    const isVrLost = roomRef.state.vrUser.HP === 0;

    if (isVrLost) {
        roomRef.broadcast('onRoundEnd', { wonSide: 'MOBILE' });
        finishSession(roomRef.sessionId, 'MOBILE');
        return updateGameState(roomRef, GameStateEnum.WAITING);
    }

    finishSession(roomRef.sessionId, 'VR');
    roomRef.broadcast('onRoundEnd', { wonSide: 'VR' });
    updateGameState(roomRef, GameStateEnum.WAITING);
```

```
};
```

gameLogic/gameLoop.ts

```
import { FMTSRoom } from '../FMTSRoom';
import { GameStateEnum } from '../types/types';
import { endRoundLogic } from './endRoundLogic';
import { getGameState } from './helpers/helpers';
import { simulateRoundLogic } from './simulateRoundLogic';
import { startGameRound } from './startGameRound';
import { waitingGameLogic } from './waitingGameLogic';


export const gameLoop = function (
    roomRef: FMTSRoom,
    deltaTime: number,
    currentTime:  number,
    roomOptions: any
) {
    // Update the game state
    switch (getGameState(roomRef)) {
        case GameStateEnum.NONE:
            break;
        case GameStateEnum.START_ROUND:
            startGameRound(roomRef);
            break;
        case GameStateEnum.WAITING:
            waitingGameLogic(roomRef, roomOptions);
```

```
            break;
        case GameStateEnum.SIMULATE_ROUND:
            simulateRoundLogic(roomRef, currentTime);
            break;
        case GameStateEnum.END_ROUND:
            endRoundLogic(roomRef);
            break;
        default:
            break;
    }
};
```

gameLogic/initGameLogic.ts

```
import { FMTSRoom } from '../FMTSRoom';
import { gameLoop } from './gameLoop';

export const initGameLoop = (
    roomRef: FMTSRoom,
    deltaTime: number,
    currentTime: number,
    roomOptions: any
) => {
    gameLoop(roomRef, deltaTime / 1000, currentTime, roomOptions);
};
```

gameLogic/simulateRoundLogic.ts

```
import { FMTSRoom } from '../FMTSRoom';
import { GameStateEnum } from '../types/types';
import { updateGameState } from './helpers/helpers';

export const simulateRoundLogic = (roomRef: FMTSRoom, currentTime: number) => {
    if (currentTime >= roomRef.state.gameRules.roundTime) {
        updateGameState(roomRef, GameStateEnum.END_ROUND);
    }

    if (roomRef.state.vrUser.HP <= 0) {
        updateGameState(roomRef, GameStateEnum.END_ROUND);
    }

    let allMobilePlayersAreDead = 0;
    roomRef.state.mobileUsers.forEach((player) => {
        if (player.HP <= 0) {
            allMobilePlayersAreDead += 1;
        }
    });

    if (allMobilePlayersAreDead === roomRef.state.mobileUsers.size) {
        updateGameState(roomRef, GameStateEnum.END_ROUND);
    }
};
```

gameLogic/startGameRound.ts

```
import { addUsersToSession } from '../api/addUsersToSession';
import { FMTSRoom } from '../FMTSRoom';
import { GameStateEnum } from '../types/types';
import { updateGameState } from './helpers/helpers';

export const startGameRound = (roomRef: FMTSRoom) => {
    const userIds = [
        roomRef.state.vrUser.userId,
        ...Array.from(roomRef.state.mobileUsers.values()).map((p)  => p.userId),
    ];
    addUsersToSession(userIds, roomRef.sessionId);
    roomRef.broadcast('beginRound', {});
    roomRef.lock();
    updateGameState(roomRef, GameStateEnum.SIMULATE_ROUND);
};
```

gameLogic/waitingGameLogic.ts

```
import { FMTSRoom } from '../FMTSRoom';
import { GameStateEnum } from '../types/types';
import { updateGameState } from './helpers/helpers';

export const waitingGameLogic = (roomRef: FMTSRoom, roomOptions: any) => {
    if (!roomRef.state.vrUser) {
        return;
    }
```

```
    const currentUsers =
        roomRef.state.mobileUsers.size + (!!roomRef.state.vrUser ? 1 : 0);

    const currentUsersReadyState = [
        roomRef.state.vrUser.ready,
        Array.from(roomRef.state.mobileUsers.values()).map((p)  => p.ready),
    ];
    const minReqPlayersToStartRound = Number(roomOptions['minReqPlayers']  || 2);
    if (currentUsers < minReqPlayersToStartRound) {
        return;
    }

    const playersReady = currentUsersReadyState.every((r) => r);

    if (playersReady === false) {
        return;
    }

    updateGameState(roomRef, GameStateEnum.SIMULATE_ROUND);
};
```

gameLogic/helpers/helpers.ts

```
import { FMTSRoom } from '../../FMTSRoom';
import { GameStateEnum } from '../../types/types';

export const getGameState = (roomRef: FMTSRoom) => roomRef.state.gameState;
```

```
export const getPreviousGameState = (roomRef: FMTSRoom) =>
    roomRef.state.previousGameState;


export const updateGameState = (roomRef: FMTSRoom, newState: GameStateEnum) => {
    roomRef.state.previousGameState = roomRef.state.gameState;
    roomRef.state.gameState = newState;
};


export const unlockIfAble = (roomRef: FMTSRoom) => {
    if (roomRef.hasReachedMaxClients() === false) {
        roomRef.unlock();
    }
};
```

constants/constants.ts

```
export const weaponMap = {
    VR: [
        { name: 'SWORD', realoadTime: 300, radius: 120, damage: 10 },
        { name: 'SPEAR', realoadTime: 400, radius: 220, damage: 10 },
    ],
    MOBILE: [
        { name: 'SWORD', realoadTime: 100, radius: 30, damage: 2 },
        { name: 'SPEAR', realoadTime: 150, radius: 45, damage: 2 },
    ],
};
```

api/addUsersToSession.ts

```
import axios from 'axios';


export const addUsersToSession = async (
    userIds: number[],
    sessionId: number
) => {
    try {
        await axios({
            method: 'POST',
            url: 'http://localhost:8080/api/v1/sessions/users/add',
            data: {
                userIds,
                sessionId,
            },
        });
    } catch (e) {
        console.error(e);
    }
};
```

api/createSession.ts

```
import axios from 'axios';


export const createSession = async () => {
    try {
```

```
        const { data } = await axios({
            method: 'POST',
            url: 'http://localhost:8080/api/v1/sessions/start',
        });

        return data.sessionId;
    } catch (e) {
        console.error(e);
    }
};
```

api/finishSession.ts

```
import axios from 'axios';
import { WonSide } from '../types/types';

export const finishSession = async (sessionId: number, wonSide: WonSide) => {
    await axios({
        method: 'POST',
        url: 'http://localhost:8080/api/v1/sessions/finish',
        data: {
            sessionId,
            wonSide,
        },
    });
};
```

config/config.ts

```
import fs from 'fs';
import path from 'path';

type Config = {
    TOKEN_SECRET: string;
    API_URL: string;
    FACEBOOK_OAUTH_PUBLIC: string;
    FACEBOOK_OATH_SECRET: string;
    HOST: string;
    KNEX: {
        DB_HOST: string;
        DB_NAME: string;
        DB_USER: string;
        DB_PASSWORD: string;
        DB_PORT: string;
        DB_POOL_MIN: string;
        DB_POOL_MAX: string;
        DB_POOL_IDLE: string;
    };
};

const getConfig = () => {
    const nodeEnv = process.env.NODE_ENV;

    const filename = `config.${
```

```
        nodeEnv === 'dev' ? 'development' : 'production'
    }.json`;

    const configFile = fs.readFileSync(path.join(__dirname, filename));

    const config = JSON.parse(configFile.toString()) as Config;

    return config;
};

export const config = getConfig();
```

config/knex.ts

```
import { config } from './config';

export const getKnexConfig = () => {
    return {
        client: 'postgresql',
        connection: {
            host: config.KNEX.DB_HOST,
            database: config.KNEX.DB_NAME,
            user: config.KNEX.DB_USER,
            password: config.KNEX.DB_PASSWORD,
            port: Number(config.KNEX.DB_PORT),
        },
        pool: {
```

```
        min: Number(config.KNEX.DB_POOL_MIN),
        max: Number(config.KNEX.DB_POOL_MAX),
        idleTimeoutMillis:  Number(config.KNEX.DB_POOL_IDLE),
      },
      migrations: {
        tableName: 'KnexMigrations',
      },
    };
};

export const Knex = {
  config: getKnexConfig(),
};
```

db/knex.ts

```
import Knex from 'knex';
import { config } from '../config/config';

export function initKnex() {
  try {
    const knex = Knex({
      client: 'pg',
      connection: {
        host: config.KNEX.DB_HOST,
        database: config.KNEX.DB_NAME,
        user: config.KNEX.DB_USER,
```

```
            password: config.KNEX.DB_PASSWORD,
            port: Number(config.KNEX.DB_PORT),
        },
        pool: {
            min: Number(config.KNEX.DB_POOL_MIN),
            max: Number(config.KNEX.DB_POOL_MAX),
            idleTimeoutMillis:  Number(config.KNEX.DB_POOL_IDLE),
        },
        acquireConnectionTimeout:  2000,
    });

    return knex;
  } catch (error) {
    throw new Error(
      'Unable to connect to Postgres via Knex. Ensure a valid connection.'
    );
  }
}

export const knex = initKnex();
```

middleware/validateToken.ts

```
import axios from 'axios';
import jwt from 'jsonwebtoken';
import jwktopem from 'jwk-to-pem';
```

```
export const validateToken = async (req: any, res: any, next: any) => {
    try {
        const bearerHeader = req.headers['authorization'];

        if (bearerHeader) {
            const bearer = bearerHeader.split(' ');
            const token = bearer[1];

            const result = await axios({
                url: 'http://localhost:8080/api/v1/jwks/public',
            });

            const jwksResponse = result.data;

            const publicKey = jwktopem(jwksResponse.keys[0]);
            try {
                const decoded = jwt.verify(token, publicKey);

                if (typeof decoded !== 'string') {
                    res.locals = {
                        id: decoded.id,
                        email: decoded.email,
                        firstName: decoded.firstName,
                        lastName: decoded.lastName,
                    };
```

```
                return next();
            }

            return res.status(401).send('Wrong token provided');
        } catch (e) {
            return res.status(401).send('Wrong token provided');
        }
    }

    return res.status(401).send('No token provided');
  } catch (e) {
    return res.status(500).send(e);
  }
};
```

middleware/validateUserAccess.ts

```
export const validateUserAccess = async (req: any, res: any, next: any) => {
  const { id } = res.locals;

  if (id !== Number(req.params.id)) {
    return res.status(400).send("You cannot modify other user's data!");
  }

  return next();
};
```

models/auth.model.ts

```
export interface IAuth {
    id: number;
    accessToken: string;
    refreshToken: string;
    userId: number;
}
```

models/sessions.model.ts

```
export type WonSide = 'VR' | 'MOBILE' | 'DRAW';

export type SesionStatus = 'IN_PROGRES' | 'COMPLETED' | 'ABORTED';

export interface ISession {
    id: number;
    startTime: Date;
    endTime: Date;
    wonSide: WonSide | null;
    status: SesionStatus;
}

export interface ISessionPlayers {
    id: number;
    userId: number;
    sessionId: number;
}
```

models/users.model.ts

```
export interface IUser {
    id: number;
    firstName: string;
    lastName: string;
    email: string;
}
```

routes/index.ts

```
import { Express } from 'express';
import { Knex } from 'knex';
import { createAuthRoutes } from './auth.routes';
import { createSessionsRoutes } from './sessions.router';
import { createUserRoutes } from './user.routes';

export const createRoutes = (app: Express, db: Knex) => {
    createAuthRoutes(app, db);
    createUserRoutes(app, db);
    createSessionsRoutes(app, db);
};
```

routes/auth.routes.ts

```
import { Express } from 'express';
import { Knex } from 'knex';
import {
    AuthController,
    AuthJWKSController,
} from '../services/auth/auth.controller';
```

```
export const createAuthRoutes = (app: Express, db: Knex) => {
    const authController = new AuthController(db);
    app.get('/api/v1/auth/social', authController.find.bind(authController));

    const authJWKSController = new AuthJWKSController(db);
    app.get(
        '/api/v1/jwks/public',
        authJWKSController.find.bind(authController)
    );
};
```

routes/sessions.routes.ts

```
import { Express } from 'express';
import { Knex } from 'knex';
import { validateToken } from '../middleware/validateToken';
import {
    SessionsController,
    SessionsStateController,
} from '../services/sessions/sessions.controller';

export const createSessionsRoutes = (app: Express, db: Knex) => {
    const sessionsController = new SessionsController(db);
    app.get(
        '/api/v1/sessions',
        validateToken,
```

```
    sessionsController.find.bind(sessionsController)
);
app.get(
    '/api/v1/sessions/:id',
    validateToken,
    sessionsController.get.bind(sessionsController)
);

const sessionsStateController = new SessionsStateController(db);
app.post(
    '/api/v1/sessions/start',
    validateToken,
    sessionsStateController.create.bind(sessionsStateController)
);
app.post(
    '/api/v1/sessions/users/add',
    validateToken,
    sessionsStateController.addUsers.bind(sessionsStateController)
);
app.post(
    '/api/v1/sessions/finish',
    validateToken,
    sessionsStateController.finish.bind(sessionsStateController)
);
app.post(
```

```
        '/api/v1/sessions/abort',
        validateToken,
        sessionsStateController.abortSession.bind(sessionsStateController)
    );
};
```

routes/users.routes/ts

```
import { Express } from 'express';
import { Knex } from 'knex';
import { validateToken } from '../middleware/validateToken';
import { validateUserAccess } from '../middleware/validateUserAccess';
import { UserController } from '../services/users/users.controller';

export const createUserRoutes = (app: Express, db: Knex) => {
    const userController = new UserController(db);
    app.get(
        '/api/v1/users',
        validateToken,
        userController.find.bind(userController)
    );
    app.get(
        '/api/v1/users/:id',
        validateToken,
        validateUserAccess,
        userController.get.bind(userController)
    );
```

```
    app.patch(
        '/api/v1/users/:id',
        validateToken,
        validateUserAccess,
        userController.patch.bind(userController)
    );
    app.delete(
        '/api/v1/users/:id',
        validateToken,
        validateUserAccess,
        userController.remove.bind(userController)
    );
};
```

server/composeServer.ts

```
import { Express } from 'express';
import { initKnex } from '../db/knex';
import { createRoutes } from '../routes';

export const composeServer = (app: Express): Express => {
    const db = initKnex();

    createRoutes(app, db);

    return app;
};
```

server/configureRequest.ts

```
import cors from 'cors';
import { json, urlencoded } from 'body-parser';
import express from 'express';

const allowedOrigins = [
    'http://localhost:3000',
    'http://192.168.0.109:3000',
    'http://192.168.0.107:3000',
];

const options: cors.CorsOptions = {
    origin: allowedOrigins,
};

export const configureRequest = () => {
    const app = express();

    app.use(cors(options));

    app.use(json());
    app.use(
        urlencoded({
            extended: true,
        })
```

```
  );

  return app;
};
```

services/auth.controller.ts

```
import { Response } from 'express';
import { Knex } from 'knex';
import { UserRepo } from '../users/users.repo';
import { AuthRepo } from './auth.repo';
import { AuthService } from './auth.service';
class AuthControllerWithServices {
  authService: AuthService;

  constructor(db: Knex) {
    this.authService = new AuthService({
      authRepo: new AuthRepo(db),
      userRepo: new UserRepo(db),
    });
  }
}

export class AuthController extends AuthControllerWithServices {
  async find(req: any, res: Response) {
    try {
      const { code } = req.query;
```

```
            const info = await this.authService.getFacebookInfo(code);

            const user = await this.authService.createUser(info);

            const token = await this.authService.createToken(user);

            await this.authService.saveUserToken(token.toString(), user.id);

            return res.status(200).json(token);
        } catch (e) {
            return res.status(500).send(e);
        }
    }
}

export class AuthJWKSController extends AuthControllerWithServices {
    async find(req: any, res: Response) {
        try {
            const tokens = await this.authService.getPublicJWKSTokens();
            return res.status(200).json(tokens);
        } catch (e) {
            return res.status(500).send(e);
        }
    }
```

```
}
```

services/auth.service.ts

```
import axios from 'axios';
import fs from 'fs';
import jose from 'node-jose';
import path from 'path';
import { config } from '../../config/config';
import { IUser } from '../../models/users/user.model';
import { UserRepo } from '../users/users.repo';
import { AuthRepo } from './auth.repo';

type ServiceProps = {
    authRepo: AuthRepo;
    userRepo: UserRepo;
};

const defaultUser = {
    id: 1,
    firstName: 'Andrii',
    lastName: 'Boiko',
    email: 'ab@gmail.com',
};

export class AuthService {
    private authRepo: AuthRepo;
```

```
private userRepo: UserRepo;

constructor({ authRepo, userRepo }: ServiceProps) {
    this.authRepo = authRepo;
    this.userRepo = userRepo;
}

async getFacebookInfo(code: string) {
    const { data: tokenData } = await axios({
        url: 'https://graph.facebook.com/v4.0/oauth/access_token',
        method: 'get',
        params: {
            client_id: config.FACEBOOK_OAUTH_PUBLIC,
            client_secret: config.FACEBOOK_OATH_SECRET,
            redirect_uri: `${config.HOST}/api/v1/auth/social`,
            code,
        },
    });

    const { data: userInfo } = await axios({
        url: 'https://graph.facebook.com/me',
        method: 'get',
        params: {
            fields: ['email', 'first_name', 'last_name'].join(','),
            access_token: tokenData.access_token,
```

```
        },
    });

    const { email, first_name, last_name } = userInfo;

    return { email, firstName: first_name, lastName: last_name };
}

async createUser(user: Omit<IUser, 'id'>) {
    const [createdUser] = await this.userRepo.saveItem(user, ['*']);

    return createdUser as IUser;
}

async getPublicJWKSTokens() {
    const ks = fs.readFileSync(path.join('src', 'config', 'keys.json'));

    const keyStore = await jose.JWK.asKeyStore(ks.toString());

    return keyStore.toJSON();
}

async createToken(user?: IUser) {
    const JWKeys = fs.readFileSync(path.join('src', 'config', 'keys.json'));
```

```
    const keyStore = await jose.JWK.asKeyStore(JWKeys.toString());

    const [key] = keyStore.all({ use: 'sig' });

    const opt = { compact: true, jwk: key, fields: { typ: 'jwt' } };

    const payload = JSON.stringify({
      iat: Math.floor(Date.now() / 1000),
      sub: 'test',
      ...(user || defaultUser),
    });

    const token = await jose.JWS.createSign(opt, key)
      .update(payload)
      .final();

    return token;
  }

  async saveUserToken(token: string, userId: number) {
    return this.authRepo.saveItem({
      accessToken: token,
      refreshToken: token,
      userId: userId,
    });
```

```
    }
}
```

services/auth.repo.ts

```
import { BaseRepo } from '../../utils/BaseRepo';
import { IAuth } from '../../models/auth/auth.model';
import { Knex } from 'knex';

export class AuthRepo extends BaseRepo<IAuth> {
    constructor(db: Knex) {
        super(db, 'auth');
    }

    async deleteUserTokens(userId: number) {
        return this.table.delete().where({ userId });
    }
}
```

services/users.controller/ts

```
import { Response } from 'express';
import { Knex } from 'knex';
import { AuthRepo } from '../auth/auth.repo';
import { SessionsRepo } from '../sessions/sessions.repo';
import { UserRepo } from './users.repo';
import { UserService } from './users.service';

class UserControllerWithServices {
```

```
    userService: UserService;

    constructor(db: Knex) {
        this.userService = new UserService({
            userRepo: new UserRepo(db),
            sessionsRepo: new SessionsRepo(db),
            authRepo: new AuthRepo(db),
        });
    }
}

export class UserController extends UserControllerWithServices {
    async find(_: any, res: Response) {
        const users = await this.userService.getAllUsers();
        return res.status(200).json(users);
    }

    async get(req: any, res: Response) {
        const { id } = req.params;

        if (!id || Number.isNaN(Number(id))) {
            return res.status(400).send('Wrong id');
        }

        const user = (await this.userService.getUserById(id)) || {};
```

```
        return res.status(200).json(user);
    }

    async patch(req: any, res: any) {
        const { id } = req.params;

        const updatedUser = await this.userService.patchUserById(id, req.body);

        return res.status(200).json(updatedUser);
    }

    async remove(req:  any, res: any) {
        const { id } = req.params;

        await this.userService.deleteUserById(id);

        return res.status(200).json('OK');
    }
}
```

services/users.service/ts

```
import { IUser } from '../../models/users/user.model';
import { AuthRepo } from '../auth/auth.repo';
```

```
import { SessionsRepo } from '../sessions/sessions.repo';
import { UserRepo } from './users.repo';

type ServiceProps = {
    userRepo: UserRepo;
    sessionsRepo: SessionsRepo;
    authRepo: AuthRepo;
};

export class UserService {
    private userRepo: UserRepo;
    private sessionsRepo: SessionsRepo;
    private authRepo: AuthRepo;

    constructor({ userRepo, sessionsRepo, authRepo }: ServiceProps) {
        this.userRepo = userRepo;
        this.sessionsRepo = sessionsRepo;
        this.authRepo = authRepo;
    }

    async getAllUsers() {
        return this.userRepo.getAllItems();
    }

    async getUserById(id: number) {
```

```
        return this.userRepo.getItemById(id);
    }

    async patchUserById(id: number, user: Partial<Omit<IUser, 'id'>>) {
        return this.userRepo.updateByFields(user, { id }, Object.keys(user));
    }

    async deleteUserById(id: number) {
        await this.sessionsRepo.deleteUserSessions(id);
        await this.authRepo.deleteUserTokens(id);
        await this.userRepo.deleteUserAccount(id);
    }
}
```

services/users.repo/ts

```
import { BaseRepo } from '../../utils/BaseRepo';
import { Knex } from 'knex';
import { IUser } from '../../models/users/user.model';

export class UserRepo extends BaseRepo<IUser> {
    constructor(db: Knex) {
        super(db, 'users');
    }

    async deleteUserAccount(id: number) {
        return this.table.delete().where({ id });
```

```
    }
}
```

services/sessions.controller.ts

```
import { Response } from 'express';
import { Knex } from 'knex';
import { SessionsRepo } from './sessions.repo';
import { SessionsService  } from './sessions.service';

class SessionsControllerWithServices {
    sessionsService: SessionsService;

    constructor(db: Knex) {
        this.sessionsService = new SessionsService({
            sessionsRepo: new SessionsRepo(db),
        });
    }
}

export class SessionsController extends SessionsControllerWithServices {
    async find(_: any, res: Response) {
        const sessions = await this.sessionsService.getAllSessions();
        return res.status(200).json(sessions);
    }

    async get(req: any, res: any) {
```

```
        const { id } = req.params;

        if (!id || Number.isNaN(Number(id))) {
            return res.status(400).send('Wrong session id');
        }

        const sessions = await this.sessionsService.getSessionsById(id);

        return res.status(200).json(sessions);
    }
}

export class SessionsStateController extends SessionsControllerWithServices {
    async create(req: any, res: any) {
        const sessionId = await this.sessionsService.createSession();

        return res.status(200).json({ sessionId });
    }

    async addUsers(req: any, res: any) {
        const { userIds, sessionId } = req.body;
        await this.sessionsService.addUsersToSession(userIds, sessionId);

        return res.status(200).json('OK');
    }
```

```
    async finish(req: any, res: any) {
        const { sessionId, wonSide } = req.body;

        await this.sessionsService.finishSession(sessionId,  wonSide);

        return res.status(200).json('OK');
    }

    async abortSession(req: any, res: any) {
        const { sessionId } = req.body;

        await this.sessionsService.abortSession(sessionId);

        return res.status(200).json('OK');
    }
}
```

services/sessions.service.ts

```
import { WonSide } from '../../models/sessions/session.model';
import { SessionsRepo } from './sessions.repo';

type ServiceProps = {
    sessionsRepo: SessionsRepo;
};
```

```
export class SessionsService {
    private sessionsRepo: SessionsRepo;

    constructor({ sessionsRepo }: ServiceProps) {
        this.sessionsRepo = sessionsRepo;
    }

    async getAllSessions() {
        return this.sessionsRepo.getAllItems();
    }

    async getSessionsById(id: number) {
        return this.sessionsRepo.getSessionWithPlayersById({ sessionId: id });
    }

    async createSession() {
        const [session] = await this.sessionsRepo.createSession();

        return session.id;
    }

    async addUsersToSession(userIds: number[], sessionId: number) {
        return this.sessionsRepo.attachPlayersToSession(userIds, sessionId);
    }
```

```
    async finishSession(sessionId: number, wonSide: WonSide) {
        return this.sessionsRepo.finishSession(sessionId, wonSide);
    }

    async abortSession(sessionId: number) {
        return this.sessionsRepo.abortSession(sessionId);
    }
}
```

services/sessions.repo.ts

```
import { Knex } from 'knex';
import * as R from 'ramda';
import { ISession, WonSide } from '../../models/sessions/session.model';
import { BaseRepo } from '../../utils/BaseRepo';

type GetSessionWithPlayersByIdParams = {
    sessionId?: number;
    userId?: number;
};

const hydrateSessions = (rawSessions: any[]) => {
    const sessions = {} as any;
    rawSessions.forEach((row) => {
        const userObject = {
            id: row.userId,
            firstName: row.firstName,
```

```
            lastName: row.lastName,
            email: row.email,
        };

        if (!(row.sessionId in sessions)) {
            const sessionObject = {
                id: row.sessionId,
                startTime: row.startTime,
                status: row.status,
                endTime: row.endTime,
                wonSide: row.wonSide,
                users: [userObject],
            };

            return (sessions[row.sessionId] = sessionObject);
        }
        return sessions[row.sessionId].users.push(userObject);
    });

    return R.values(sessions);
};

export class SessionsRepo extends BaseRepo<ISession> {
    constructor(db: Knex) {
        super(db, 'sessions');
```

```
}

async deleteUserSessions(userId: number) {
    return this.table.delete().where({ userId });
}

async getSessionWithPlayersById({
    sessionId,
    userId,
}: GetSessionWithPlayersByIdParams) {
    const rawSessions = await this.table
        .select('*')
        .where(R.pickBy(Boolean, { sessionId, userId }))
        .join(
            'sessions_players',
            'sessions.id',
            '=',
            'sessions_players.sessionId'
        )
        .join('users', 'sessions_players.userId', '=', 'users.id');

    return hydrateSessions(rawSessions);
}

async createSession() {
```

```
    return this.table.insert(
        {
            startTime: new Date(),
        },
        ['id']
    );
}

async attachPlayersToSession(userIds: number[], sessionId: number) {
    return this.getAnotherTable('sessions_players').insert(
        userIds.map((userId) => ({ userId, sessionId }))
    );
}

async finishSession(sessionId: number, wonSide: WonSide) {
    return this.updateByFields(
        {
            wonSide,
            endTime: new Date(),
            status: 'COMPLETED',
        },
        { id: sessionId }
    );
}
```

```
    async abortSession(sessionId: number) {
        return this.updateByFields(
            {
                endTime: new Date(),
                status: 'ABORTED',
                wonSide: null,
            },
            { id: sessionId }
        );
    }
}
```

utils/BaseRepo.ts

```
import { Knex } from 'knex';

export type TableNames = 'users' | 'sessions' | 'auth' | 'sessions_players';

export class BaseRepo<T> {
    readonly db;
    readonly tableName;

    constructor(db: Knex, tableName: TableNames) {
        this.db = db;
        this.tableName = tableName;
    }
```

```
get table() {
    return this.db(this.tableName);
}

async getAllItems() {
    return this.db(this.tableName).select('*');
}

async getItemById(id: number, returning: string[] = ['*']) {
    const [resultItem] = await this.db(this.tableName)
        .select(...returning)
        .where({ id });
    return resultItem;
}

async saveItem(item: Partial<Omit<T, 'id'>>, returning: string[] = []) {
    return this.db(this.tableName).insert(item, returning);
}

async updateByFields(
    item: Partial<Omit<T, 'id'>>,
    queryFields: Partial<T>,
    returning: string[] = []
) {
    return this.table.where(queryFields).update(item, returning);
```

```
    }

    async hasItemByFields(fields: Partial<T>): Promise<boolean> {
        const [item] = await this.db(this.tableName)
            .select('id')
            .where({ ...fields });

        if (item) {
            return true;
        }

        return false;
    }

    getAnotherTable(tableName: TableNames) {
        return this.db(tableName);
    }
}
```

utils/generateJWKS.ts

```
import fs from 'fs';

import jose from 'node-jose';
import path from 'path';
```

```
const keyStore = jose.JWK.createKeyStore();


keyStore.generate('RSA', 2048, { alg: 'RS256', use: 'sig' }).then(() => {
    fs.writeFileSync(
        path.join('src', 'config', 'keys.json'),
        JSON.stringify(keyStore.toJSON(true), null, '  ')
    );
});
```