МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«До захисту допущено»
В.о. завідувача кафедри
Оксана ШОВКОПЛЯС
_____
(підпис)
грудня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 «Комп'ютерні науки»
освітньо-професійної програми «Інформатика»
на тему: Інформаційна технологія керування та аналізу кіберспортивних турнірів з дисципліни CS2
здобувача групи ІН.м-33 Кисленка Ярослава Володимировича

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ Ярослав КИСЛЕНКО
(підпис)

Керівник Оксана ШОВКОПЛЯС
доцент кафедри КН, к.ф.-м.н.,
доцент
_____
(підпис)

Суми – 2024

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_____ Оксана ШОВКОПЛЯС
(підпис)

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістра

зі спеціальності 122 «Комп'ютерні науки», освітньо-професійної програми «Інформатика»
здобувача групи ІН.м-33 Кисленка Ярослава Володимировича

1. Тема роботи: «Інформаційна технологія керування та аналізу кіберспортивних турнірів з дисципліни CS2»
затверджую наказом по СумДУ від *«3» грудня 2024 року № 1257-VI*
2. Термін здачі здобувачем кваліфікаційної роботи *до 06 грудня 2024 року*
3. Вхідні дані до кваліфікаційної роботи _____
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити) *1) Аналіз проблеми предметної області, постановка й формування завдань дослідження. 2) Огляд існуючих рішень та технологій 3) Проєктування системи 4) Розробка інформаційної технології керування та аналізу кіберспортивних турнірів з CS2 5) Аналіз результатів.*
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)_____
6. Консультанти до проекту (роботи), із зазначенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» _____ 20 ___ р.

Завдання прийняв до виконання _____ Керівник _____
(підпис) (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	*Аналіз предметної області та постановка задачі*	11.09.24-6.10.24	
2	*Проєктування системи*	07.10.24-25.10.24	
3	*Розробка серверної частини*	26.10.24-12.11.24	
4	*Розробка клієнтської частини*	13.11.24-20.11.24	
5	*Аналіз отриманих результатів*	21.11.24-24.11.24	
6	*Оформлення пояснювальної записки до кваліфікаційної роботи*	25.11.24-04.12.24	

Здобувач вищої освіти _____ Керівник _____
(підпис) (підпис)

АНОТАЦІЯ

Записка: 99 стор., 44 рис., 2 додатки, 23 використаних джерел.

Обґрунтування актуальності теми роботи – Сучасна кіберспортивна індустрія, зокрема в дисципліні Counter-Strike 2, стрімко розвивається, вимагаючи від організаторів турнірів комплексного підходу до управління та аналізу. Ефективна організація турнірів потребує автоматизації багатьох процесів від автоматизації процесів організації турнірів до детального аналізу ігрових даних. Відсутність універсального та зручного інструменту для цих цілей створює потребу в розробці інформаційної технології, яка б задовольняла сучасні вимоги кіберспортивної індустрії.

Об’єкт дослідження — процеси організації, управління та аналізу кіберспортивних турнірів з дисципліни CS2.

Мета роботи — розробка інформаційної технології для автоматизації процесів управління кіберспортивними турнірами та проведення аналізу ігрових даних у дисципліні Counter-Strike 2.

Методи дослідження — аналіз літератури та існуючих рішень, системне та об'єктно-орієнтоване проєктування, вибір і застосування сучасних технологій, програмна реалізація та аналіз результатів.

Результати — розроблено інформаційну технологію для автоматизації процесів управління кіберспортивними турнірами та аналізу ігрових даних у дисципліні CS2, яка забезпечує можливості створення та управління турнірами, командами та матчами, автоматизацію проведення матчів через інтеграцію з ігровими серверами, збір та обробку ігрових даних, а також зручний вебінтерфейс для адміністраторів і користувачів, що підвищує ефективність організації турнірів та покращує досвід гравців і глядачів.

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ, КІБЕРСПОРТ, АВТОМАТИЗАЦІЯ, АНАЛІЗ ІГРОВИХ ДАНИХ, COUNTER-STRIKE 2

ЗМІСТ

ВСТУП

Обґрунтування вибору теми роботи. Стрімкий розвиток кіберспортивної індустрії, зокрема в дисципліні Counter-Strike 2, потребує сучасних інформаційних рішень для автоматизації організації турнірів та глибокого аналізу ігрових даних, що визначає актуальність цієї теми дослідження.

Актуальність. Відсутність універсальних інструментів для ефективного керування кіберспортивними подіями та аналізу ігрових показників обумовлює необхідність розробки інформаційної технології, здатної оптимізувати процеси управління турнірами й підвищити якість взаємодії між учасниками та глядачами.

Об’єкт дослідження. Процеси організації, управління та аналізу кіберспортивних турнірів з дисципліни CS2..

Предмет дослідження. Програмні засоби автоматизації управління кіберспортивними турнірами та аналізу ігрових даних в CS2.

Гіпотеза. Впровадження інформаційної технології, що поєднує автоматизоване управління турнірами та аналіз ігрових даних, підвищить ефективність організації змагань з CS2, покращить якість взаємодії між учасниками та глядачами, а також надасть розширені аналітичні можливості для покращення ігрових стратегій.

Наукова новизна. Запропоновано підхід до інтеграції автоматизованого управління кіберспортивними турнірами з аналітикою ігрових даних у єдину інформаційну технологію, що дозволяє поглибити розуміння динаміки ігор, покращити процес ухвалення рішень організаторами та сприяти розвитку стратегії команд.

Апробація матеріалів роботи. Основні результати роботи оприлюднені та обговорені на міжнародній науково-технічній конференції студентів та молодих вчених «Інформатика, математика, автоматика» (ІМА – 2024).

Структура. Дана робота робота складається зі вступу, інформаційного огляду, вибору методу розв’язання задачі, моделювання, розробки інфораційної технології, висновків, списку використаних джерел та додатків.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Сутність та особливості кіберспорту

Існує велика кількість визначень кіберспорту, оскільки його роглядають і визначають не лише академічні дослідники, але й аналітики ринку та асоціації кіберспорту. Загалом, кіберспорт часто асоціюється з конкурентним відеоігровим змаганням. Змагання організовуються у вигляді ліг та турнірів, що сприяють як індивідуальним, так і командним змаганням. Для підготовки до цих змагань гравці беруть участь у високоструктурованих тренувальних заходах, таких як буткемпи або регулярні тренувальні матчі – так звані "scrims" [1]. Цифрові технології сприяють як змаганням, так і тренувальній діяльності, забезпечуючи гравцям доступ до комп’ютерів, консолей або інших електронних систем, таких як цифрові платформи та пристрої доповненої або віртуальної реальності.

Історія кіберспорту починається ще в 1980-х роках з виникнення перших формальних змагань у відеоіграх. У цей період популярність здобули аркадні ігри, такі як Pac-Man (1980), Asteroids (1979) та Centipede (1981), які стали платформою для змагань між гравцями. Гравці змагалися за найвищі бали, що стимулювало розвиток конкурентної атмосфери в аркадних закладах. Одним із перших значних кроків у формалізації кіберспорту стало створення U.S. National Video Game Team у 1980-х роках, яка об'єднувала найкращих гравців для участі в різноманітних змаганнях і виставках [2].

У пізні 1990-х роках розвиток інтернету суттєво вплинув на еволюцію кіберспорту. Випуск ігор, таких як Street Fighter II (1991) від Capcom, сприяв популяризації змагань між гравцями, організованих у вигляді ліг та турнірів. Street Fighter II був першою грою, яка зосередила увагу на елементах геймдизайну, спрямованих на конкурентну гру, що значно підвищило рівень майстерності гравців та складність змагань [2].

На сьогоднішній день кіберспорт є глобальною індустрією, що продовжує стрімко розвиватися. За даними Newzoo аналітичної компанії, яка

спеціалізується на кіберспортивному ринку, у 2022 році глобальна аудиторія кіберспорту сягнула 532 мільйонів осіб, а доходи галузі — понад 920 мільйонів доларів США (рис. 1.1, 1.2) [3]. Це свідчить про зростаючу популярність і серйозне визнання кіберспорту у світовому масштабі. Згадані аспекти були представлені на Міжнародній науково-практичній конференції «Інформаційні технології і автоматизація – 2024» [4].

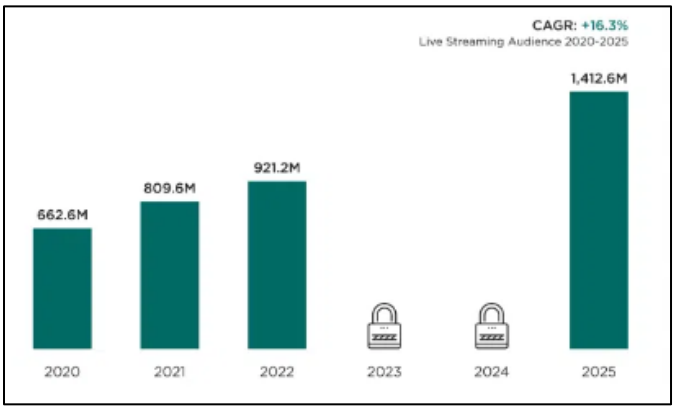


Рисунок 1.1 – Графік росту аудиторії кіберспорту

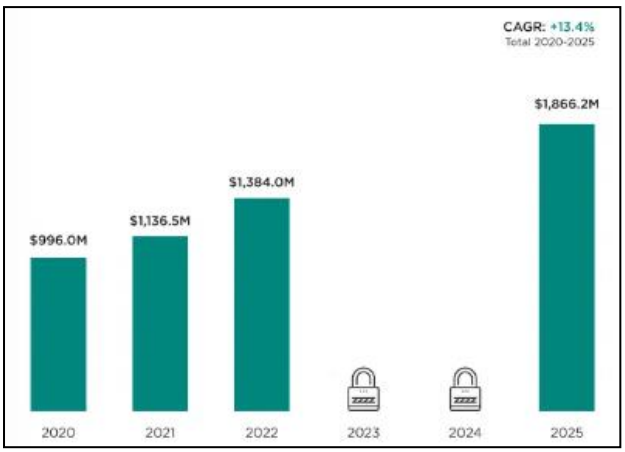


Рисунок 1.2 – Графік росту інвестицій в кіберспорт

Кіберспорт підтримують провідні світові бренди, такі як Coca-Cola, Red Bull, Intel та Nissan. Ключовим чинником його зростаючої популярності є вдосконалення технологій потокової передачі даних, відомих також як стрімінг, що дозволяє мільйонам глядачів спостерігати за турнірами в режимі реального часу [5]. Завдяки стрімінговим платформам, таким як Twitch та YouTube, глядачі мають можливість переглядати змагання з будь-якої точки світу, що сприяє глобалізації кіберспорту та залученню ще більшої аудиторії.

В Україні кіберспорт також набуває значного розвитку. У 2023 році українськомовні трансляції кіберспортивних турнірів зібрали понад 15,5 мільйонів годин перегляду, що на 194% більше порівняно з попереднім роком. Найпопулярнішим турніром став BLAST.tv Paris Major 2023 з гри Counter-Strike: Global Offensive, який зібрав понад 60 000 глядачів на українськомовних трансляціях, встановивши новий рекорд для України [6].

Екосистема кіберспорту складається з різноманітних учасників, кожен з яких відіграє важливу роль у забезпеченні функціонування та розвитку цієї галузі. Основні компоненти екосистеми включають:

- **Гравці та команди**: Професійні та аматорські гравці, які беруть участь у змаганнях та турнірах. Команди складаються з окремих гравців, які спільно розробляють стратегії та тактики для досягнення перемоги.
- **Організатори турнірів**: Компанії та організації, які планують та проводять кіберспортивні змагання. Вони забезпечують необхідну інфраструктуру, правила та фінансування для проведення турнірів.
- **Розробники ігор**: Компанії, що створюють відеоігри, які використовуються в кіберспорті. Вони часто співпрацюють з організаторами турнірів для балансування гри та забезпечення чесної конкуренції.
- **Спонсори**: Бренди та компанії, які фінансово підтримують кіберспортивні події, команди та окремих гравців. Спонсорські угоди допомагають забезпечити призові фонди та інші ресурси для розвитку кіберспорту.
- **Глядачі та фанати**: Аудиторія, яка спостерігає за змаганнями як у режимі реального часу, так і після їх проведення через стрімінгові платформи. Вони створюють додаткові джерела доходів через продаж квитків, мерчандайзинг та рекламу.
- **Медіа та PR-агентства**: Організації, які відповідають за висвітлення подій кіберспорту в ЗМІ, а також за просування та

популяризацію турнірів та гравців .

Рисунок 1.3 демонструє загальний огляд екосистеми кіберспорту та її основних учасників.

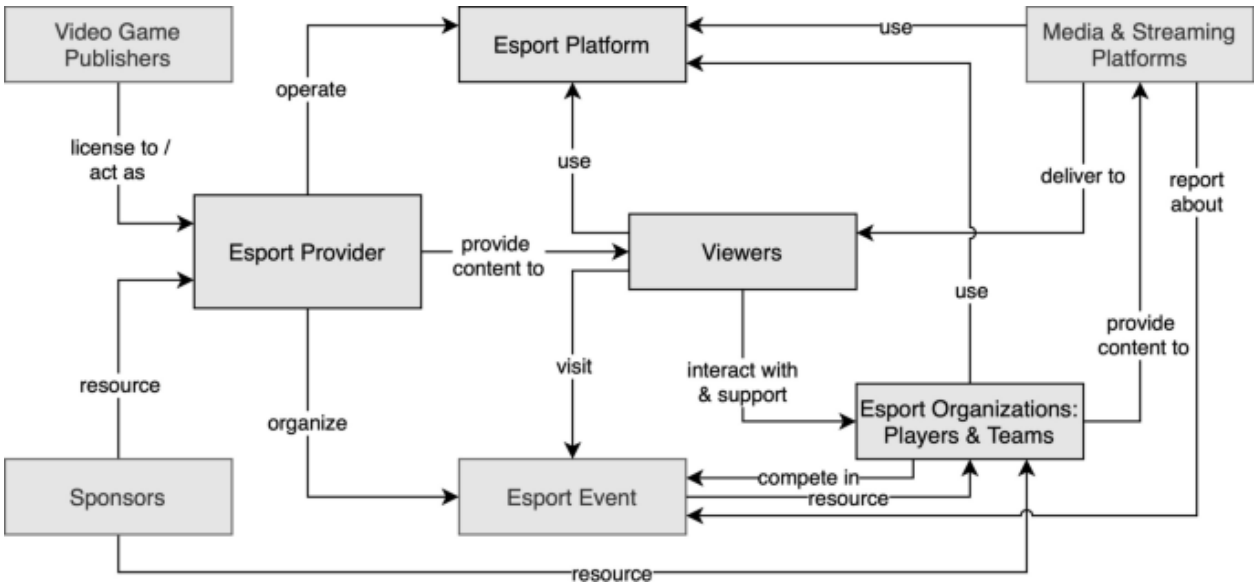


Рисунок 1.3 – Екосистема кіберспорту та її учасники [1]

Попри значний розвиток, кіберспорт стикається з викликами щодо визнання його офіційним видом спорту. Одна з головних перешкод - відсутність єдиного керівного органу та стандартизації правил, що необхідні для інституалізації. Наприклад, на відміну від традиційних видів спорту, де існують національні та міжнародні федерації, як-от Міжнародний олімпійський комітет, у кіберспорті кожна організація пропонує власні змагання та набір правил [7].

Водночас, кіберспорт вже визнаний офіційним видом спорту рядом країн, зокрема в Україні, де він отримав офіційний статус у вересні 2020 року. Це дозволило кіберспортсменам розвивати кар'єру, брати участь у міжнародних змаганнях під національним прапором та отримувати державну підтримку [8].

1.2 Аналіз дисципліни CS2

Counter-Strike 2 (CS2) — це сучасне оновлення легендарного шутера від першої особи, розроблений компанією Valve [9]. Цей проєкт представляє

еволюцію серії Counter-Strike, яка бере свій початок ще з модифікації для гри Half-Life у 1999 році. Завдяки постійному вдосконаленню, франшиза перетворилася на символ кіберспорту, ставши популярною як серед аматорів, так і серед професійних гравців.

У 2018 році Valve зробила вирішальний крок, перетворивши CS:GO (попередню версію гри) на безкоштовну гру (Free-to-Play). Така зміна стала можливою завдяки новій моделі монетизації, заснованій на продажі косметичних предметів, зокрема скінів зброї. Крім того, введення системи Trust Factor, яка знижує ймовірність потрапляння у гру з чітерами (гравцями, що використовують нечесні програми для переваги над суперниками), дозволило зберегти високу якість ігрового процесу.

Гра має різноманітні ігрові карти, що активно використовуються в турнірних матчах, формують так званий пул активних мап, до якого входять Dust 2, Mirage, Inferno, Nuke, Vertigo, Ancient та Anubis. Час від часу набір карт оновлюється: старі локації отримують редизайн і замінюють ті, що потребують модернізації.

1.2.1 Правила та геймплей

У грі змагаються дві команди, кожна з яких складається з п’яти гравців. Одна команда виступає в ролі терористів (Ts), а інша – контр-терористів (Cts). Існують декілька режимів гри, основний вимагає від команд виконання різних завдань: терористи повинні встановити бомбу на визначеній зоні, тоді як контр-терористи мають завдання запобігти встановленню бомби або знешкодити її.

Кожен матч складається з серії раундів, тривалість яких становить 1 хвилину 55 секунд. Гра починається з так званого пістолетного раунду, де кожен гравець має початковий бюджет в 800$, який можна витратити на придбання зброї та обладнання. Економічна система гри відіграє ключову роль, оскільки правильне управління фінансами дозволяє командам купувати кращі види зброї та обладнання в наступних раундах. Гравці можуть

отримувати додаткові кошти за вбивства різних видів зброї, таких як ножі, пістолети, автоматичні гвинтівки, снайперські рушниці та гранати.

Перемога в раунді може бути досягнута шляхом повного знищення команди супротивника або виконання основного завдання, встановлення або знешкодження бомби. Винагороди за перемогу в раунді варіюються в залежності від способу перемоги: встановлення бомби або її вибух приносить більше грошей, ніж просто перемога шляхом вбивства всіх противників. Також існує система серій перемог, яка збільшує отримувані кошти за кожну послідовну перемогу, що мотивує команди підтримувати стабільний і виграшний ігровий процес.

Матчі тривають до 24 раундів, причому команда, яка першою здобуде 13 перемог, оголошується переможцем. При досягненні рівного рахунку 12:12 проводиться перерва, після якої команди змінюють сторони. У разі нічийного результату можуть бути проведені овертайми, що складаються з додаткових раундів для визначення остаточного переможця. Овертайми починаються з максимальної економіки для обох команд, що забезпечує рівні умови для подальшої боротьби.

Важливим аспектом геймплею є перехід гравців від індивідуальної гри до командної взаємодії. Розвиток навичок гравців тісно пов'язаний із їхнім переходом до професійного рівня, де комунікація та стратегічне мислення стають ключовими факторами успіху [10].

Окрім цього, конфігурація обладнання гравців відіграє суттєву роль у їхньому геймплеї. Вибір правильного обладнання, такого як гарнітури для точного визначення напрямку звуків вибухів, впливає на ефективність гри [10].

1.2.2 Професійна сцена

Counter-Strike має розвинену професійну сцену, яка включає численні великі турніри, організовані сторонніми компаніями. Понад 100 турнірів з CS проводяться щороку, залучаючи аудиторію з сотень тисяч глядачів по всьому світу. Призові фонди цих змагань нерідко досягають мільйонів доларів. За

даними щоденної статистики платформи Steam, у CS2 щодня грає понад мільйон гравців, що робить її однією з найпопулярніших ігор на цій платформі [11].

Найпрестижнішою подією в календарі є Major, що проводиться кожен сезон за спонсорства Valve. У цьому турнірі 24 найкращі команди світу змагаються за титул чемпіонів світу та призовий фонд у розмірі $1 000 000. Призові кошти розподіляються між 16 найкращими командами.

Професійну сцену відстежують платформи, такі як HLTV, яка складає рейтинг команд протягом року на основі очок, зароблених на великих турнірах. Цей рейтинг слугує важливим індикатором успіхів команд і впливає на їх запрошення на великі події.

Значна привабливість CS2 для аудиторії пов'язана з високою конкурентністю матчів та їх непередбачуваністю, що стимулює інтерес глядачів. Дослідження показують, що аудиторія особливо зацікавлена у матчах, де рівень суперництва високий, а переможець не є очевидним [11].

1.2.3 Довідкова інформація

Valve Corporation є провідною світовою компанією в індустрії програмного забезпечення для відеоігор. Компанія відома своїми культовими відеоіграми, такими як Half-Life та Counter-Strike, а також створенням Steam — найбільшого у світі онлайн-порталу для геймінгу [12].

Корпорація була заснована колишніми двома співробітниками Microsoft Гейбом Ньюеллом та Майком Харрінгтоном, і цікавий факт — компанія була створена у день весілля Ньюелла [13].

Компанія розпочала з модифікації ігрового двигуна Quake, створивши власний двигун GoldSrc. На його основі у 1998 році було розроблено перший продукт Valve — Half-Life. Гра стала революційною в індустрії, і за версією IGN у 2014 році історія жанру шутерів від першої особи (FPS) поділяється на «до-Half-Life» і «після-Half-Life» періоди [14].

Для підтримки творчості спільноти Valve випустила комплект розробника програмного забезпечення (SDK) для двигуна GoldSrc, який дозволяв користувачам створювати модифікації ігор. Однією з таких модифікацій стала Counter-Strike, яка набула величезної популярності. Valve оцінила її потенціал і найняла розробників, щоб створити окрему гру Counter-Strike, яка згодом стала однією з найвідоміших у світі.

Після успіху Half-Life компанія працювала над модифікаціями, використовуючи новий ігровий двигун Source, та почала розробку Half-Life 2 на цій же платформі. Source також став основою для створення Team Fortress 2 на основі Team Fortress Classic. Ігри Team Fortress 2 та Portal були розроблені студентськими командами, яких Valve найняла для роботи.

Source – 3D-ігровий двигун, розроблений Valve на мові програмування C++. Його розробка триває з червня 2004 року. Основні переваги Source — модульна база, гнучкість, а також технології синхронізації мовлення, вираження емоцій та фізики.

Source 2 — наступне покоління ігрового двигуна від Valve, який був офіційно анонсований у 2015 році. Source 2 базується на успішному фундаменті свого попередника, але пропонує значно покращену графіку, нові технології та оптимізацію для сучасного обладнання.

Окрім створення ігор, Valve розробила Steam — сервіс для розповсюдження відеоігор. Його ідея виникла через необхідність підтримувати оновлення для ігор, таких як Counter-Strike, щоб гравці мали однакові версії. Steam був презентований у 2002 році та створений самою Valve, оскільки інші розробники відмовилися допомогти. Спочатку на платформі були доступні тільки ігри Valve, але пізніше компанія дозволила стороннім розробникам продавати свої продукти, отримуючи частину доходу за доставку контенту. Steam став найпопулярнішим способом придбання цифрових ігор, охоплюючи до 70% всіх цифрових продажів.

1.2.4 Методи збору ігрових даних

Збір ігрових даних є важливим аспектом аналізу відеоігор, професійної сцени, та поведінки гравців. CS2 не надає офіційного API для отримання інформації про матчі в режимі реального часу або після їх завершення. Проте Valve пропонує альтернативні способи інтеграції та отримання ігрових даних через HTTP POST-запити.

Одним із ключових способів збору даних є логування. Цей метод дозволяє налаштувати сервер або клієнт гри для відправлення текстових повідомлень на заданий HTTP-ендпоінт. Завдяки логуванню можна вручну визначати та вибирати цікаві дані для аналізу, що надає гнучкість у зборі та обробці інформації відповідно до специфічних потреб користувача.

Ще одним важливим інструментом збору даних є інтеграція стану гри (Game State Integration, GSI). На відміну від логування, GSI працює через передачу повідомлень у форматі JSON. Однак GSI має обмеження у виборі доступної інформації, оскільки всі дані заздалегідь визначені розробниками гри.

В таблиці 1.1 наведено детальну порівняльну характеристику двох методів збору даних ігрових сесій.

Таблиця 1.1 – Порівняльна таблиця методів збору ігрових даних

Критерій	**Game State Integration (GSI)**	**Логування**
Опис	Метод інтеграції, що дозволяє клієнту гри надсилати JSON-повідомлення про поточний стан гри на локальний або віддалений вебсервер.	Метод збору даних шляхом налаштування ігрових серверів для надсилання журналів подій на визначений сервер через HTTP-запити.
Формат даних	JSON-повідомлення з попередньо визначеними параметрами (позиції гравців, озброєння, статистика раундів тощо).	Журнали подій у вигляді звичайного тексту з часовими мітками

Продовження таблиці 1.1

Критерій	Game State Integration (GSI)	Логування
Гнучкість	Обмежена гнучкість, оскільки всі дані визначені розробниками гри і не можуть бути змінені користувачем.	Висока гнучкість, дозволяє вручну визначати та вибирати потрібні дані для збору та аналізу.
Час передачі даних	Реальний час – дані надсилаються негайно під час гри.	Реальний час.
Легкість налаштування	Просте налаштування через конфігураційні файли GSI, але обмежене можливостями.	Вимагає додаткової конфігурації серверів та налаштування формату журналів, що може бути складнішим для користувачів без технічних знань.
Доступність даних	Обмежений набір даних, доступних для збору, визначений розробниками гри.	Широкий набір даних, який можна налаштувати відповідно до специфічних потреб аналізу.
Використання	Ідеально підходить для базового моніторингу гри та інтеграції з існуючими системами аналізу, які підтримують GSI.	Підходить для глибокого аналізу та кастомізованого збору даних, особливо коли потрібні специфічні або нестандартні метрики.
Обмеження	Неможливість отримання додаткових або нестандартних даних, які не підтримуються GSI.	Може вимагати більше ресурсів для обробки та зберігання великих обсягів журналів, а також потребує більш складної налаштування та підтримки.
Сумісність	Обмежена сумісність лише з тими системами, які можуть обробляти JSON-повідомлення від GSI.	Широка сумісність з різними системами аналізу та зберігання даних, оскільки формат журналів може бути адаптований під різні потреби.
Приклади використання	Відображення статистики гри в реальному часі на сторонніх сервісах, таких як панелі гравців або інтеграція з системами стрімінгу	Глибокий аналіз поведінки гравців, відстеження специфічних подій гри, створення детальних звітів для тренерів або організаторів турнірів

1.3 Огляд існуючих програмних продуктів

Для формування списку функціональних вимог інформаційної технології управління та аналізу кіберспортивних турнірів з дисципліни CS2 необхідно проаналізувати існуючі аналоги. Одним із таких продуктів є eBot — відкритий і популярний додаток для управління турнірами в CS2. Розглянемо його функціональні можливості.

eBot забезпечує автоматизацію процесів, які раніше вимагали ручного втручання. Додаток дозволяє створювати, адмініструвати та відстежувати турніри й матчі, надаючи користувачам доступ до необхідної інформації в режимі реального часу. Платформа орієнтована як на організаторів змагань, так і на звичайних користувачів.

Панель базового користувача в eBot зосереджена на споживанні інформації про матчі та статистику. Користувачі мають доступ до таких розділів, як активні та архівовані матчі, глобальна статистика, а також детальна статистика по картах та конкретних матчах. Наприклад, на домашній сторінці користувач може переглянути список поточних і майбутніх матчів, зокрема їхній статус, команди-учасники, рахунок, карту та сезон (рис. 1.4).

Оновлення рахунків матчів здійснюється в режимі реального часу за допомогою вебсокетів, що дозволяє відображати актуальні дані без необхідності ручного оновлення сторінки.

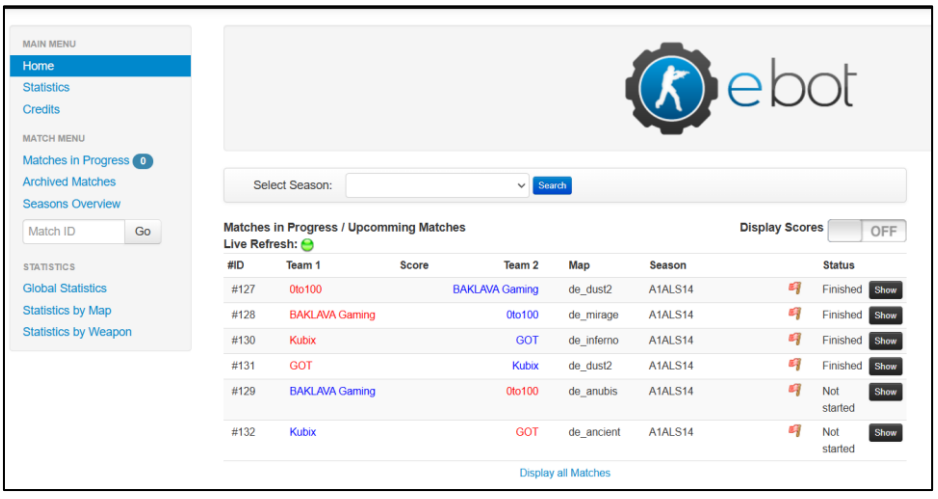
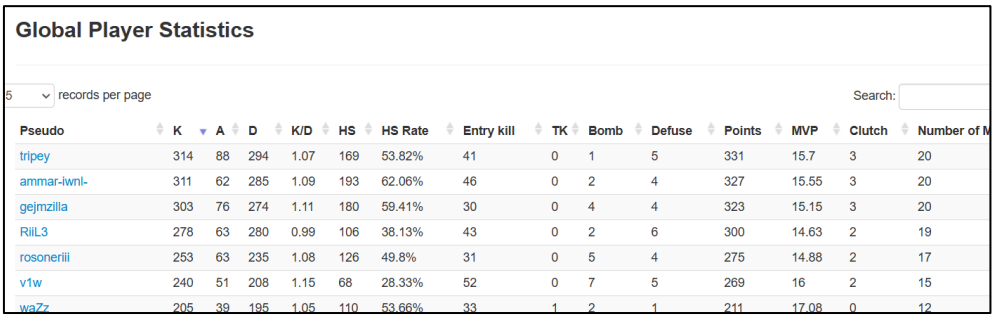


Рисунок 1.4 – Домашня сторінка

У розділі глобальної статистики надається інформація про ефективність гравців, включаючи такі показники, як кількість вбивств, співвідношення K/D, кількість попадань у голову тощо (рис. 1.5).



Pseudo	K	A	D	K/D	HS	HS Rate	Entry kill	TK	Bomb	Defuse	Points	MVP	Clutch	Number of M
tripey	314	88	294	1.07	169	53.82%	41	0	1	5	331	15.7	3	20
ammar-iwnl-	311	62	285	1.09	193	62.06%	46	0	2	4	327	15.55	3	20
gejmzilla	303	76	274	1.11	180	59.41%	30	0	4	4	323	15.15	3	20
RiiL3	278	63	280	0.99	106	38.13%	43	0	2	6	300	14.63	2	19
rosoneriii	253	63	235	1.08	126	49.8%	31	0	5	4	275	14.88	2	17
v1w	240	51	208	1.15	68	28.33%	52	0	7	5	269	16	2	15
waZz	205	39	195	1.05	110	53.66%	33	1	2	1	211	17.08	0	12

Рисунок 1.5 – Глобальна статистика

Крім того, статистика по картах дозволяє аналізувати виграні раунди для обох сторін, а також інші ключові метрики, що допомагають краще розуміти баланс гри на певній мапі (рис. 1.6).

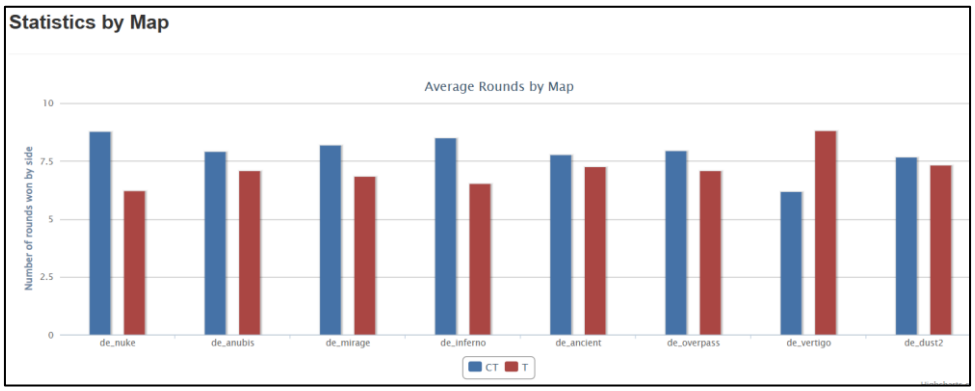


Рисунок 1.6 – Статистика по картах

Детальна інформація про конкретні матчі включає хронологію раундів, статистику зброї, карту активності та навіть унікальні події матчу (рис. 1.7).

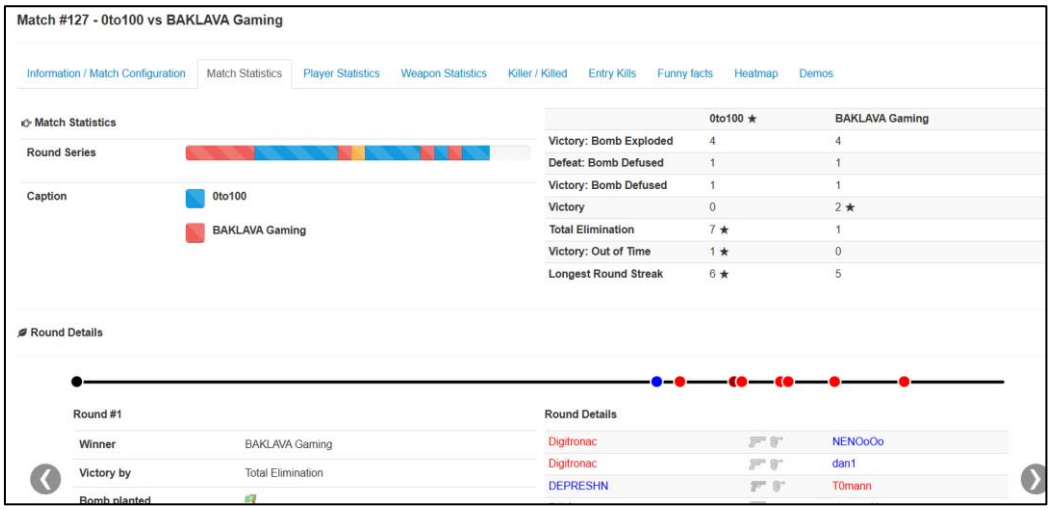


Рисунок 1.7 – Детальна інформація по матчах

Для організаторів eBot пропонує інструменти для створення турнірів і матчів, управління командами, сервером і автоматичного оновлення статистики. Це значно спрощує організацію змагань, забезпечуючи точність та надійність даних.

Наприклад, функціонал створення сезону дозволяє організатору вказати назву турніру, дати початку та закінчення, пов’язану подію, посилання на зовнішній ресурс, завантажити логотип сезону та активувати його (рис. 1.8).

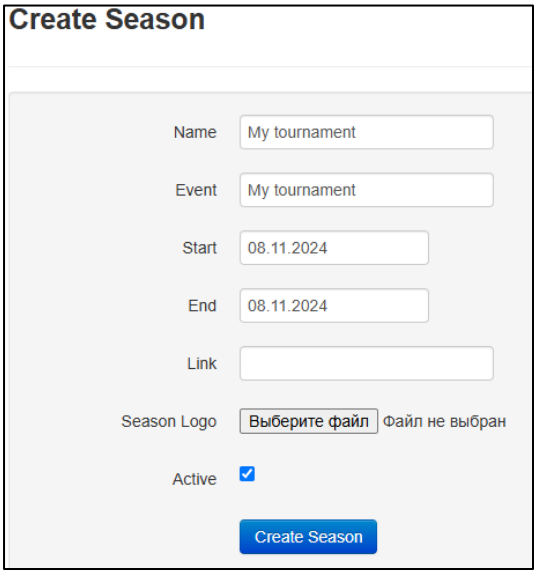


Рисунок 1.8 – Створення сезону

Модуль для створення нового сервера дозволяє вказати IP-адресу сервера, внутрішню назву, пароль RCON і GOTV IP (рис. 1.9).

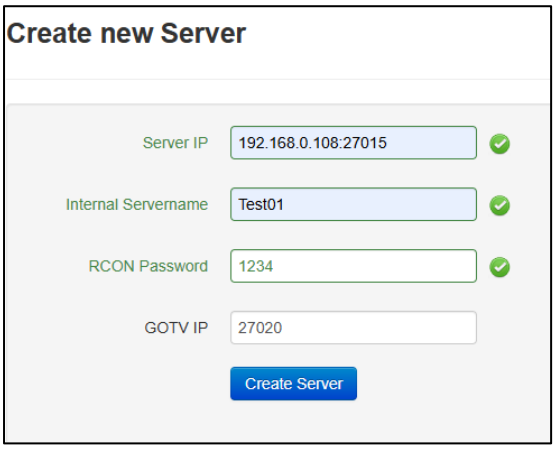


Рисунок 1.9 – Додавання сервера

Функціонал створення команди включає налаштування її назви, скороченого імені, прапора країни, посилання на зовнішній ресурс і прив’язку до сезону (рис. 1.10).

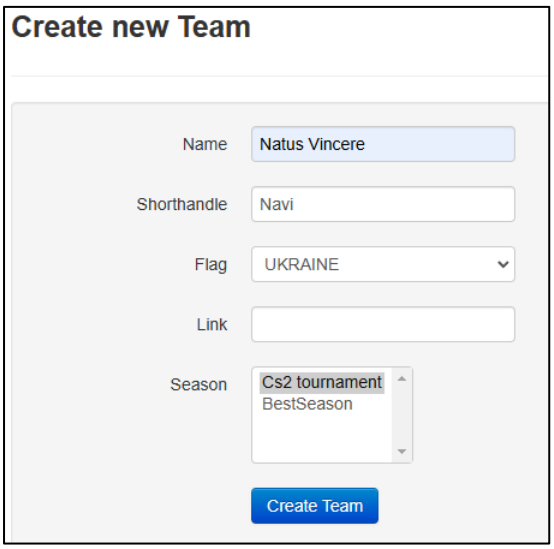


Рисунок 1.10 – Створення команди

Система також надає функціонал для редагування й видалення об'єктів. Наприклад, на сторінці управління командами можна легко змінювати дані або видаляти записи за потребою (рис. 1.11).

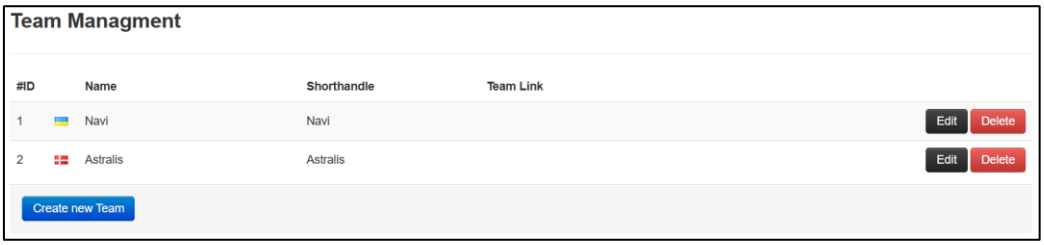


Рисунок 1.11 – Управління командами

Функціонал створення матчу в eBot надає широкі можливості для налаштування ігрових умов. Організатор може обрати команди-учасники, прив’язати матч до певного сезону, задати правила гри, максимальну кількість раундів, пароль для доступу, а також вибрати карту та сервер для проведення. Додатково є можливість налаштувати такі параметри, як проведення ножового раунду, овертайму, автоматичний старт матчу та режим вибору карти (рис. 1.12).

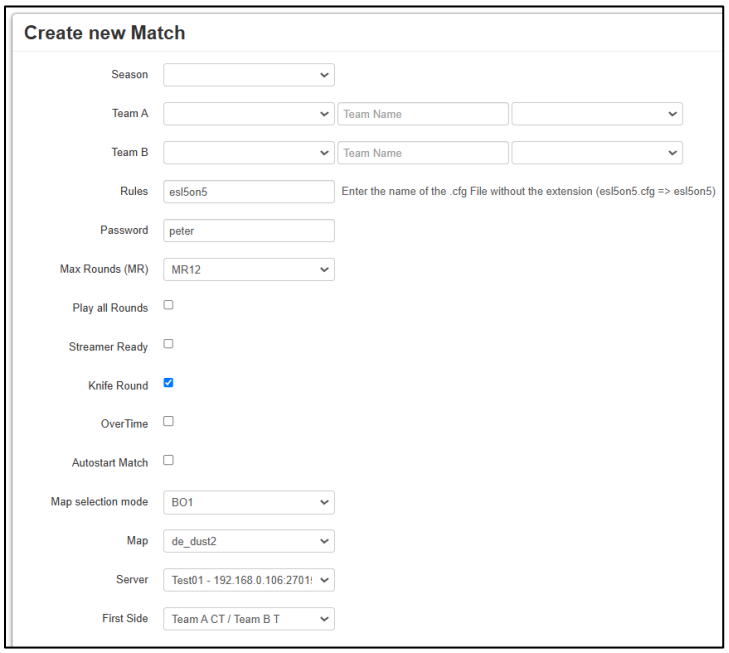


Рисунок 1.12 – Налаштування матчу

Інтерфейс моніторингу матчів у eBot надає організаторам зручний огляд поточних і запланованих матчів. У таблиці відображаються такі дані, як ID матчу, команди-учасники, рахунок, вибрана карта, сезон, хостинг-сервер, пароль для доступу та поточний статус матчу. Додатково організатор має

доступ до функцій управління матчем, таких як зупинка, перезапуск, використання RCON-команд, початок або пропуск ножового раунду (рис. 1.13).

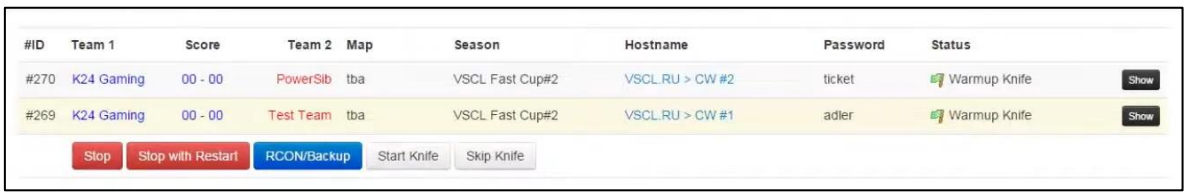


Рисунок 1.13 – Моніторинг матчів

Get5Panel – інший додаток, який буде розглянуто. Хоча Get5Panel не є таким популярним, як eBot, він займає свою нішу на ринку управління кіберспортивними турнірами та матчами.

Get5Panel відрізняється підтримкою авторизації через Steam, що є важливою перевагою. Кожен користувач отримує ідентифікатор Steam ID, який перевіряється під час підключення до ігрового сервера, унеможливлюючи доступ несанкціонованих гравців. Це особливо важливо для професійних турнірів, де безпека є пріоритетом. На відміну від цього, eBot реалізує безпеку через генерацію паролів для доступу до серверів, що є менш інтегрованим, але також ефективним методом.

Що стосується налаштування матчів, Get5Panel демонструє більшу гнучкість порівняно з eBot. Організатори можуть обирати формати матчів, такі як BO1, BO2, BO3, BO5 або навіть BO7. Формат BO (Best Of) визначає максимальну кількість карт у серії, де переможцем стає команда, яка першою виграє більшість карт. Наприклад, у форматі BO3 потрібно виграти дві карти, а у BO5 — три.

Крім цього, Get5Panel дозволяє налаштовувати мап-пул, що дає можливість організаторам визначати набір карт для матчів залежно від специфіки турніру. Це дозволяє створювати більш адаптовані ігрові сценарії, забезпечуючи організаторам значну свободу в управлінні турнірами.

На рисунку 1.14 зображений інтерфейс налаштування матчу в Get5Panel, де можна вибрати формат серії, такі як BO1 чи BO7, і сформувати мап-пул із доступних карт, наприклад, Inferno, Mirage, Nuke тощо.

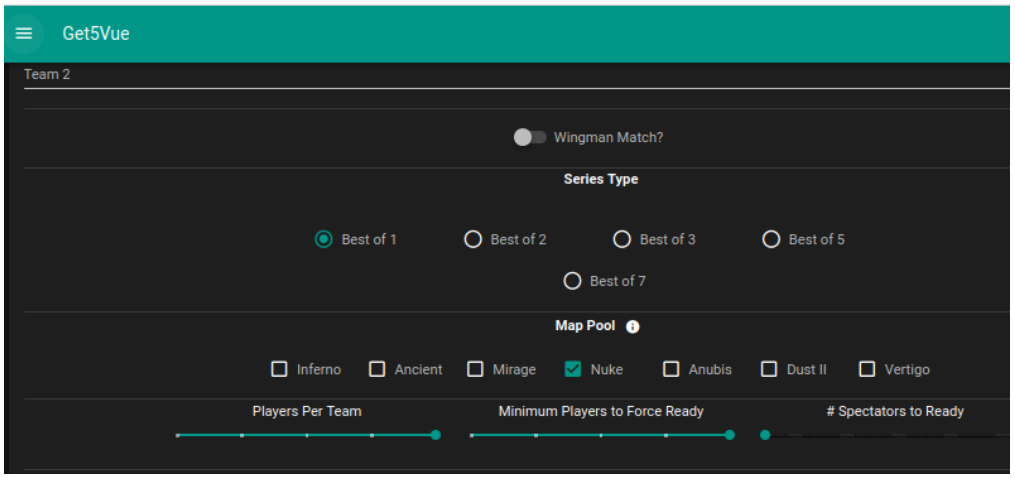


Рисунок 1.14 – Налаштування формату матчів і мап-пулу в Get5Panel

Однак, якщо порівнювати аналітичні можливості, eBot значно перевершує Get5Panel. Як було показана на рисунку 1.7, eBot надає детальний аналіз матчів, включаючи статистику раундів, результати окремих сценаріїв (наприклад, перемога через вибух бомби чи повну ліквідацію), дії гравців і навіть теплові карти активності. Такі глибокі можливості аналізу роблять eBot більш придатним для глибокого розбору матчів і розробки стратегій.

Отже, обидві платформи мають свої переваги, і їхній вибір залежить від конкретних потреб організаторів турнірів. Якщо пріоритетом є глибокий аналіз матчів і стратегічне планування, eBot є кращим вибором. У випадку, коли важливішими є гнучкість налаштувань матчів і інтеграція безпеки через Steam, перевагу слід віддати Get5Panel.

У результаті аналізу програмних продуктів було сформовано ключові вимоги для інформаційної технології управління та аналізу кіберспортивних турнірів з дисципліни CS2. Ці вимоги враховують сильні та слабкі сторони існуючих рішень, таких як eBot та Get5Panel, а також специфічні потреби організаторів турнірів.

Детальна інформація у порівнянні можливостей цих додатків та потенційної власної розробки представлена у таблиці 1.2.

Таблиця 1.2 – Порівняльна таблиця додатків

Функціонал	**eBot**	**Get5Panel**	**Власна розробка**
Авторизація користувачів	Базова	Через Steam	Через Steam
Безпека доступу до серверів	Паролі для серверів	Перевірка Steam ID гравців	Перевірка Steam ID
Налаштування формату матчів	Підтримка BO1	Широкий вибір форматів (BO1-BO7)	Широкий вибір форматів (BO1-BO7)
Мап-пул	Обмежений однією картою	Можливість налаштування мап-пулу	Можливість налаштування мап-пулу
Аналітичні можливості	Глибокий аналіз матчів та статистика	Обмежена статистика	Глибокий аналіз матчів та статистика
Інтерфейс користувача	Інтуїтивний, але застарілий дизайн	Сучасний інтерфейс	Сучасний інтерфейс
Вартість	Безкоштовний	Безкоштовний	Безкоштовний

Отже, була виконана порівняльна характеристика додатків eBot та Get5Panel, що дозволяє визначити їхні переваги та недоліки. Цей аналіз служить основою для розробки власного рішення, яке буде оптимально відповідати специфічним вимогам управління та аналізу кіберспортивних турнірів у CS2.

1.4 Постановка задачі

Сучасна кіберспортивна індустрія, особливо в дисципліні Counter-Strike 2, стикається з відсутністю універсальних та зручних інструментів для комплексної організації турнірів та детального аналізу ігрових даних. Існує потреба в автоматизації процесів управління турнірами, включаючи створення та управління матчами, командами, гравцями, а також збору та аналізу статистичних даних.

Мета роботи полягає в розробці інформаційної технології для автоматизації процесів управління кіберспортивними турнірами та проведення

аналізу ігрових даних у дисципліні Counter-Strike 2, що задовольняє сучасні вимоги індустрії.

Для досягнення цієї мети були визначені такі основні завдання:

1) Дослідити популярність дисципліни Counter-Strike 2 та стрімкий розвиток кіберспортивної індустрії для підтвердження актуальності створення нової інформаційної технології.
2) Здійснити аналіз існуючих систем керування та аналізу кіберспортивних турнірів, виявити їх обмеження та визначити вимоги до розроблюваного рішення.
3) Створити інформаційну технологію для автоматизації управління та аналізу кіберспортивних турнірів з дисципліни CS2:
   - побудувати клієнт-серверну архітектуру з використанням сучасних технологій для забезпечення продуктивності та масштабованості;
   - побудувати діаграму варіантів використання та розробити концептуальну та логічну моделі бази даних для стабільної роботи системи;
   - реалізувати серверну частину з впровадженням функціоналу взаємодії з ігровими серверами через RCON для автоматизації матчів та збору даних;
   - створити клієнтську частину з вебінтерфейсом на основі React.js, що забезпечить зручність використання для адміністраторів та користувачів;
   - впровадити авторизацію через платформу Steam для забезпечення безпеки даних та спрощення процесу аутентифікації.
4) Провести аналіз функціональності створеної системи для перевірки її відповідності поставленим вимогам.

Функціональні вимоги:

- **Управління турнірами:** можливість створення, редагування та

видалення турнірів з вказанням інформації про назву, дати проведення, статус та інші атрибути.

- **Управління командами та гравцями:** можливість додавання, редагування та видалення команд та гравців, призначення гравців до команд.
- **Організація матчів:** автоматизація процесу створення та управління матчами, включаючи призначення команд, карт, часу початку та інших параметрів.
- **Інтеграція з ігровими серверами:** взаємодія з ігровими серверами Counter-Strike 2 через протокол RCON для автоматичного запуску матчів, зміни налаштувань та збору логів.
- **Збір та аналіз ігрових даних:** автоматичний збір логів матчів, їх обробка та збереження у базі даних, надання детальної статистики по гравцях, командах та матчах.
- **Інтерфейс користувача:** зручний вебінтерфейс для адміністраторів та користувачів, що дозволяє взаємодіяти з системою, переглядати інформацію та статистику.
- **Авторизація та безпека:** підтримка авторизації користувачів через платформу Steam, розмежування прав доступу між адміністраторами та звичайними користувачами.

Нефункціональні вимоги:

- **Продуктивність:** система повинна забезпечувати швидку обробку запитів та взаємодію з ігровими серверами без затримок.
- **Масштабованість:** архітектура системи повинна дозволяти легке розширення функціональності та підтримку збільшення кількості користувачів та даних.
- **Надійність та стійкість:** система повинна працювати стабільно, забезпечувати збереження даних та відновлення після можливих збоїв.
- **Безпека:** забезпечення захисту даних, безпечне зберігання паролів

серверів.

- **Юзабіліті:** інтерфейс користувача повинен бути інтуїтивно зрозумілим та зручним у використанні.

2 ВИБІР ЗАСОБІВ РЕАЛІЗАЦІЇ ТА МОДЕЛЮВАННЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ

2.1 Вибір засобів реалізації

Для успішної реалізації інформаційної технології управління та аналізу кіберспортивних турнірів з дисципліни Counter-Strike 2 було обрано сучасний набір технологій та інструментів, які забезпечують високу продуктивність, гнучкість та масштабованість системи.

Серверна частина (бекенд) розробляється за допомогою платформи Node.js та фреймворку Express.js. Node.js є кросплатформним середовищем виконання, що дозволяє запускати JavaScript-код на серверній стороні. Завдяки подієво-орієнтованій, неблокуючій та асинхронній архітектурі Node.js забезпечує високу швидкість обробки великих обсягів даних та ефективне управління мережею, що є критично важливим для систем з високим навантаженням [15]. У даному проекті це особливо актуально, оскільки сервер повинен одночасно обробляти величезну кількість логів, що надходять з різних ігрових серверів, забезпечуючи стабільність та швидкість реакції системи.

Express.js — це мінімалістичний та гнучкий вебфреймворк для Node.js, який надає набір функцій для створення веб та мобільних додатків. Він спрощує процес розробки серверної частини, забезпечуючи ефективну маршрутизацію, підтримку проміжного програмного забезпечення (middleware) та інтеграцію з різними шаблонізаторами для генерації динамічних HTML-сторінок [16].

Клієнтська частина (фронтенд) реалізується за допомогою бібліотеки React.js та менеджера стану Redux. React.js є бібліотекою для створення інтерфейсів користувача, яка використовує компонентний підхід та віртуальний DOM, що підвищує продуктивність та спрощує розробку складних інтерфейсів. React.js дозволяє розробляти великі та складні вебзастосунки, які можуть змінювати дані без перезавантаження сторінки, забезпечуючи швидку та динамічну взаємодію з користувачем [17]. Redux, у

свою чергу, забезпечує керування станом застосунку в передбачуваний спосіб, що особливо важливо для великих та динамічних додатків.

Для роботи з базами даних обрано MySQL — популярну реляційну СУБД, яка відзначається наступними перевагами:

- Висока швидкість та продуктивність: MySQL може працювати на апаратних системах із обмеженими ресурсами, що робить її придатною для високонавантажених проєктів.
- Простота встановлення: MySQL легко встановлюється та конфігурується, а більшість сучасних дистрибутивів Linux вже містять її у стандартному пакеті.
- Підтримка стандартів: MySQL відповідає сучасним стандартам SQL, що забезпечує її інтеграцію в більшість технологічних стеків.
- Масштабованість: Підходить для проєктів різного масштабу — від невеликих додатків до складних корпоративних систем.
- Активна спільнота: Велика база користувачів та розробників, які забезпечують постійний розвиток, підтримку та виправлення помилок.
- Відкритий код: Це дозволяє адаптувати СУБД під потреби конкретного проєкту та економити на ліцензіях [18].

Після вибору MySQL було вирішено використовувати Sequelize ORM для спрощення взаємодії між додатком та базою даних. Sequelize дозволяє працювати з базою даних за допомогою об'єктно-орієнтованого підходу, замінюючи складні SQL-запити на зручні методи на JavaScript [19]. Бібліотека підтримку різних типів баз даних, автоматичну валідацію даних, безпеку від SQL-ін'єкцій, міграцію та багато іншого.

Інтеграція з ігровими серверами здійснюється через RCON (Remote Console), TCP/IP-протокол, який використовується для віддаленого адміністрування серверів Counter-Strike 2. Протокол RCON дозволяє виконувати команди на сервері без прямого доступу до фізичного обладнання. Для забезпечення безпеки кожен запит RCON вимагає попередньої

аутентифікації за допомогою пароля, який задається через змінну `rcon_password` [20]. Запити та відповіді передаються у вигляді структурованих TCP-пакетів, що дозволяє точно контролювати виконання команд і отримувати відповіді навіть у випадку великих обсягів даних.

Збір та обробка даних виконується за допомогою логування та парсингу отриманих журналів подій з серверів гри. Для цього написаний спеціалізований клас, такий як `LogParser`, які аналізує текстові записи логів, виділяють ключові події (наприклад, вбивства гравців, початки та закінчення раундів) та перетворює їх у структуровані дані.

Безпека та авторизація реалізується через інтеграцію з платформою Steam за допомогою OpenID. Це дозволяє додатку автентифікувати SteamID користувача, не вимагаючи від нього введення імені користувача або пароля Steam [21].

У додатку А міститься розділ з плануваннями робіт.

2.2 Архітектура інформаційної системи

Архітектура проєкту побудована на основі клієнт-серверної моделі, де клієнтська та серверна частини взаємодіють через мережу Інтернет. На рисунку 3.1 зображено структуру серверної частини, яка базується на Express REST API.

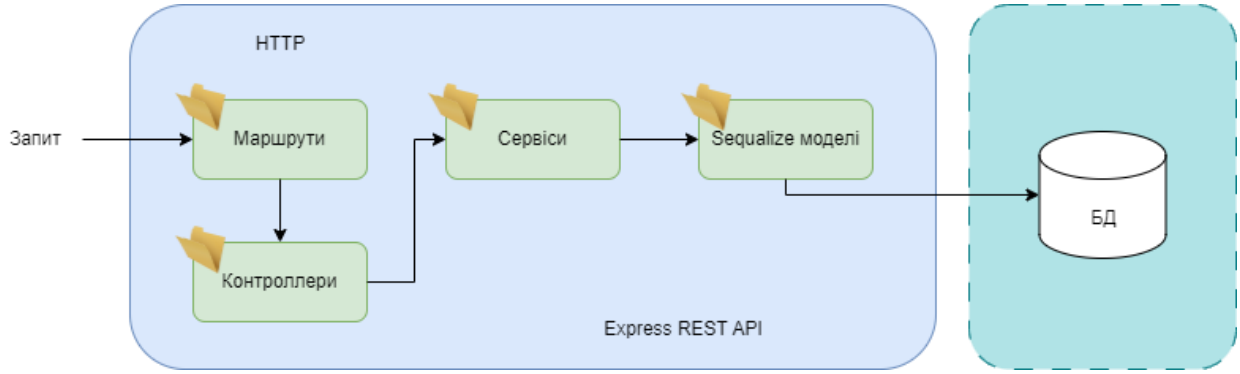


Рисунок 3.1 – Архітектура серверної частини на основі Express REST API

Вона включає маршрути, що обробляють HTTP-запити та спрямовують їх до контролерів. Контролери, у свою чергу, виконують координацію запитів,

звертаючись до сервісів, що містять основну бізнес-логіку. Сервіси взаємодіють із Sequelize-моделями, які відповідають за роботу з базою даних.

На рисунку 3.2 показано модель розподілу функцій між клієнтом і сервером. Клієнт виконує лише презентаційні функції, зосереджуючись на взаємодії з користувачем та відображенні інформації, тоді як сервер реалізує бізнес-логіку, управління базою даних та доступ до СУБД.

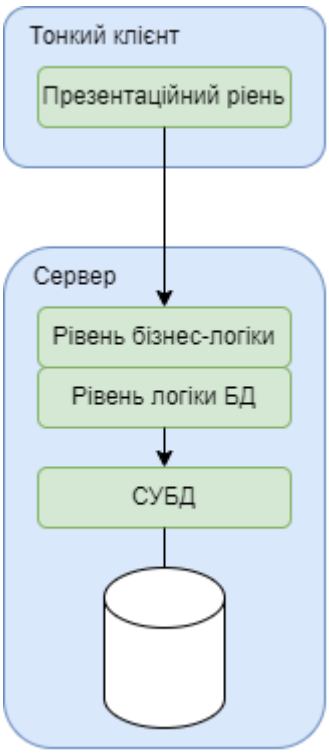


Рисунок 3.2 – Модель розподілу функцій між клієнтом і сервером

Завдяки цьому клієнт залишається максимально легким, а сервер бере на себе основні обчислювальні навантаження, включаючи виконання CRUD-операцій через RESTful API.

2.3 Побудова діаграми варіантів використання

Для того, щоб описати, як відбувається взаємодія користувачів із системою управління та аналізу кіберспортивних турнірів по дисципліні CS2, необхідно спроектувати діаграму варіантів використання. Ця діаграма відображає відносини між користувачами системи (акторами) та

функціональними можливостями системи (варіантами використання) (рис. 3.3).

Діаграма варіантів використання є графічним представленням, що складається з акторів, варіантів використання та зв'язків між ними, обмежених межами системи. Вона допомагає візуалізувати функціональність системи та зрозуміти, як користувачі взаємодіють із системою для виконання своїх завдань. Діаграми варіантів використання є важливим інструментом UML, що широко використовується для моделювання складних об'єктно-орієнтованих систем [22].

Суть цієї діаграми полягає в тому, що система представляється у вигляді сукупності акторів, які взаємодіють із нею через різні варіанти використання. Кожен варіант використання описує певну послугу або функцію, яку система надає актору. Це визначає набір дій, які система виконує під час діалогу з актором, не вдаючись у деталі реалізації цих дій.

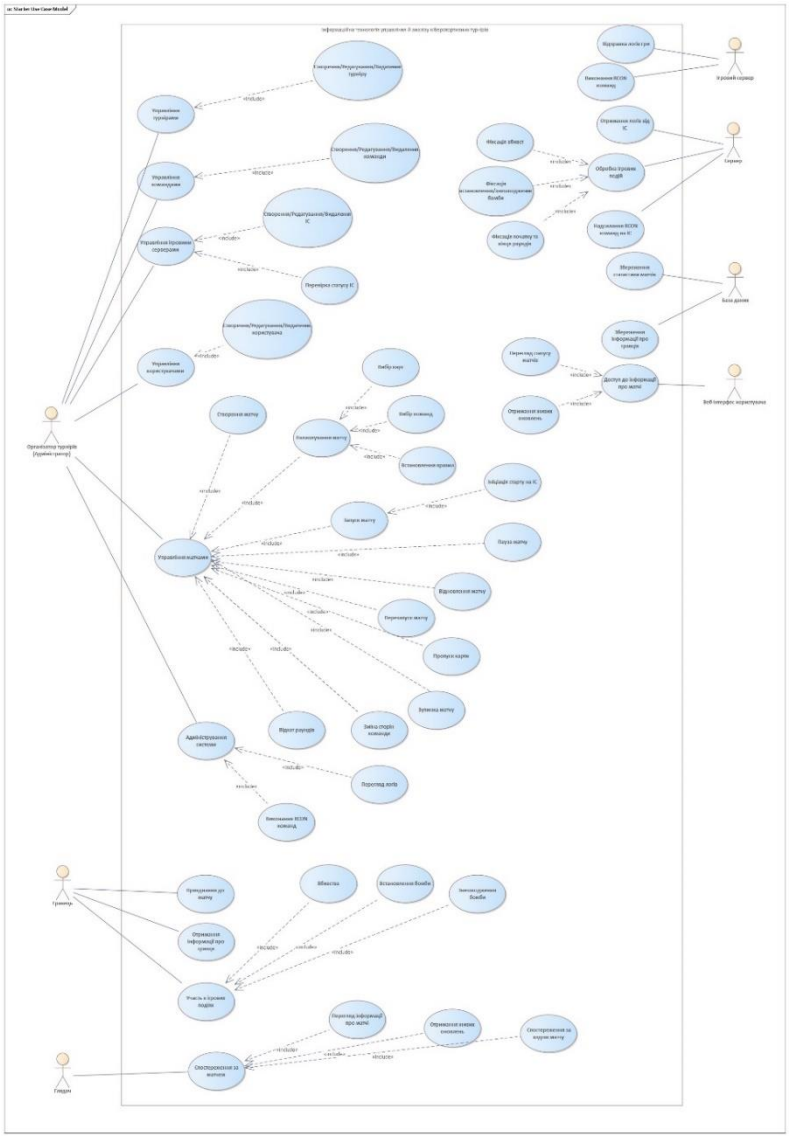


Рисунок 3.3 – Діаграма варіантів використання

2.4 Концептуальний рівень моделювання

Побудова моделей широко застосовується в різних галузях науки і техніки. Наприклад, в авіабудуванні перед тим, як підняти літак в повітря, його зменшену модель досліджують в аеродинамічній трубі. Аналогічно, в інформаційних технологіях, перед фізичною реалізацією БД розробляють її концептуальну та логічну моделі.

Концептуальне (інфологічне) моделювання спрямоване на створення моделі, яка відображає ключові об'єкти (сутності) предметної області, їх характеристики (атрибути) і зв’язки між ними. Це дозволяє створити узагальнене уявлення про дані реального світу, які в майбутньому будуть представлені в базі даних [23].

Концептуальна модель бази даних є високорівневим описом інформаційних потреб системи, незалежним від конкретної системи управління базами даних (СУБД) або технічних деталей реалізації. Графічно відобразимо концептуальну модель на рисунку 3.4.

Для опису даних і сутностей можна використовувати різні підходи, такі як семантичні моделі, графові представлення чи моделі типу «сутність-зв’язок».

Для представлення моделі зазвичай застосовуються UML-діаграми або простий текстовий опис, що є зручним для спілкування з клієнтами, які не мають технічних знань.

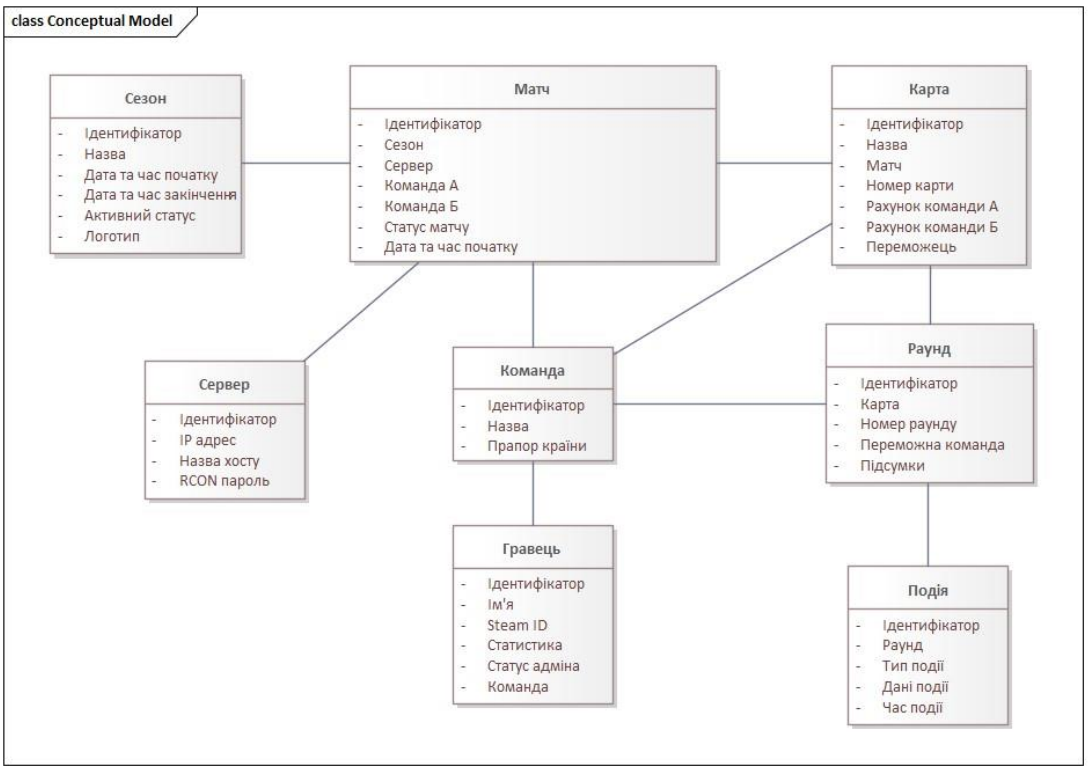


Рисунок 3.4 – Графічна реалізація концептуальної моделі

Концептуальна модель бази даних, представлена на рисунку 3.4, відображає структуру даних, необхідну для організації кіберспортивних змагань. Вона описує ключові сутності, такі як сезон, матч, команда, сервер, карта, раунд, гравець та подія, а також зв’язки між ними.

Ця модель відповідає принципам концептуального моделювання та може бути подалі деталізована для логічної та фізичної реалізації бази даних.

2.5 Логічний рівень моделювання

Логічне (даталогічне) моделювання деталізує концептуальну модель, перетворюючи її на логічну схему. На цьому рівні вже проєктуються сутності, атрибути та зв’язки відповідно до правил, визначених для обраного типу бази даних або навіть конкретної СУБД.

На цьому етапі з'являються основні елементи бази даних: таблиці, поля, ключі, зв’язки, індекси та представлення [23]. Оскільки їхня реалізація залежить від типу бази даних і конкретної СУБД, створити строго абстрактну

модель вже неможливо, і доводиться враховувати особливості обраної системи.

Для візуалізації моделей на цьому рівні зазвичай використовують UML-нотацію, яка поступово витісняє інші підходи, такі як IDEF1X чи нотація Чена.

Використаємо табличне представлення для деталізації сутностей, їх атрибутів та зв'язків між ними у логічній моделі бази даних. В таблиці 3.1 наведено опис сутностей системи.

Таблиця 3.1 – Табличне представлення логічної моделі БД

Таблиця БД	**Поле таблиці**	**Тип даних**	**Коментарій**
event	e_id	BIGINT	Унікальний ідентифікатор події
	e_round	BIGINT	Ідентифікатор раунду, до якого належить подія
	e_type	VARCHAR(50)	Тип події
	e_time	FLOAT	Час події
kill_event	ke_id	BIGINT	Унікальний ідентифікатор події вбивства
	ke_killer_player	BIGINT	Ідентифікатор гравця, що здійснив вбивство
	ke_victim_player	BIGINT	Ідентифікатор вбитого гравця
	ke_weapon	VARCHAR(100)	Зброя, використана для вбивства
	ke_headshot	BOOL	Чи було вбивство пострілом у голову
m2m_match_map	mm_match_map_id	BIGINT	Унікальний ідентифікатор зв'язку матчу та карти
	mm_match	BIGINT	Ідентифікатор матчу
	mm_map	BIGINT	Ідентифікатор карти

Продовження таблиці 3.1

Таблиця БД	**Поле таблиці**	**Тип даних**	**Коментарій**
m2m_match_map	mm_map_order	TINYINT	Порядковий номер карти у матчі
	mm_score_team_a	SMALLINT	Рахунок команди А на цій карті
	mm_score_team_b	SMALLINT	Рахунок команди В на цій карті
	mm_winner_team	BIGINT	Ідентифікатор команди-переможця на цій карті
m2m_match_map_player	mmp_match_map_id	BIGINT	Ідентифікатор матчу
	mmp_player_id	BIGINT	Ідентифікатор гравця
	mmp_kills	INT	Кількість вбивств гравця у матчі
	mmp_deaths	INT	Кількість смертей гравця у матчі
	mmp_assists	INT	Кількість асистів гравця у матчі
	mmp_hsp	INT	Кількість вбивств пострілом у голову
	mmp_kdr	INT	Співвідношення вбивств до смертей
	mmp_adr		Середня шкода, нанесена гравцем за раунд
	mmp_mvp	INT	Кількість нагород MVP, отриманих гравцем
	mmp_ef	INT	Завдана шкода з гранат
	mmp_3k	INT	Кількість зроблених 3K (три вбивства за раунд)
	mmp_4k	INT	Кількість зроблених 4K

Продовження таблиці 3.1

Таблиця БД	**Поле таблиці**	**Тип даних**	**Коментарій**
Match	m_id	BIGINT	Унікальний ідентифікатор матчу
	m_season	BIGINT	Ідентифікатор сезону або турніру
	m_server	BIGINT	Ідентифікатор ігрового серверу
	m_team_a	BIGINT	Ідентифікатор команди A
	m_team_b	BIGINT	Ідентифікатор команди B
	m_status	VARCHAR(20)	Статус матчу
	m_start_date	DATETIME	Дата та час початку матчу
player	p_id	BIGINT	Унікальний ідентифікатор гравця
	p_steam_id	VARCHAR(100)	Ідентифікатор Steam гравця
	p_name	VARCHAR(50)	Ім'я гравця
	p_is_admin	BOOL	Чи є гравець адміністратором системи
round	r_id	BIGINT	Унікальний ідентифікатор раунду
	r_match_map	BIGINT	Ідентифікатор зв'язку матчу та карти
	r_number	SMALLINT	Номер раунду в матчі
	r_winning_team	BIGINT	Ідентифікатор команди-переможця раунду
	r_win_type	VARCHAR(255)	Тип перемоги
m2m_match_map_player	mmp_5k	INT	Кількість зроблених 5K

Продовження таблиці 3.1

Таблиця БД	**Поле таблиці**	**Тип даних**	**Коментарій**
m2m_match_map_player	mmp_clutchk	INT	Кількість виграних клатчів
	mmp_firstk	INT	Кількість перших вбивств у раундах
	mmp_pistolk	INT	Кількість вбивств з пістолетів
	mmp_sniperk	INT	Кількість вбивств з снайперських гвинтівок
	mmp_blindk	INT	Кількість вбивств противників у сліпому стані
	mmp_firedmg	INT	Шкода, нанесена вогнем
	mmp_defuse	INT	Кількість знешкоджених бомб
	mmp_bombe	INT	Кількість встановлених бомб
m2m_team_player	tp_team	BIGINT	Ідентифікатор команди
	tp_player	BIGINT	Ідентифікатор гравця
	tp_join_date	DATETIME	Дата приєднання гравця до команди
	tp_leave_date	DATETIME	Дата виходу гравця з команди
map	m_id	BIGINT	Унікальний ідентифікатор карти
	m_name	VARCHAR(50)	Назва карти
round_summary	rs_id	BIGINT	Унікальний ідентифікатор підсумку раунду
	rs_score_team_a	INT	Рахунок команди A після раунду

Продовження таблиці 3.1

Таблиця БД	**Поле таблиці**	**Тип даних**	**Коментарій**
round_summary	rs_score_team_b	INT	Рахунок команди B після раунду
season	s_id	BIGINT	Унікальний ідентифікатор сезону
	s_name	VARCHAR(50)	Назва сезону або турніру
	s_start_date	DATETIME	Дата початку сезону
	s_end_date	DATETIME(4)	Дата завершення сезону
	s_active	BOOL	Чи є сезон активним
	s_logo	VARCHAR(255)	Шлях до логотипу сезону
server	s_id	BIGINT	Унікальний ідентифікатор серверу
	s_ip_address	VARCHAR(50)	IP-адреса ігрового серверу
	s_rcon_password	VARCHAR(50)	RCON-пароль для доступу до серверу
	s_hostname	VARCHAR(100)	Ім'я хоста або назва серверу

Тепер представимо отримавшу модель графічно в нотації UML (рис. 3.5)

.

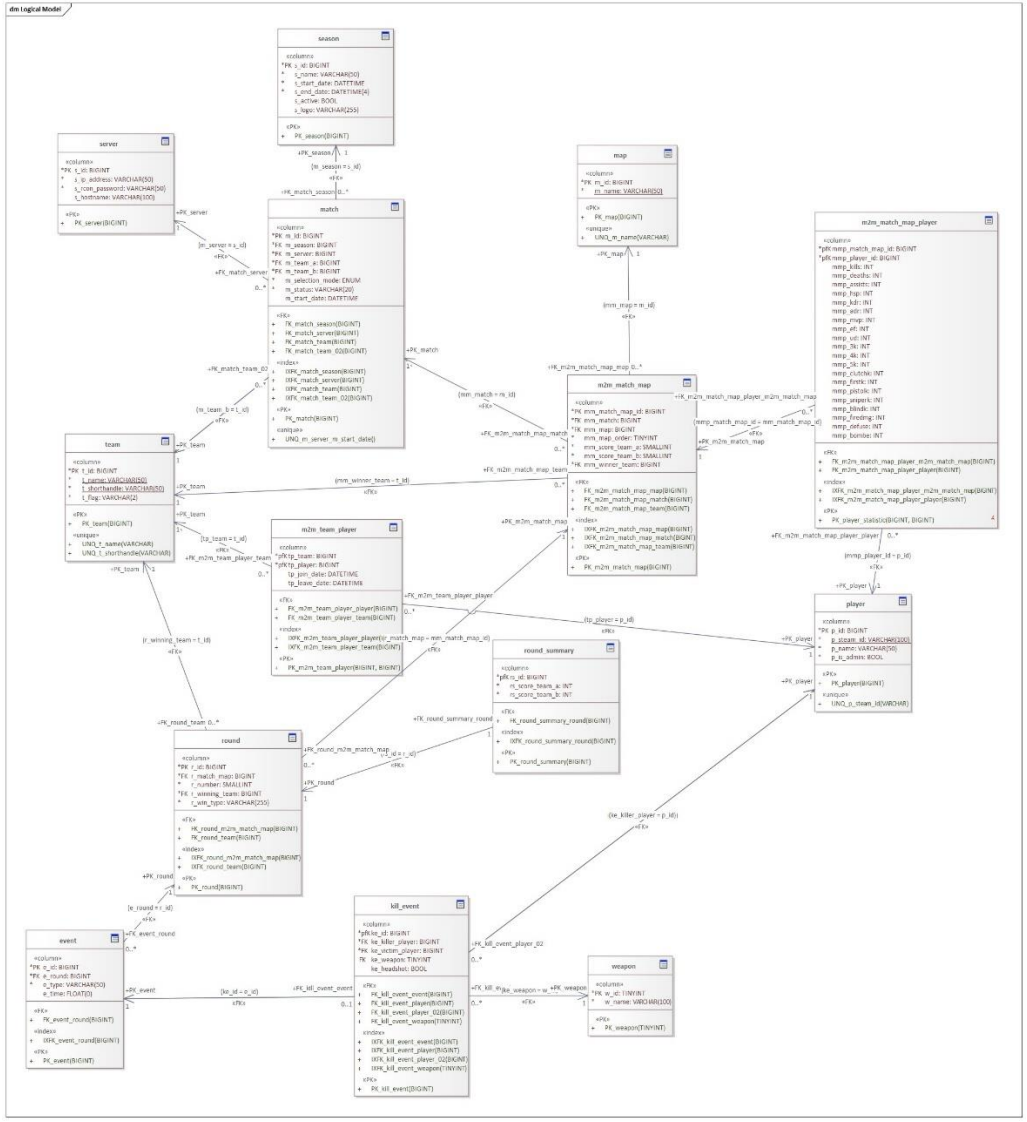


Рисунок 3.5 – Графічне представлення логічної моделі в нотації UML

3 РОЗРОБКА ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ

3.1 Створення БД

Створення бази даних було реалізовано за допомогою реляційної системи управління базами даних MySQL. Для роботи використано програмний інтерфейс MySQL Workbench, який забезпечує зручність у створенні та адмініструванні бази даних.

Процес створення бази даних розпочався з налаштування з'єднання з сервером MySQL через MySQL Workbench. Після відкриття програми було створено нове підключення, вказавши необхідні параметри доступу, такі як хост, порт, ім'я користувача та пароль.

Потрапивши на головний екран програми, перемістимо курсор на бокове меню зі списком схем і у вільному місце натиснимо праву кнопку миші для визову контексного меню. Обираємо пукнт «Create Schema…» (рис. 4.1).

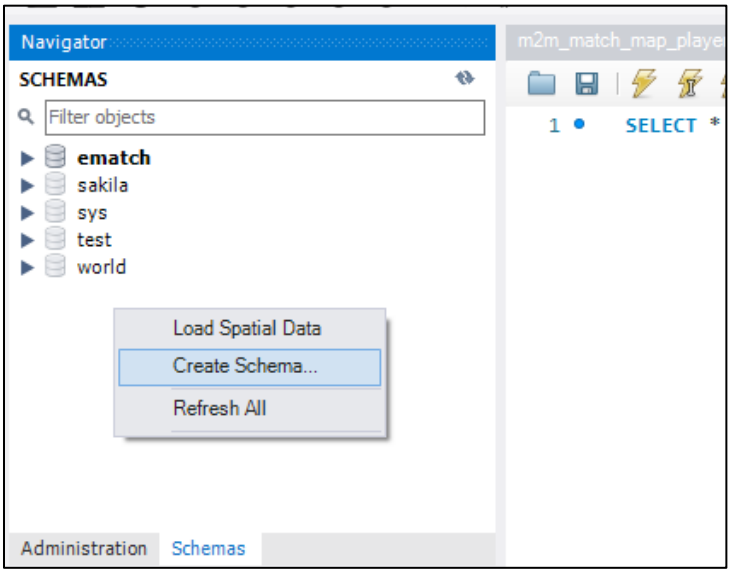


Рисунок 4.1 – Створення БД

У новому вікні вказуємо назву бази даних «ematch» та натискаємо кнопку «Apply» (рис. 4.2).

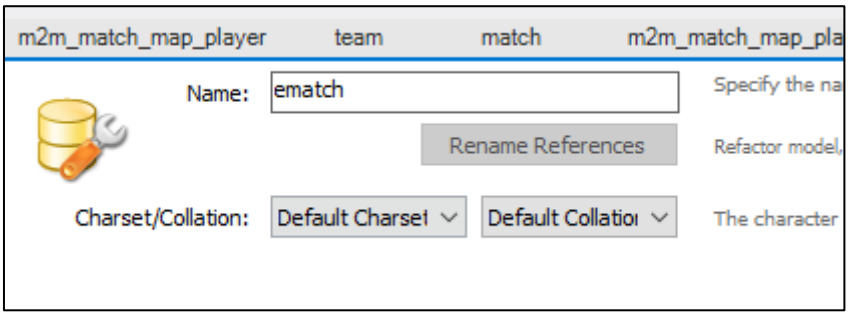


Рисунок 4.2 – Введення назви БД

Процес подальшої роботи з базою даних буде здійснюватися за допомогою ORM бібліотеки Sequelize, який забезпечить ефективне управління моделями та їх взаємозв'язками.

3.2 Ініціалізація сервера

Ініціалізація сервера була виконана за допомогою фреймворку Express.js, який забезпечує ефективне управління маршрутами та middleware для обробки HTTP-запитів. Для цього було створено основний файл сервера `index.js`, де налаштовано основні параметри застосунку, підключено необхідні middleware, такі як `CORS`, `cookie-parser`, `express-session`, а також інтегровано `Passport.js` для автентифікації користувачів. Також було інтегровано маршрутизатор для обробки API-запитів та середовище для обробки помилок. Нижче наведено основний код сервера:

```
require('dotenv').config();
const express = require('express');
const cors = require('cors');
const cookieParser = require('cookie-parser');
const session = require('express-session');
const passport = require('./middlewares/passport');
const { sequelize, connectDB } = require('./db/sequelize');
const router = require('./router');
const errorMiddleware =
require('./middlewares/errorMiddleware');

const app = express();
const PORT = process.env.PORT || 5000;

app.use(cors({
    origin: process.env.CLIENT_URL,
    credentials: true,
}));
```

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

app.use(session({
    secret: process.env.SESSION_SECRET || 'your_session_secret',
    resave: false,
    saveUninitialized: false,
    cookie: {
        httpOnly: true,
        secure: false,
        maxAge: 24 * 60 * 60 * 1000,
    },
}));

app.use(passport.initialize());
app.use(passport.session());

app.use('/api', router);
app.use(errorMiddleware);

const start = async () => {
    try {
        await connectDB();
        await sequelize.sync();
        app.listen(PORT, () => console.log(`Server started on
PORT: ${PORT}`));
    } catch (e) {
        console.error('Помилка запуску сервера:', e);
    }
};

start();
```

3.3 Підключення до БД

Конфігурація підключення здійснювалася за допомогою файлу `db/sequelize.js`, де були визначені необхідні параметри доступу, такі як назва бази даних, ім'я користувача, пароль, хост та порт. Ці параметри зчитуються з файлу `.env`, що зберігає змінні середовища.

Нижче наведено приклад коду для встановлення з'єднання з базою даних:

```
require('dotenv').config();
const { Sequelize } = require('sequelize');

const sequelize = new Sequelize(
    process.env.DB_NAME,
    process.env.DB_USER,
    process.env.DB_PASSWORD,
```

```
    {
        host: process.env.DB_HOST,
        port: process.env.DB_PORT,
        dialect: 'mysql',
        logging: false,
    }
);

const connectDB = async () => {
    try {
        await sequelize.authenticate();
        console.log('Підключення до бази даних успішно
встановлено');
    } catch (e) {
        console.log('Не вдалося підключитися до бази даних:',
e);
    }
};

module.exports = { sequelize, connectDB };
```

3.4 Створення моделей

Створення моделей було здійснено за допомогою ORM-бібліотеки Sequelize, яка перетворює таблиці бази даних у JavaScript-об'єкти. Кожна модель представляє окрему сутність системи та визначає її поля та взаємозв'язки з іншими моделями (рис. 4.3).

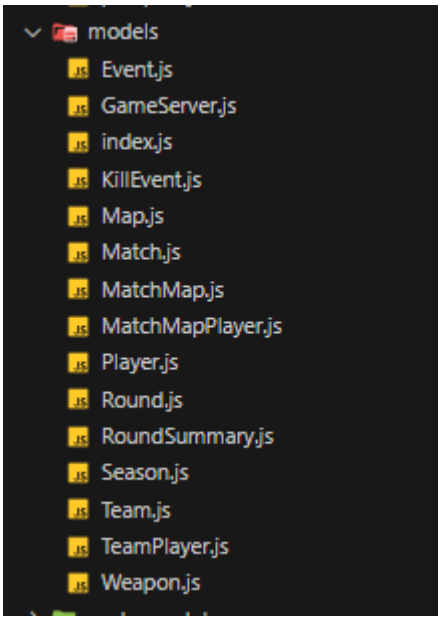


Рисунок 4.3 – Структура моделей бази даних для управління турнірами

В таблиці 4.1 представлено опис кожної сутності, її призначення та основні характеристики.

Таблиця 4.1 – Опис моделей БД

Модель	**Опис**
Event.js	Представляє події, що відбуваються під час матчу, наприклад, вбивства гравців.
GameServer.js	Містить інформацію про ігрові сервери, які використовуються в турнірах, зокрема IP-адресу, пароль, назву.
KillEvent.js	Записує інформацію про вбивства під час гри, включаючи виконавця, жертву, зброю і факт здійснення вбивства в голову.
Map.js	Описує мапи (карти), на яких проходять матч.
Match.js	Відображає матчі, зокрема команди, карти, формат гри (BO1, BO3 тощо) та статус.
MatchMap.js	Пов'язує матчі з картами, описує порядок і стан кожної карти в матчі, включаючи сторони команд.
MatchMapPlayer.js	Зберігає статистику гравців для кожної карти, зокрема кількість вбивств, смертей і допомог.
Player.js	Містить інформацію про гравців, включаючи Steam ID, ім'я, роль у турнірі та статус адміністратора.
Round.js	Описує окремі раунди матчу, їхній номер, команду-переможця та тип перемоги.
RoundSummary.js	Зберігає підсумкову інформацію про раунди, як-от рахунок команд та загальний час раунду.
Season.js	Містить інформацію про турнірні сезони, їхню назву, дати початку та завершення.
Team.js	Відображає команди, що беруть участь у турнірі, включаючи їх назви, шорт-хендли та прапори.
TeamPlayer.js	Зберігає інформацію про склад команд, зокрема дату приєднання та виходу гравців із команди.
Weapon.js	Містить дані про зброю, яка використовується у матчах, включаючи її назву.

Наприклад, модель `Player` відповідає таблиці `players` в базі даних і містить інформацію про гравців турнірів:

```
const { DataTypes } = require('sequelize');
const { sequelize } = require('../db/sequelize');

const Player = sequelize.define('Player', {
    id: {
        type: DataTypes.BIGINT,
        primaryKey: true,
        autoIncrement: true,
    },
    steamId: {
        type: DataTypes.STRING(100),
        allowNull: false,
        unique: true,
    },
    name: {
        type: DataTypes.STRING(50),
        allowNull: false,
    },
    isAdmin: {
        type: DataTypes.BOOLEAN,
        defaultValue: false,
    },
}, {
    tableName: 'players',
    timestamps: false,
});

module.exports = Player;
```

У даній моделі визначені основні поля, такі як унікальний ідентифікатор гравця, Steam ID, ім'я та статус адміністратора. Для встановлення зв'язків між моделями використовуються асоціації Sequelize. Наприклад, модель `Player` пов'язана з моделлю `Team` через таблицю `m2m_team_player`, що реалізує зв'язок "багато-до-багатьох":

```
const Player = require('./Player');
const Team = require('./Team');
const TeamPlayer = require('./TeamPlayer');

Player.belongsToMany(Team, { through: TeamPlayer, as: 'teams',
foreignKey: 'playerId' });
Team.belongsToMany(Player, { through: TeamPlayer, as: 'players',
foreignKey: 'teamId' });
```

3.5 Авторизація через Steam

Для забезпечення безпечної та зручної авторизації користувачів було обрано інтеграцію з платформою Steam. Крім цього, для організації та проведення турнірних матчів при підключенні гравця до ігрового сервера перевіряється його Steam ID, що дозволяє підтвердити його ідентичність та гарантувати безпеку проведення змагань.

Інтеграція авторизації через Steam була реалізована за допомогою `Passport.js` — популярної бібліотеки для автентифікації в Node.js додатках. Використання стратегії Passport-Steam дозволяє легко налаштувати процес автентифікації через Steam OpenID.

Першим кроком було встановлення необхідних залежностей:

```
npm install passport passport-steam express-session
```

Далі було налаштовано `Passport.js` з використанням стратегії Steam у файлі `middlewares/passport.js`:

```
const passport = require('passport');
const SteamStrategy = require('passport-steam').Strategy;
const Player = require('../models/Player');

passport.serializeUser((user, done) => {
    done(null, user.p_id);
});

passport.deserializeUser(async (id, done) => {
    try {
        const user = await Player.findByPk(id);
        done(null, user);
    } catch (err) {
        done(err, null);
    }
});

passport.use(new SteamStrategy({
    returnURL: process.env.STEAM_RETURN_URL,
    realm: process.env.STEAM_REALM,
    apiKey: process.env.STEAM_API_KEY
},
async (identifier, profile, done) => {
    try {
        const steamId = profile.id;
        let user = await Player.findOne({ where: { p_steam_id:
steamId } });
```

```
        if (!user) {
            user = await Player.create({
                p_steam_id: steamId,
                p_name: profile.displayName,
                p_is_admin: false
            });
        }
        return done(null, user);
    } catch (error) {
        return done(error, null);
    }
}));

module.exports = passport;
```

У цьому коді визначено стратегію Steam, де зазначено URL для повернення після автентифікації, область застосування (realm) та API ключ Steam. Під час автентифікації перевіряється, чи існує користувач з відповідним Steam ID у базі даних. Якщо ні, створюється новий запис у таблиці `Player`.

3.6 Маршрутизація

Для організації обробки HTTP-запитів у системі було налаштовано маршрутизацію за допомогою Express.js. Маршрути відповідають за різні функціональні області додатку, забезпечуючи структурований та логічний розподіл запитів між відповідними контролерами. Основний маршрутизатор був створений у файлі `router/index.js`, де визначено всі необхідні маршрути для обробки запитів від клієнта.

Для захисту певних маршрутів та забезпечення доступу лише авторизованим користувачам було створено middleware-функції `authMiddleware` та `adminMiddleware` у файлах `middlewares/authMiddleware.js` та `middlewares/adminMiddleware.js` відповідно. Ці middleware перевіряють, чи користувач автентифікований та чи має він адміністративні права перед доступом до захищених маршрутів.

Приклад маршрутизатора у файлі `router/index.js`:

```
const Router = require('express').Router;
const playerController =
require('../controllers/playerController');
const gameServerController =
```

```
require('../controllers/gameServerController');
const seasonController =
require('../controllers/seasonController');
const teamController = require('../controllers/teamController');
const matchController =
require('../controllers/matchController');
const logController = require('../controllers/logController');
const passport = require('../middlewares/passport');
const authMiddleware = require('../middlewares/authMiddleware');
const adminMiddleware =
require('../middlewares/adminMiddleware');

const router = new Router();

// Маршрути для гравця
router.get('/auth/steam', passport.authenticate('steam'));
router.get('/auth/steam/return', passport.authenticate('steam',
{ failureRedirect: '/' }), playerController.steamCallback);
router.get('/me', playerController.getCurrentUser);
router.post('/logout', playerController.logout);
router.get('/players/leaders',
playerController.getPlayerLeaders);

// Маршрути для серверів
router.post('/servers', authMiddleware, adminMiddleware,
gameServerController.createServer);
router.get('/servers', authMiddleware, adminMiddleware,
gameServerController.getAllServers);
router.get('/servers/:id', authMiddleware, adminMiddleware,
gameServerController.getServerById);
router.put('/servers/:id', authMiddleware, adminMiddleware,
gameServerController.updateServer);
router.delete('/servers/:id', authMiddleware, adminMiddleware,
gameServerController.deleteServer);
router.get('/servers/:id/status',
gameServerController.checkServerStatus);

// ... Інші маршрути
module.exports = router;
```

Приклад middleware-функції `middlewares/authMiddleware.js`:

```
const ApiError = require('../exceptions/apiError');
module.exports = function(req, res, next) {
    if (req.isAuthenticated()) {
        return next();
    }
    return next(ApiError.Unauthorized());
};
```

3.7 Контролери

Контролери відповідають за прийом запитів, виклик відповідних сервісів для обробки бізнес-логіки та повернення відповідей клієнту (рис. 4.4).

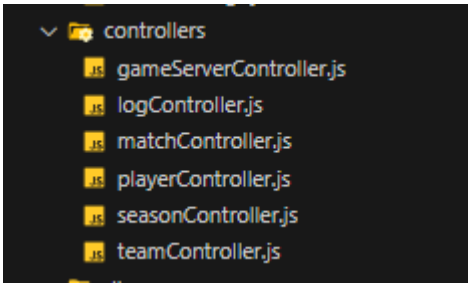


Рисунок 4.4 – Структура контролерів проєкту

В таблиці 4.2 представлено опис кожного контролера, його призначення та основні характеристики.

Таблиця 4.2 – Опис контролерів

Контролер	Опис
gameServerController.js	Відповідає за управління ігровими серверами, такими як створення, видалення, оновлення інформації та перевірка статусу серверів.
logController.js	Обробляє лог-файли серверів, витягує ключові дані про події гри для подальшого аналізу.
matchController.js	Керує матчами: створення, оновлення, отримання списку активних, завершених чи архівних матчів.
playerController.js	Відповідає за управління інформацією про гравців, включаючи створення профілів, отримання лідерів або деталей про гравців.
seasonController.js	Керує інформацією про сезони, включаючи створення, редагування, отримання деталей та видалення сезонів.
teamController.js	Відповідає за управління командами, включаючи створення, оновлення, отримання лідерів та деталей про команди.

Приклад контролера для управління гравцями `controllers/playerController.js`:

```
const PlayerService = require('../services/playerService');

exports.getAllPlayers = async (req, res) => {
    try {
        const players = await PlayerService.getAllPlayers();
        res.status(200).json(players);
    } catch (error) {
        res.status(500).json({ message: 'Помилка при отриманні
гравців' });
    }
};

exports.createPlayer = async (req, res) => {
    try {
        const { steamId, name, isAdmin } = req.body;
        const newPlayer = await PlayerService.createPlayer({
steamId, name, isAdmin });
        res.status(201).json(newPlayer);
    } catch (error) {
        res.status(500).json({ message: 'Помилка при створенні
гравця' });
    }
};

// Інші методи: updatePlayer, deletePlayer
```

У цьому прикладі контролер `playerController` використовує сервіс `PlayerService` для виконання операцій з гравцями. Це дозволяє контролерам залишатися тонкими та зосередженими на обробці запитів та відповідей, делегуючи бізнес-логіку сервісам.

3.8 Сервіси

Сервіси відповідають за реалізацію бізнес-логіки системи та взаємодію з моделями бази даних (рис. 4.5). Вони забезпечують абстракцію між контролерами та моделями, дозволяючи повторно використовувати логіку та спрощувати тестування.

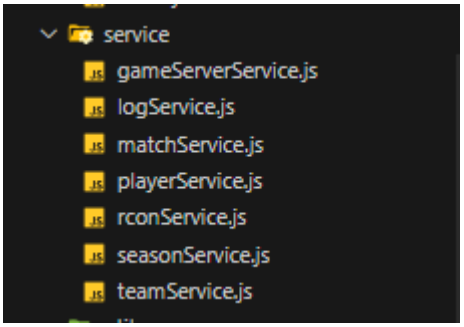


Рисунок 4.5 – Структура сервісів проєкту

В таблиці 4.3 представлено опис кожного сервіса, його призначення та основні характеристики.

Таблиця 4.3 – Опис сервісів

Сервіс	**Опис**
gameServerService.js	Реалізує логіку для управління ігровими серверами, включаючи інтеграцію з RCON для налаштування серверів.
logService.js	Відповідає за управління логами матчів: створює лог-файли для кожного матчу, зберігає логи у файловій системі та надає можливість обробляти логи через слухачі (listeners).
matchService.js	Реалізує бізнес-логіку матчів, включаючи підготовку серверів, управління статусом матчів та обробку результатів гри.
playerService.js	Обробляє дані гравців, зокрема автентифікацію, збереження статистики та управління ролями користувачів.
rconService.js	Забезпечує підключення до ігрових серверів через RCON для виконання команд та управління ігровими процесами.
seasonService.js	Відповідає за логіку управління сезонами, зокрема визначення активних сезонів та збереження їхніх параметрів.
teamService.js	Реалізує функціонал для управління командами, їхнім складом, а також інтеграцію з турнірною статистикою.

Приклад сервісу для управління гравцями:

```
const Player = require('../models/Player');

exports.getAllPlayers = async () => {
    return await Player.findAll();
};

exports.createPlayer = async ({ steamId, name, isAdmin }) => {
    return await Player.create({ p_steam_id: steamId, p_name:
name, p_is_admin: isAdmin });
};

// Інші методи: updatePlayer, deletePlayer
```

У цьому прикладі сервіс `PlayerService` містить методи для отримання всіх гравців та створення нового гравця. Він взаємодіє з моделлю `Player`, виконуючи необхідні операції з базою даних. Контролери викликають ці методи, аби виконати відповідні дії, що забезпечує чітке розділення обов'язків між різними частинами системи.

3.9 Інтеграція ігрових серверів

Інтеграція ігрових серверів реалізована через протокол RCON, що забезпечує віддалене управління сервером, виконання команд і моніторинг стану. Для цього використано бібліотеку `rcon-srcds`. Параметри доступу (IP, порт, пароль) зберігаються у зашифрованому вигляді в базі даних.

Функціонал інтеграції включає перевірку стану серверу, застосування спеціальних конфігурацій для етапів матчу, таких як розігрів, ножовий раунд або основна гра, а також обробку логів матчів. Наприклад, для перевірки стану серверу використовується метод, який викликає команду `status` через клієнт RCON. Код цієї функції виглядає наступним чином:

```
const rconService = require('./services/rconService');
const { GameServer } = require('./models');

async function checkServerStatus(serverId) {
    const server = await GameServer.findByPk(serverId);
    if (!server) {
        throw new Error('Сервер не знайдено');
    }

    const rcon = await rconService.createRconClient(server);
```

```
    try {
        const response = await rcon.execute('status');
        await rcon.disconnect();
        return { online: true, response };
    } catch (error) {
        return { online: false, error: error.message };
    }
}
```

Конфігурації для різних етапів матчу зберігаються в окремих файлах. Наприклад, конфігурація для розігріву містить необхідні команди для підготовки сервера. Застосування такої конфігурації реалізовано наступним чином:

```
const warmupConfig = [
    'mp_warmuptime 3600',
    'mp_warmup_pausetimer 1',
    'mp_maxmoney 60000',
    'mp_startmoney 60000',
    'mp_free_armor 1',
    'mp_warmup_start',
];

async function applyWarmupConfig(rcon) {
    try {
        for (const command of warmupConfig) {
            await rcon.execute(command);
        }
    } catch (error) {
        console.error('Помилка при застосуванні конфігурації:',
error);
    }
}
```

Обробка логів матчів виконується через прийом HTTP-запитів, які надходять від серверу. Логи записуються у файл із зазначенням часу події. Код для обробки логів представлений нижче:

```
const fs = require('fs');
const path = require('path');

async function handleLog(matchId, logMessage) {
    const logPath = path.join(__dirname, 'logs',
`match_${matchId}.log`);
    fs.appendFileSync(logPath, `${new Date().toISOString()} -
${logMessage}\n`);
}
```

Для зберігання паролів серверів використовується шифрування на основі AES-256-CBC. Шифрування забезпечує захист даних як у стані зберігання, так і під час передачі. Приклад реалізації шифрування паролів наведено нижче:

```
const crypto = require('crypto');

const ENCRYPTION_KEY = process.env.ENCRYPTION_KEY; // 32-byte
key
const ENCRYPTION_IV = process.env.ENCRYPTION_IV; // 16-byte IV

function encryptPassword(password) {
    const cipher = crypto.createCipheriv('aes-256-cbc',
Buffer.from(ENCRYPTION_KEY), Buffer.from(ENCRYPTION_IV));
    let encrypted = cipher.update(password, 'utf8', 'hex');
    encrypted += cipher.final('hex');
    return encrypted;
}

function decryptPassword(encryptedPassword) {
    const decipher = crypto.createDecipheriv('aes-256-cbc',
Buffer.from(ENCRYPTION_KEY), Buffer.from(ENCRYPTION_IV));
    let decrypted = decipher.update(encryptedPassword, 'hex',
'utf8');
    decrypted += decipher.final('utf8');
    return decrypted;
}
```

3.10 Зчитування логів

Логи серверу розбираються за допомогою написаного класа-парсера `LogParser`, який обробляє події, витягує корисну інформацію та зберігає її у структурованому форматі. Наприклад, функція `parseKillLog` відповідає за розпізнавання подій вбивств. У логах серверу подія вбивства записується в уніфікованому форматі, і ця функція використовує регулярний вираз для розпізнавання необхідних даних:

```
static parseKillLog(log) {
    const killRegex =
/"([^"]+)<\d+><([^>]+|BOT)><(TERRORIST|CT)>" killed
"([^"]+)<\d+><([^>]+|BOT)><(TERRORIST|CT)>" with "([^"]+)"(?:
\((headshot)\))?/;
    const match = log.match(killRegex);
    if (match) {
        return {
            attackerName: match[1],  // Ім'я гравця, який
здійснив вбивство
```

```
            attackerSteamId: match[2], // Steam ID нападника
            attackerTeam: match[3], // Команда нападника
(TERRORIST/CT)
            victimName: match[4],  // Ім'я жертви
            victimSteamId: match[5], // Steam ID жертви
            victimTeam: match[6], // Команда жертви
(TERRORIST/CT)
            weapon: match[7], // Зброя, використана для вбивства
            headshot: match[8] ? true : false, // Чи був
зроблений постріл у голову
        };
    }
    return null; // Якщо рядок не відповідає формату події
вбивства
}
```

Результати парсингу зберігаються в базі даних для подальшого аналізу, створення статистики та відображення на сайті. Інформація про подію вбивства зберігається в таблиці `KillEvent`, яка має зв’язок 1:1 із загальною таблицею подій `Event`.

3.11 Відображення даних у вебінтерфейсі

Відображення даних у вебінтерфейсі було реалізовано за допомогою бібліотеки React для побудови користувацького інтерфейсу та Redux для управління станом додатку. Клієнтська частина забезпечує зручну та інтуїтивну взаємодію користувачів із системою, дозволяючи виконувати такі дії, як перегляд матчів, керування серверами, командами та турнірами.

Структура клієнтської частини побудована з використанням компонентів, що відповідають за окремі частини інтерфейсу:

- **App.js** — головний компонент, який містить маршрутизацію та загальну логіку додатку.
- **Компоненти**: `Header`, `Sidebar`, `MainContent` та інші, що відповідають за відображення загальних елементів інтерфейсу.
- **Сторінки**: `Home`, `ManageServers`, `ManageTeams` тощо, які відповідають за відображення конкретних розділів сайту.
- **Redux slices**: `authSlice`, `gameServerSlice`, `teamSlice`, `tournamentSlice`, які відповідають за управління станом додатку.

Приклад компонента `App.js`:

```
import React from 'react';
import { Routes, Route } from 'react-router-dom';
import Header from './components/Header';
import Sidebar from './components/Sidebar';
import Home from './pages/Home';

function App() {
  return (
    <div className="flex min-h-screen bg-secondary">
      <Sidebar />
      <div className="flex flex-col flex-1">
        <Header />
        <main className="flex-1 bg-secondary overflow-y-auto
text-white">
          <Routes>
            <Route path="/" element={<Home />} />
            {/* Інші маршрути */}
          </Routes>
        </main>
      </div>
    </div>
  );
}

export default App;
```

Управління станом додатку здійснюється за допомогою Redux. Наприклад, `authSlice.js` відповідає за автентифікацію користувачів:

```
import { createSlice, createAsyncThunk } from
'@reduxjs/toolkit';
import api from '../../api';

export const fetchCurrentUser =
createAsyncThunk('auth/fetchCurrentUser', async (_, {
rejectWithValue }) => {
  try {
    const response = await api.get('/me');
    return response.data.player;
  } catch (error) {
    return rejectWithValue(error.response.data.message);
  }
});

const authSlice = createSlice({
  name: 'auth',
  initialState: { user: null, status: 'idle', error: null },
  reducers: { /* ... */ },
  extraReducers: { /* ... */ },
});
```

```
export default authSlice.reducer;
```

Відображення даних здійснюється через компоненти, такі як Table для відображення таблиць з серверами, командами чи турнірами:

```
import Table from '../components/Table';

function ManageServers() {
  const servers = useSelector(state =>
state.gameServers.servers);
  const columns = [
    { header: 'ID', accessor: 's_id' },
    { header: "Ім'я сервера", accessor: 's_hostname' },
    // Інші колонки...
  ];

  return (
    <div className="p-6 bg-secondary min-h-screen">
      <h2 className="font-semibold mb-6 text-white">Управління
ігровими серверами</h2>
      <Table columns={columns} data={servers} />
    </div>
  );
}

export default ManageServers;
```

Дизайн інтерфейсу створений з використанням Tailwind CSS, що дозволяє швидко та ефективно стилізувати компоненти. Налаштування кольорів та стилів знаходяться у файлі `tailwind.config.js`:

```
module.exports = {
  theme: {
    extend: {
      colors: {
        primary: '#0c101a',
        secondary: '#11141c',
        accent: '#6080ff',
        // Інші кольори...
      },
    },
  },
  // Інші налаштування...
};
```

Взаємодія з сервером здійснюється через Axios. Клієнт API налаштований у файлі `api.js`:

```
import axios from 'axios';

const api = axios.create({
```

```
  baseURL: process.env.REACT_APP_API_URL ||
'http://localhost:5000/api',
  withCredentials: true,
});

export default api;
```

3.12 Аналіз результатів

Робота web-додатку

Після запуску додатку користувач потрапляє на головну сторінку, де відображаються актуальні матчі, лідери команд та гравців. Інтерфейс розроблений з акцентом на зручність та інформативність, що дозволяє користувачам швидко отримати необхідну інформацію. На головній сторінці відображається список останніх матчів, де кожна картка містить інформацію про команди-учасники, рахунок, статус та формат матчу (рис. 4.6).

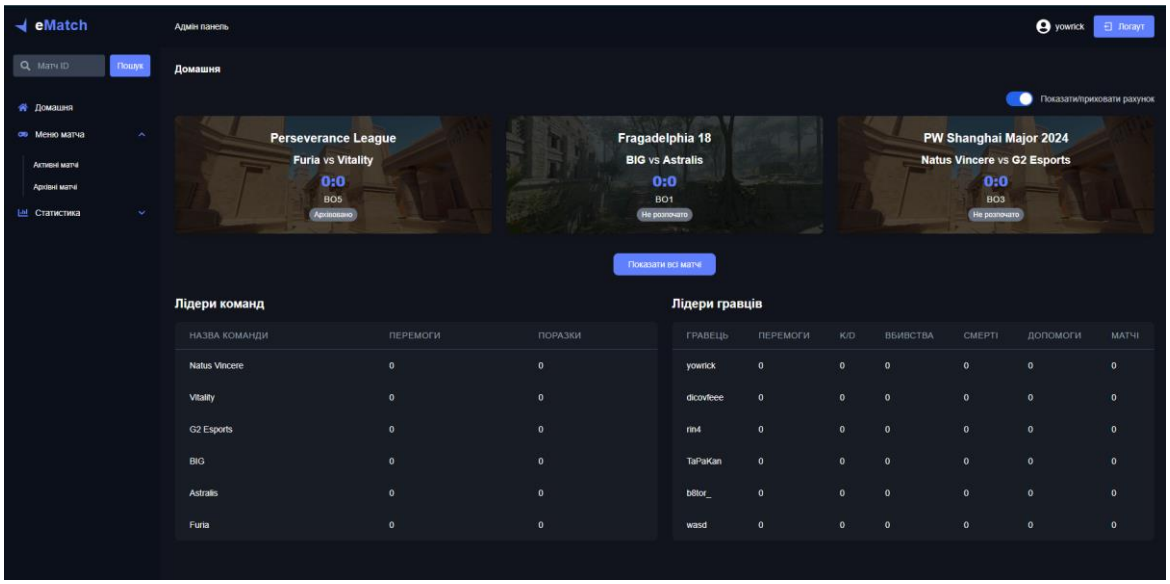


Рисунок 4.6 – Головна сторінка

Для тих, хто не бажає бачити результати заздалегідь, передбачено перемикач, який дозволяє приховувати або відображати рахунок матчів. Окрім цього, представлені таблиці з лідерами команд та гравців, що містять статистичні показники, такі як перемоги, поразки, K/D, вбивства та інші. Для швидкого переходу до повного списку активних матчів є кнопка "Показати всі матчі".

Сторінка управління серверами доступна лише для адміністраторів і призначена для керування ігровими серверами (рис. 4.7).

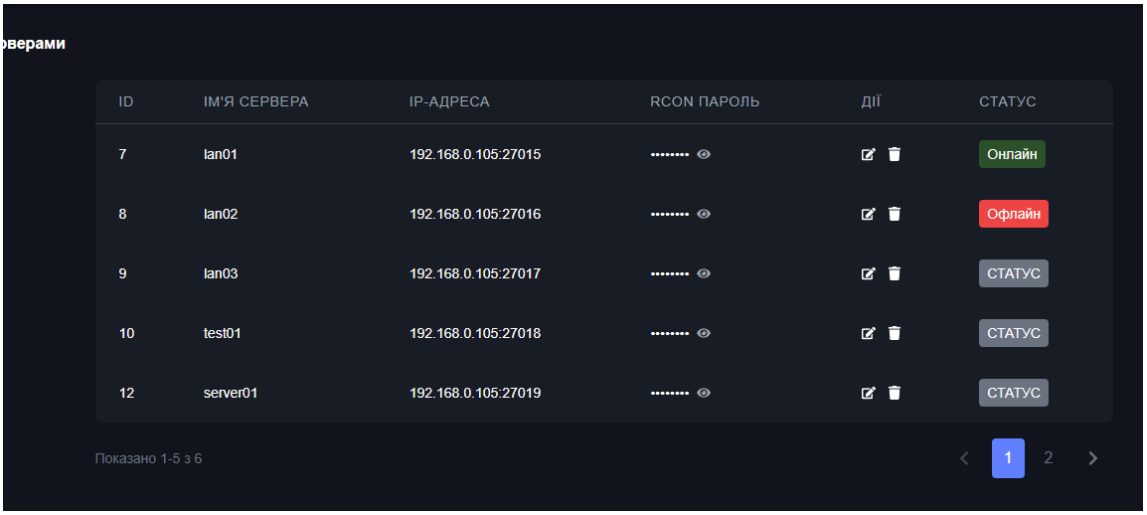


Рисунок 4.7 – Сторінка управління серверами

Інтерфейс відображає таблицю з усіма доданими серверами, де зазначені їхні ID, назва, IP-адреса та поточний статус. Адміністратори мають можливість перевіряти статус серверів, щоб дізнатися, чи вони онлайн чи офлайн. Для кожного сервера передбачені дії редагування та видалення. При натисканні на кнопку редагування відкривається модальне вікно, де можна змінити назву, IP-адресу та RCON пароль сервера (рисунок 4.8).

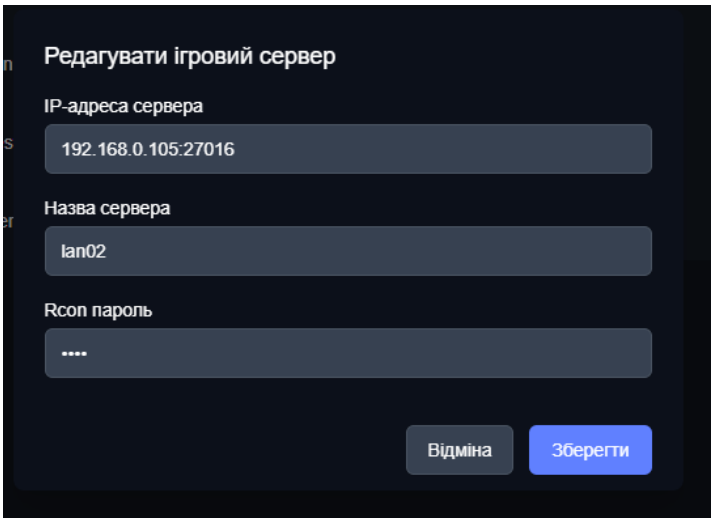


Рисунок 4.8 – Модальне вікно редагування сервера

Видалення сервера супроводжується модальним вікном, яке просить підтвердити дію та попереджає про її незворотність.

Управління командами є ще однією важливою функцією для адміністраторів. Сторінка дозволяє створювати нові команди (функціонал бокового меню), а також редагувати та видаляти існуючі (рис. 4.9).

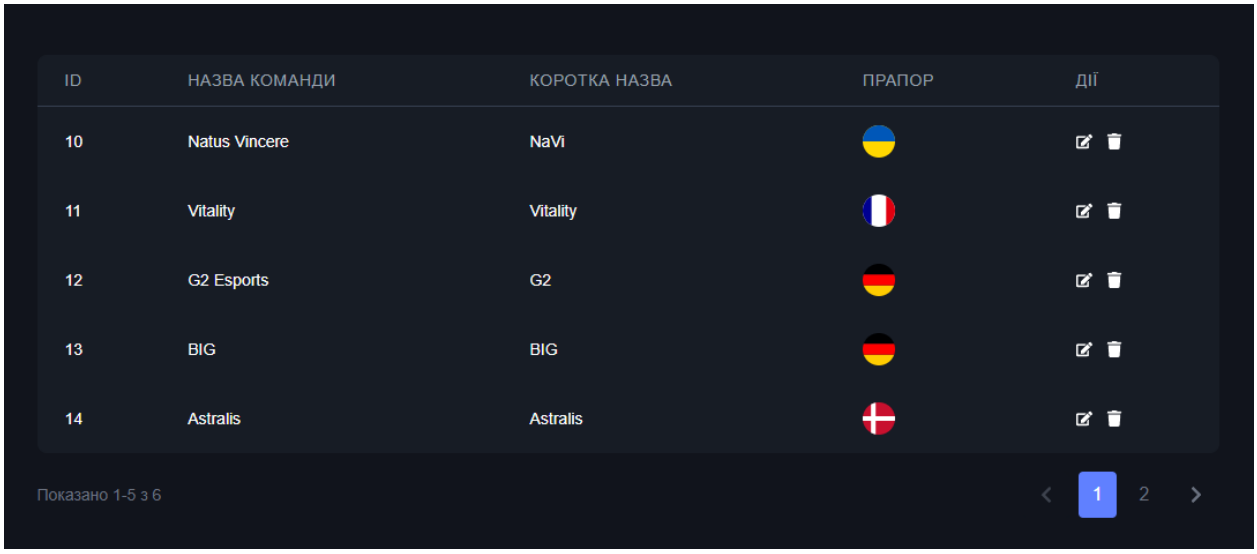


Рисунок 4.9 – Сторінка управління командами турнірів

Інтерфейс представляє таблицю з інформацією про команди, включаючи їх ID, назву, коротку назву та прапор країни. Для редагування існуючої використовується модальне вікно, де адміністратор може ввести або змінити назву команди, коротку назву та код країни для відображення відповідного прапора. Видалення команди підтверджується через модальне вікно, яке попереджає про можливі наслідки (рис. 4.10).

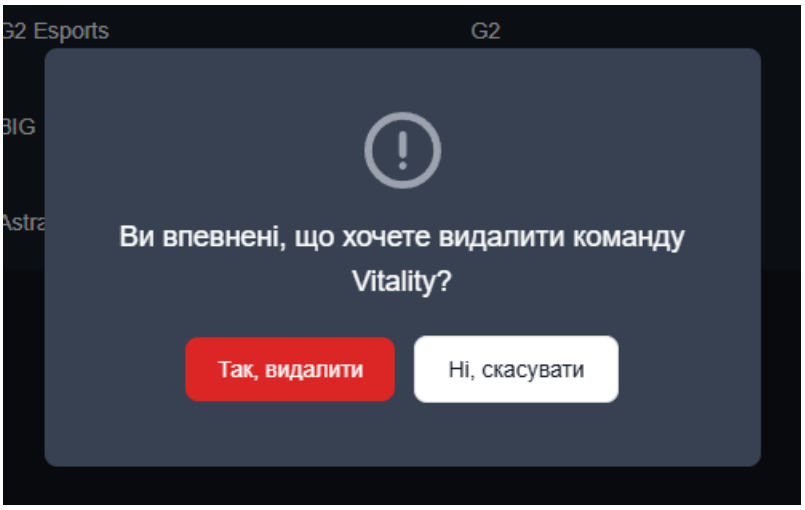


Рисунок 4.10 – Модальне вікно видалення команди

Сторінка управління турнірами надає адміністраторам можливість керувати турнірами, створювати нові та редагувати або видаляти існуючі (рис. 4.11).

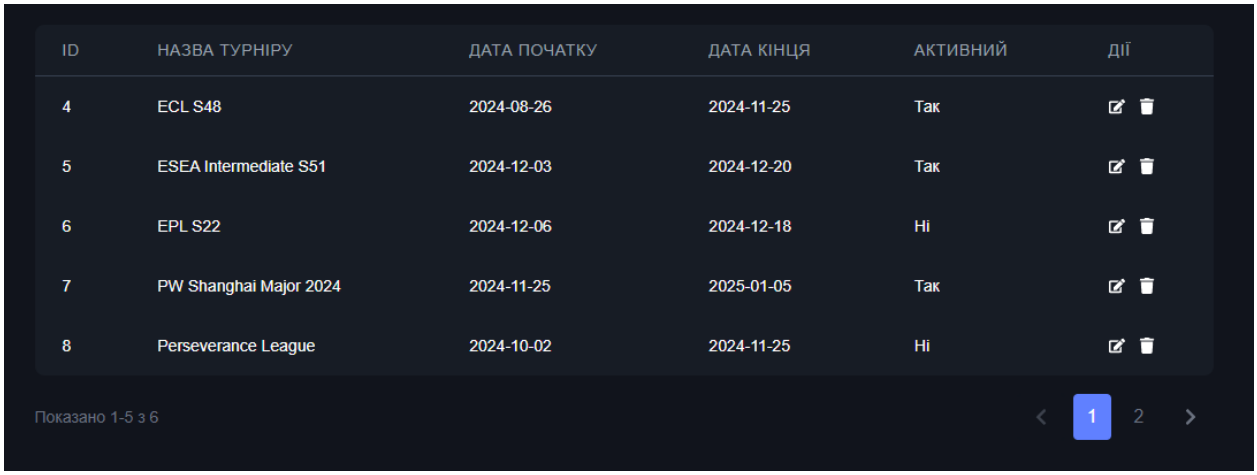


Рисунок 4.11 – Сторінка управління турнірами

Інтерфейс відображає таблицю з турнірами, де зазначені їх ID, назва, дати початку та кінця, а також статус активності. Редагування турніру здійснюється через модальне вікно, де можна змінити назву, дати проведення та статус активності (рис. 4.12).

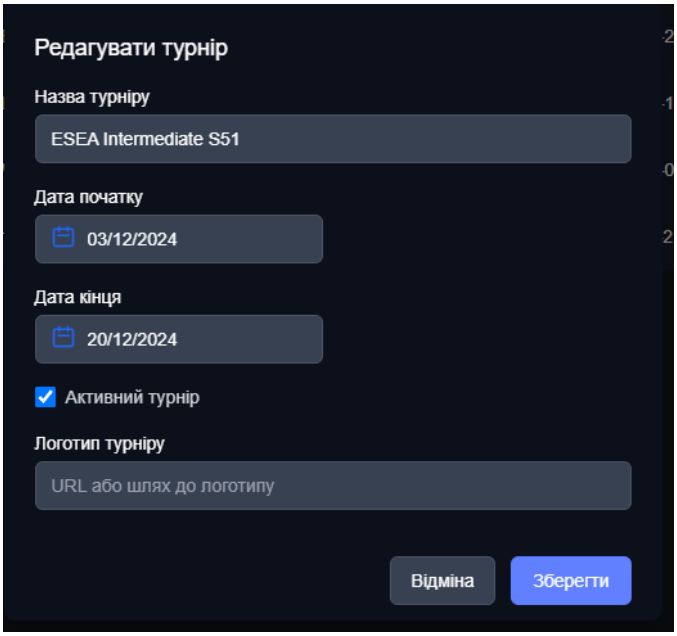


Рисунок 4.12 – Модальне вікно редагування турніру

Видалення турніру супроводжується модальним вікном для підтвердження дії.

Створення матчу є однією з ключових функцій для адміністраторів, яка дозволяє детально налаштувати майбутній матч (рис. 4.13).

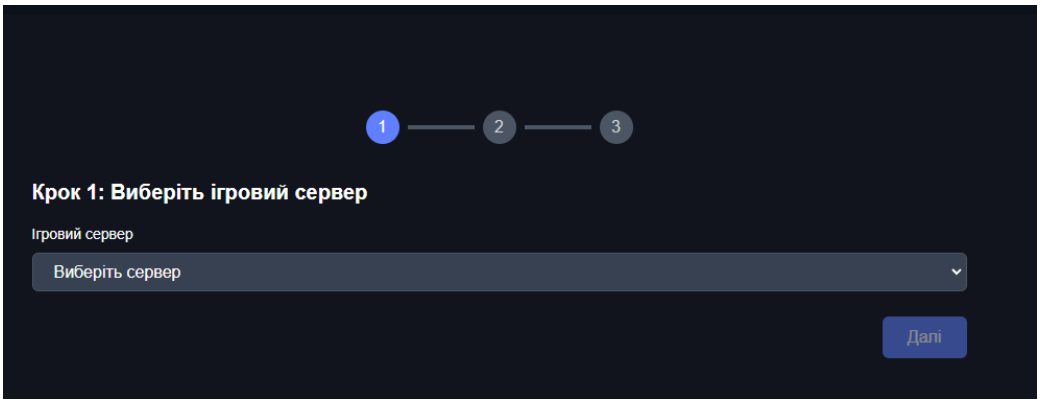


Рисунок 4.13 – Перший етап при створенні матчу

Процес створення матчу складається з кількох кроків, що послідовно ведуть адміністратора через вибір ігрового сервера, турніру та налаштування самого матчу. Спочатку обирається сервер, на якому буде проведено матч, з переліку доступних серверів. Далі вибирається турнір, до якого належатиме матч. На останньому кроці відбувається налаштування матчу: вибір команд А та Б, встановлення формату матчу (BO1, BO3, BO5), вибір карт з мап пулу та призначення стартових сторін для кожної карти (рис. 14).

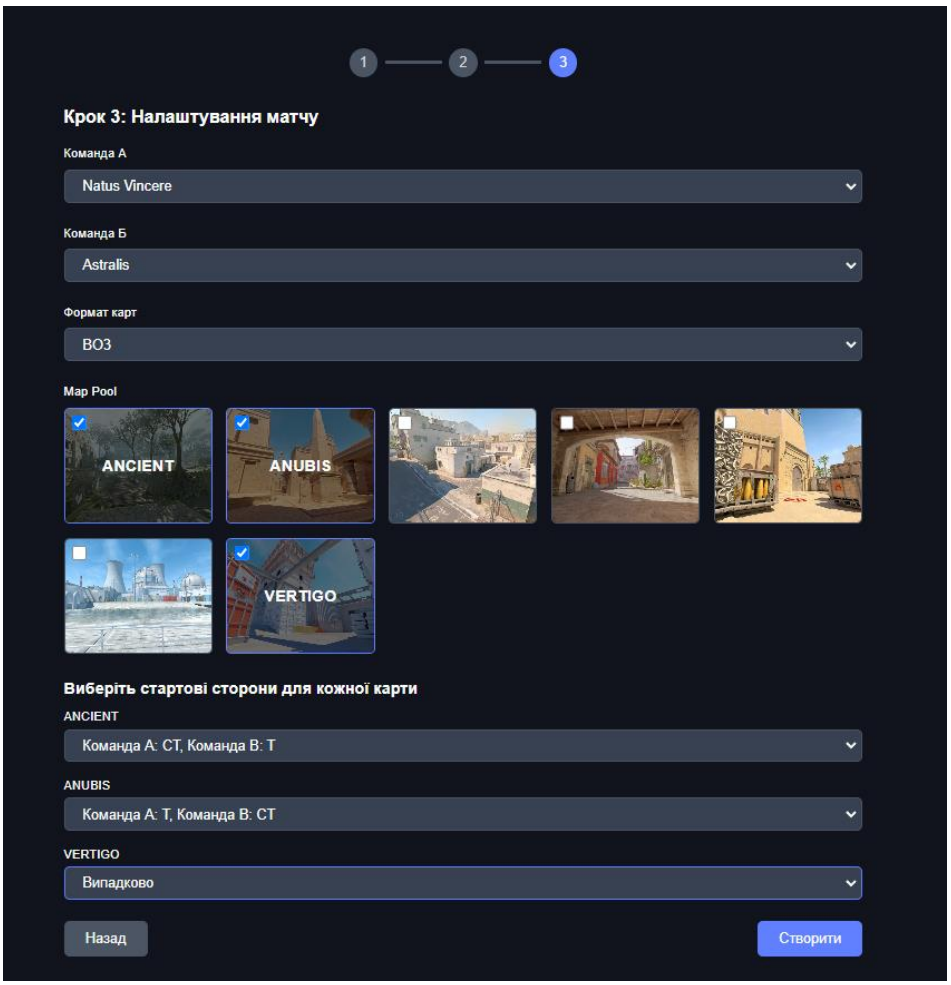


Рисунок 4.14 – Фінальний етап налаштування матчу

Система автоматично перевіряє коректність введених даних та відповідність кількості обраних карт обраному формату.

Сторінка активних матчів відображає список поточних матчів з можливістю взаємодії (рис. 4.15).

Рисунок 4.15 – Сторінка активних матчів

Інтерфейс представляє таблицю з детальною інформацією про матчі, включаючи команди-учасники, рахунок, карти, формат, турнір та статус матчу. При натисканні на кнопку “Показати” користувач може отримати детальну інформацію про матч. Адміністратори мають можливість керувати матчами в

реальному часі: почати матч, поставити його на паузу, завершити або архівувати в залежності від статусу матчу. (рис. 4.16).

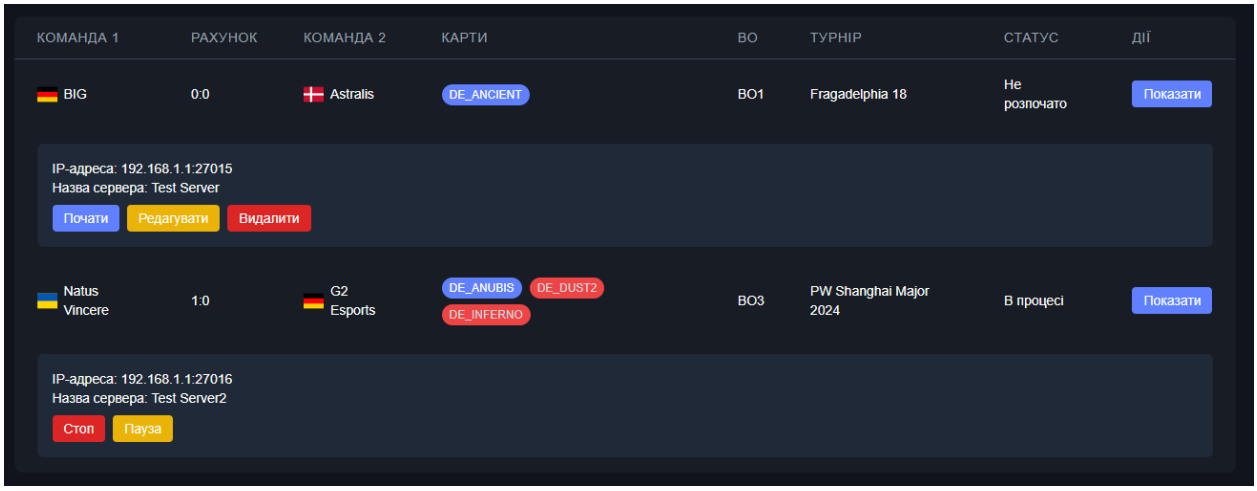


Рисунок 4.16 – Функціонал управління матчами

Сторінка архівних матчів надає можливість переглянути завершені та архівовані матчі. Інтерфейс схожий на сторінку активних матчів, але зосереджений на минулих подіях.

Перегляд матчу є однією з найбільш інформативних та інтерактивних сторінок додатку. Вона надає детальний аналіз матчу з різних аспектів, розбитий на вкладки для зручності користувача.

Перша вкладка "Інформація про матч / конфігурація" містить загальний огляд матчу, його налаштування та поточний стан. Тут можна бачити формат матчу, команди-учасники, турнір, статус та список карт (рис .4.17).

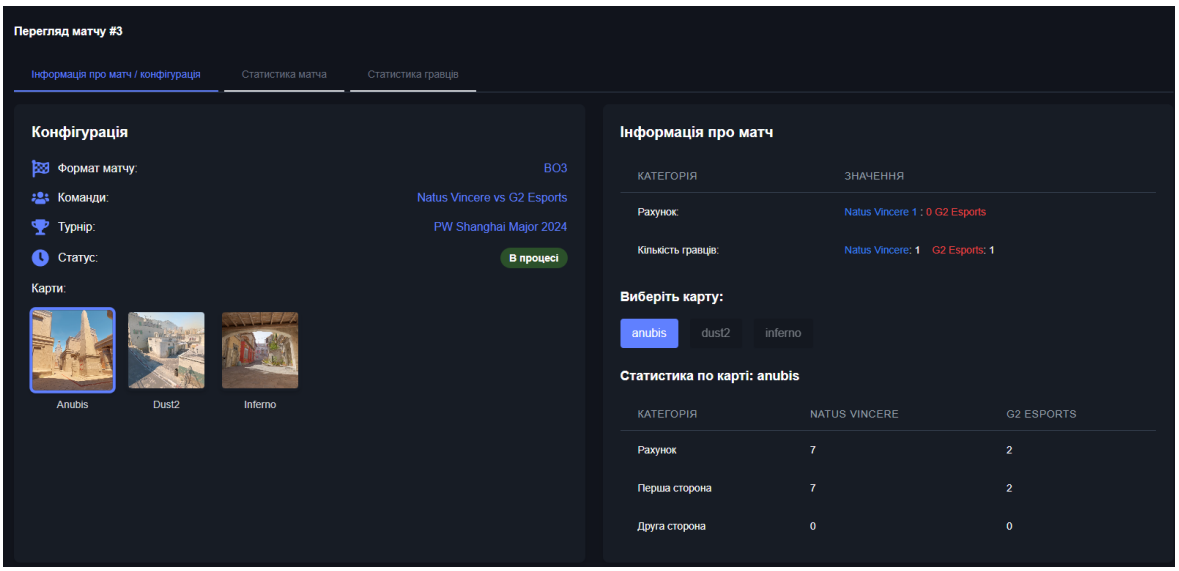


Рисунок 4.17 – Вкладка перегляду інформації про матч та його кофігарції

Кожна карта представлена зображенням та може бути обрана для детального перегляду.

Друга вкладка "Статистика матчу" забезпечує аналіз перебігу матчу на обраній карті (рис. 4.18). Користувач може переглянути прогрес-лінію раундів, яка візуально відображає серії перемог команд та динаміку матчу. Аналіз раундів представлений у вигляді таблиці, де деталізовано типи перемог, такі як бомба вибухнула, бомба знешкоджена, повна ліквідація тощо. Інтерактивний слайдер раундів дозволяє переглянути події кожного раунду, включаючи вбивства, час та використану зброю.

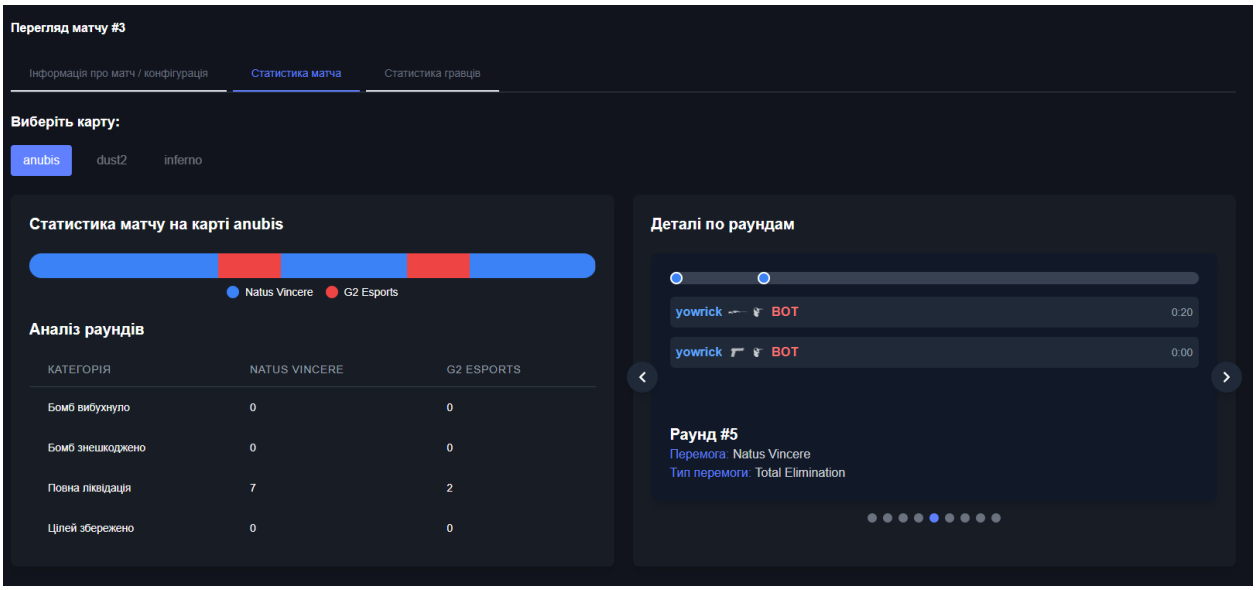


Рисунок 4.18 – Вкладка статистики матчу

При наведенні курсора на події або елементи прогрес-лінії відображаються тултіп-підказки з додатковою інформацією, що робить аналіз більш інтуїтивним.

Третя вкладка "Статистика гравців" фокусується на індивідуальній продуктивності гравців у матчі (рис. 4.19).

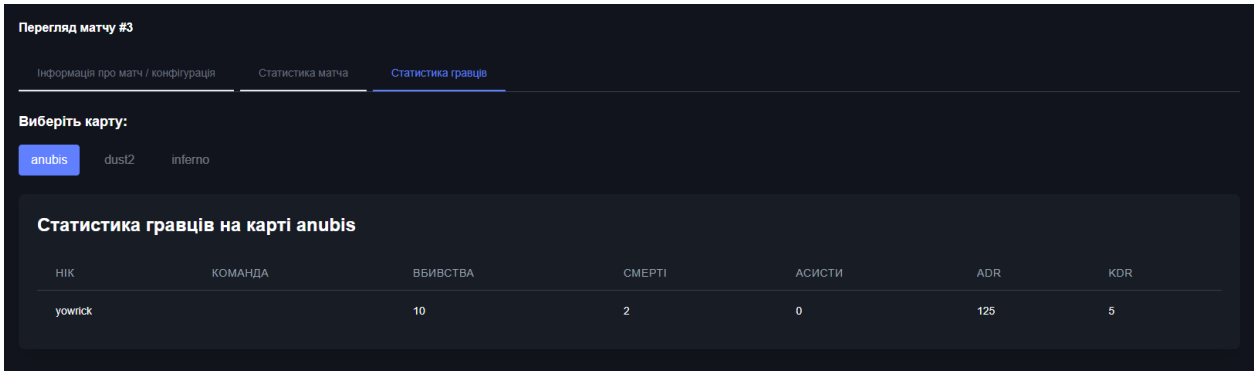


Рисунок 4.19 – Вкладка статистики гравців матчу

Тут представлена таблиця з показниками кожного гравця, такими як кількість вбивств, смертей, асистів, середній дамаг за раунд (ADR), коефіцієнт вбивств/смертей (KDR) та інші.

Ораганізація матчу

У системі ретельно продумано та реалізовано процес проведення кіберспортивних матчів.

Після запуску матчу, гравці підключаються до ігрового сервера через клієнт гри та проходять фазу розминки. Під час цієї фази вони можуть перевірити свої налаштування, розім'ятися та підготуватися до гри. Система очікує, поки всі гравці підтвердять свою готовність за допомогою спеціальних команд у чаті, таких як !ready або !rdy. Це забезпечує, що всі учасники готові розпочати матч одночасно (рис. 4.20).

Рисунок 4.20 – Фаза очікування гравців

Після того, як всі гравці підтвердили готовність, відбувається ножовий раунд. Цей раунд визначає, яка з команд отримає право вибору сторони (терористи або контр-терористи) для початку основної гри. Команди змагаються лише з використанням ножів, і переможець цього раунду отримує перевагу у виборі сторони. Після завершення ножового раунду система автоматично запитує у переможців їх вибір сторони через чат. Гравці можуть обрати сторону, надсилаючи команди “!ct” або “!t” (рис. 4.21).

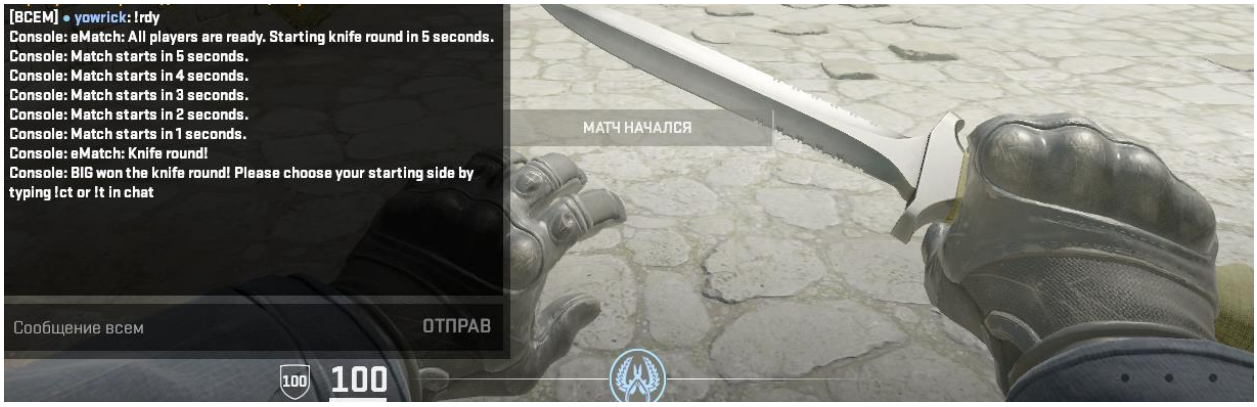


Рисунок 4.21 – Фаза очікування вибору сторони

Коли сторони визначено, система налаштовує сервер відповідно до вибору та розпочинає основний матч. Під час гри система в режимі реального часу відслідковує та записує всі події, такі як вбивства, перемоги в раундах та інші ключові моменти (рис. 4.22).

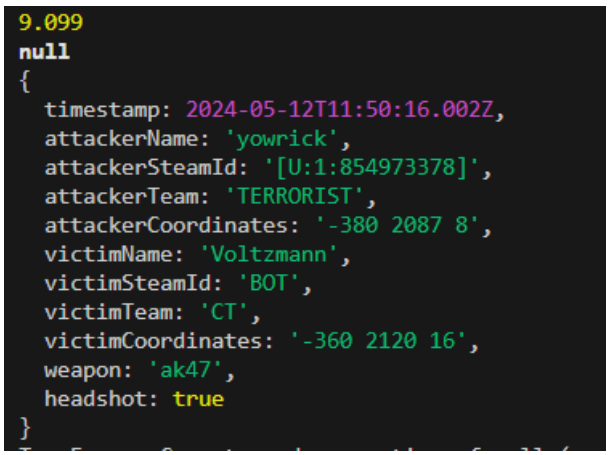


Рисунок 4.22 – Відслідкування сервером події вбивства під час матчу

Після завершення всіх раундів та карт, відповідно до формату матчу, система автоматично визначає переможця. Результати матчу фіксуються, статус оновлюється на "Завершено".

ВИСНОВКИ

Після завершення розробки та тестування інформаційної технології для керування та аналізу кіберспортивних турнірів з дисципліни CS2, було узагальнено та оцінено результати виконаної роботи. Підсумки підтвердили актуальність проєкту, ефективність застосованих підходів та досягнення поставлених цілей. Основні висновки такі:

1) На основі проведеного дослідження популярності CS2 та аналізу тенденцій розвитку кіберспортивної індустрії підтверджено актуальність створення інформаційної технології для автоматизації турнірної діяльності.
2) Проведений аналіз існуючих систем керування та аналізу кіберспортивних турнірів виявив низку обмежень, що дозволило визначити вимоги до розроблюваного рішення.
3) У межах кваліфікаційної роботи створено інформаційну технологію для управління та аналізу турнірів з CS2, зокрема:
   - побудовано клієнт-серверну архітектуру із застосуванням Node.js, Express.js, React.js, MySQL, що забезпечило продуктивність і масштабованість;
   - побудована діаграму варіантів використання та розроблено концептуальну та логічну моделі бази даних, які гарантують стабільність та надійність системи;
   - реалізовано серверну частину з інтеграцією до ігрових серверів через RCON, що дозволило автоматизувати запуск матчів та збір ігрових даних;
   - створено клієнтський вебінтерфейс на базі React.js, забезпечивши зручний доступ для адміністраторів і користувачів;
   - впроваджено авторизацію через Steam, що підвищило рівень безпеки та спростило процес автентифікації.

4) Проведена оцінка функціональності системи засвідчила відповідність розробленого рішення визначеним вимогам.

Аналіз результатів показав, що розроблена інформаційна технологія ефективно автоматизує процеси управління кіберспортивними турнірами та надає можливості для аналізу ігрових даних. Система дозволяє створювати та керувати турнірами, командами, матчами та гравцями, інтегрується з ігровими серверами для автоматичного проведення матчів та збирає детальну статистику для подальшого аналізу.

Запропонована інформаційна технологія підвищує ефективність організації кіберспортивних турнірів, покращує досвід гравців і глядачів та сприяє розвитку кіберспортивної індустрії в цілому. Вона автоматизує рутинні процеси, зменшує кількість помилок, пов'язаних із людським фактором, та забезпечує високий рівень надійності й безпеки даних. Це, у свою чергу, дозволяє організаторам зосередитися на стратегічних аспектах проведення турнірів, покращуючи загальну якість кібер-заходів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Werder K. Esport // Business and Information Systems Engineering. Springer Gabler, 2022. Т. 64, № 3. С. 393–399.
2. Snavely T.L. History and analysis of eSport systems. 2014.
3. Newzoo Global Esports & Live Streaming Market Report | Free Version [Електронний ресурс]. URL: https://newzoo.com/resources/trend-reports/newzoo-global-esports-live-streaming-market-report-2022-free-version (Дата звернення: 04.11.2024).
4. Я. Кисленко, О. Шовкопляс. Інформаційна технологія керування та аналізу даних для кіберспортивних турнірів // матеріали та програма міжнародної наукової конференції молодих учених, м. Суми-Астана, 22-26 квітня 2024 року. Суми, 2024. С. 81.
5. Jenny S.E. et al. eSports Venues: A New Sport Business Opportunity // Journal of Applied Sport Management. University of Tennessee, 2018. Т. 10, № 1. С. 34–49.
6. Кіберспортивні трансляції українською мовою у 2023 році набрали 15,5 млн годин перегляду. Це вдвічі більше ніж торік | dev.ua [Електронний ресурс]. URL: https://dev.ua/news/kibersportyvni-transliatsii-ukrainskoiu-movoiu-1703498328 (Дата звернення: 04.11.2024).
7. Jenny S.E. et al. Virtual(ly) Athletes: Where eSports Fit Within the Definition of “Sport” // Quest. Routledge, 2017. Т. 69, № 1. С. 1–18.
8. Кіберспорт в Україні – офіційний вид спорту з 7 вересня 2020 [Електронний ресурс]. URL: https://zaxid.net/kibersport_v_ukrayini_ofitsiyniy_vid_sportu_z_7_veresnya_2020_n1507356 (Дата звернення: 04.11.2024).
9. Counter-Strike 2 - Wikipedia [Електронний ресурс]. URL: https://en.wikipedia.org/wiki/Counter-Strike_2 (Дата звернення: 03.12.2024).
10. Exploring E-sports: A Case Study of Gameplay in Counter-strike [Електронний ресурс]. URL:

https://www.researchgate.net/publication/237523631_Exploring_E-sports_A_Case_Study_of_Gameplay_in_Counter-strike (Дата звернення: 03.12.2024).

11. Gasparetto T., Safronov A. Streaming demand for eSports: Analysis of Counter-strike: Global offensive // Convergence. SAGE Publications Ltd, 2023. Т. 29, № 5. С. 1369–1388.
12. Puranam P., Håkonsson D.D. Valve's Way // Journal of Organization Design. 2015. Т. 4, № 2.
13. Valve Corporation - Wikipedia [Електронний ресурс]. URL: https://en.wikipedia.org/wiki/Valve_Corporation (Дата звернення: 03.12.2024).
14. Half-Life - #1 Top Shooters - IGN [Електронний ресурс]. URL: https://www.ign.com/lists/shooters/1 (Дата звернення: 03.12.2024).
15. Node.js Challenges in Implementation [Електронний ресурс]. URL: https://www.researchgate.net/publication/318310544_Nodejs_Challenges_in_Implementation (Дата звернення: 04.12.2024).
16. Express - це фреймворк для веб-застосунків, побудованих на Node.js [Електронний ресурс]. URL: https://expressjs.com/uk/?utm_source=chatgpt.com (Дата звернення: 04.12.2024).
17. Archana B. Present day web-development using reactjs // International Research Journal of Engineering and Technology (IRJET) e-I. 2020.
18. Seyed M. M. Learning MySQL: Get a Handle on Your Data. 2007. 4 с.
19. "Node.js Database Magic: Exploring the Powers of Sequelize ORM" [Електронний ресурс]. URL: https://medium.com/@rishu__2701/node-js-database-magic-exploring-the-powers-of-sequelize-orm-a22a521d9d9d (Дата звернення: 04.12.2024).
20. Source RCON Protocol - Valve Developer Community [Електронний ресурс]. URL:

https://developer.valvesoftware.com/wiki/Source_RCON_Protocol (Дата звернення: 04.12.2024).

21. Steam Community :: Steam Web API Documentation [Електронний ресурс]. URL: https://steamcommunity.com/dev? (Дата звернення: 04.12.2024).
22. Arwa Y. Aleryani. Comparative Study between Data Flow Diagram and Use Case Diagram // International Journal of Scientific and Research Publications. 2016.
23. Svyatoslav Kulikov. Relational Databases By Examples. 2024. C. 8

ДОДАТОК А
ПЛАНУВАННЯ РОБІТ

А1 Деталізація мети проекту методом SMART.

Продуктом дипломного проєкту є інформаційна технологія керування та аналізу кіберспортивних турнірів з дисципліни CS2. Результати деталізації мети за методом SMART наведені в табл. А.1.

Таблиця А.1 – Деталізація мети методом SMART

Specific (конкретна)	Розробити інформаційну технологію, що автоматизує процеси управління та аналізу кіберспортивних турнірів з дисципліни CS2, включаючи створення та управління турнірами, командами, гравцями, автоматизацію матчів та аналіз ігрових даних.
Measurable (вимірювана)	Результатом роботи є функціональна інформаційна система, яка забезпечує: 1) автоматизацію процесів управління кіберспортивними турнірами; 2) інтеграцію з ігровими серверами CS2; 3) збір та аналіз статистичних даних; 4) зручний вебінтерфейс для адміністраторів та користувачів.
Achievable (досяжна)	Для виконання проєкту наявні необхідні знання в галузі веброзробки (Node.js, Express.js, React.js), баз даних (MySQL), протоколів інтеграції з ігровими серверами (RCON), а також навички написання технічної документації. Враховуючи доступні ресурси та обмеження, мета є досяжною.
Relevant (релевантна)	Ця технологія дозволить організаторам кіберспортивних турнірів ефективніше керувати змаганнями, автоматизувати рутинні процеси та покращити аналіз ігрових даних, що сприятиме розвитку кіберспортивної індустрії та підвищенню якості турнірів.
Time-framed (обмежена у часі)	Розробити та впровадити інформаційну технологію керування та аналізу кіберспортивних турнірів з дисципліни CS2 відповідно до календарного плану проєкту, завершивши роботу до 4 грудня 2024 року.

А2 Планування змісту структури робіт IT-проєкту (WBS).

Структура декомпозиції робіт (Work Breakdown Structure, WBS) є ключовим інструментом управління проєктами, що дозволяє систематично розбити проєкт на керовані компоненти. WBS забезпечує ієрархічну структуру, яка допомагає планувати та контролювати виконання завдань, розподіляти ресурси та відстежувати прогрес.

Добре спроєктована WBS виконує такі функції:

- Надає чіткий план виконання проєкту, розбиваючи його на фази, результати та робочі пакети.
- Полегшує оцінку вартості та бюджетування проєкту.
- Сприяє розподілу завдань та відповідальності між учасниками проєкту.
- Слугує основою для складання технічних завдань, специфікацій та звітів про хід виконання.

WBS діаграма проєкту зображена на рисунку А.1.

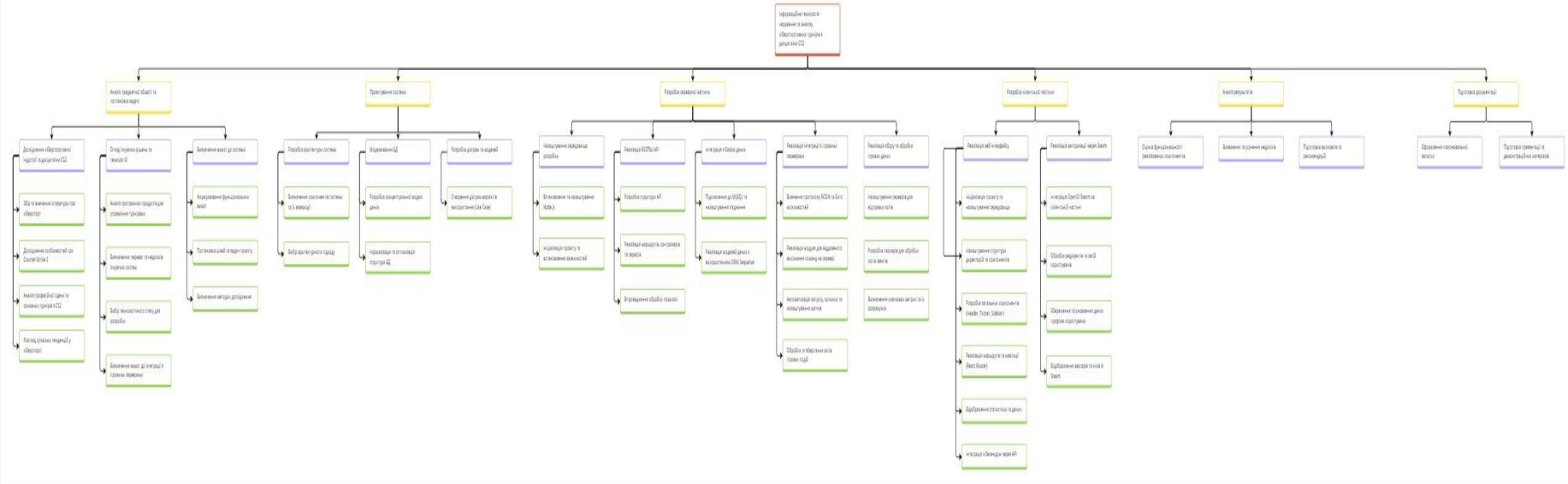


Рисунок А.1 – WBS діаграма

А3 Планування структури організації.

Виконавці проєкту:

- Шовкопляс Оксана Анатоліївна (менеджер проєкту)
- Кисленко Ярослав Володимирович (виконавець проєкту)

На основі цієї структури складено таблицю А.2.

Таблиця А.2 – Виконавці проєкту

Роль	Ім’я	Проєктна роль
Розробник	Кисленко Ярослав Володимирович	Розробляє продукт проєкту, здійснює тестування функціональних вимог та забезпечує якість програмного забезпечення.
Менеджер проєкту (керівник дипломної роботи)	Шовкопляс Оксана Анатоліївна	Відповідає за дотримання термінів, надає консультації та підтримку розробнику в питаннях виконання проєкту.

А4 Організаційна структура проєкту (OBS)

Організаційна структура проєкту (Organizational Breakdown Structure, OBS) є інструментом управління проєктом, який ілюструє ієрархічну структуру організації, що бере участь у проєкті. OBS допомагає визначити відповідальність за виконання завдань та сприяє ефективній комунікації між учасниками.

OBS проєкту представлена схематично на рисунку А.2.

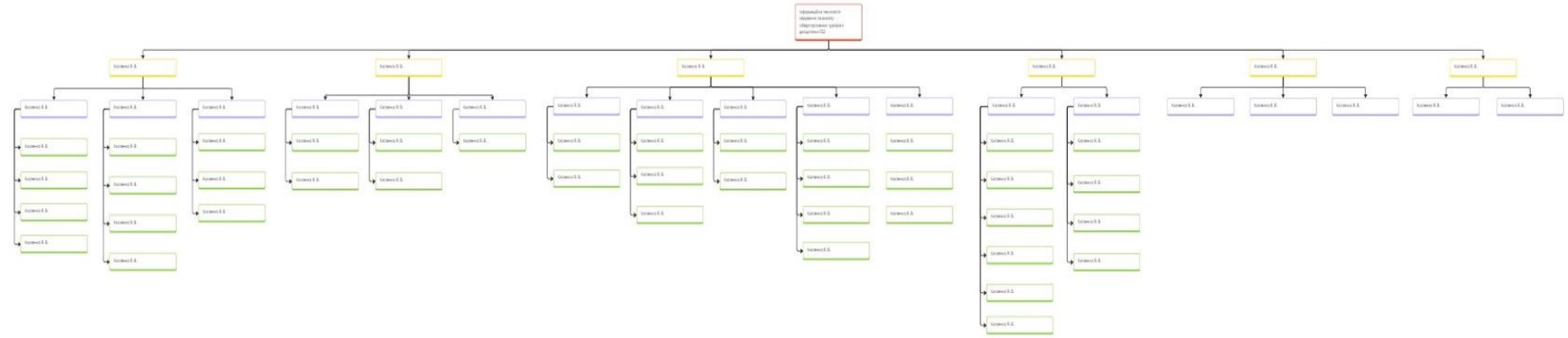


Рисунок А.2 – OBS діаграма

А5 Побудова календарного графіка виконання інформаційної технології

Для точного планування тривалості виконання робіт, беручи до уваги наявні ресурси та календарні обмеження, створено графік у вигляді діаграми Ганта. Цей графік слугує реальним розподілом завдань по календарним датам, виконуючи функцію розкладу робіт. Діаграма Ганта є оптимальним рішенням для візуалізації процесу виконання проєкту, оскільки вона дозволяє наочно побачити всі етапи роботи, їхні часові рамки та взаємозалежності.

Діаграма Ганта представлена горизонтальними смугами, розташованими на часовій шкалі, яка йде горизонтально по нижній частині графіка. Кожна смуга відповідає конкретному завданню або задачі, розміщеній вертикально по осі завдань. Початок та кінець смуги відображають дату початку та завершення відповідного завдання, а її довжина відображає тривалість виконання. На рисунку А.3. представлена діаграма Ганта стосовно роботи.

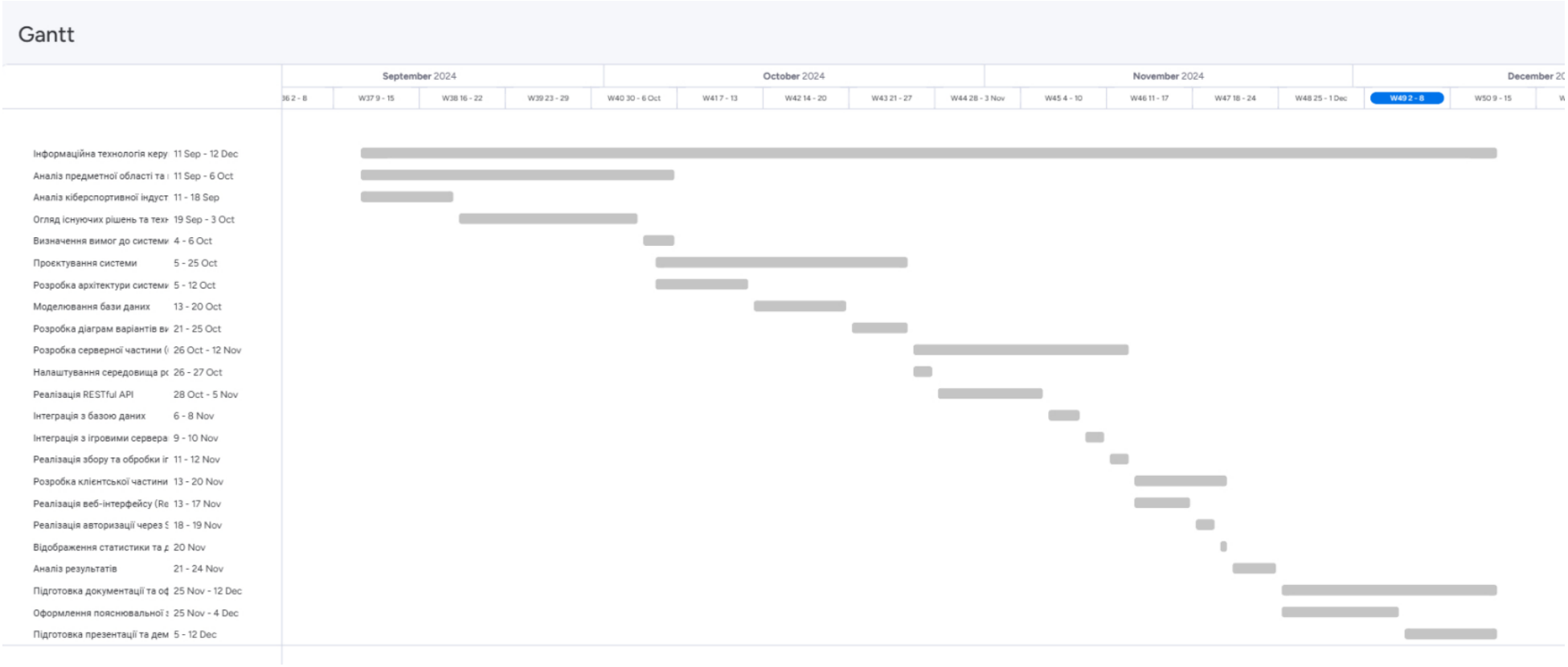


Рисунок А.3 – Діаграма Ганта

ДОДАТОК Б
ЛІСТИНГ ПРОГРАМНОГО КОДУ

Б1 index.js

```
require('dotenv').config();
const express = require('express');
const cors = require('cors');
const cookieParser = require('cookie-parser');
const session = require('express-session');
const { sequelize, connectDB } = require('./db/sequelize');
const router = require('./router');
const errorMiddleware =
require('./middlewares/errorMiddleware');
const passport = require('./middlewares/passport');
const PORT = process.env.PORT || 5000;
const app = express();

app.use(cors({
    origin: process.env.CLIENT_URL,
    credentials: true,
}));

app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(express.text());

app.use(cookieParser());

app.use(session({
    secret: process.env.SESSION_SECRET || 'your_session_secret',
    resave: false,
    saveUninitialized: false,
    cookie: {
        httpOnly: true,
        secure: false,
        maxAge: 24 * 60 * 60 * 1000,
    },
}));

app.use(passport.initialize());
app.use(passport.session());

app.use('/api', router);
app.use(errorMiddleware);

const start = async () => {
    try {
        await connectDB();
        // await sequelize.sync({ force: true });
        await sequelize.sync();
        app.listen(PORT, () => console.log(`Server started on
PORT: ${PORT}`));
```

```
    } catch (e) {
        console.log(e);
    }
}

start();
```

Б2 db/sequelize.js

```
require('dotenv').config();
const { Sequelize } = require('sequelize');

const sequelize = new Sequelize(
    process.env.DB_NAME,
    process.env.DB_USER,
    process.env.DB_PASSWORD,
    {
        host: process.env.DB_HOST,
        port: process.env.DB_PORT,
        dialect: 'mysql',
        logging: false,
    }
);

const connectDB = async () => {
    try {
        await sequelize.authenticate();
        console.log('Connection to the database successfully
established');
    } catch (e) {
        console.log('Failed to connect to the database:', e);
    }
};

module.exports = { sequelize, connectDB };
```

Б3 middlewares/authMiddleware.js

```
const ApiError = require('../exceptions/apiError');

module.exports = function(req, res, next) {
    if (req.isAuthenticated()) {
        return next();
    }
    return next(ApiError.Unauthorized());
};
```

Б4 middlewares/passport.js

```
const passport = require('passport');
const SteamStrategy = require('passport-steam').Strategy;
const playerService = require('../service/playerService');

function steamId64ToSteamId3(steamId64) {
```

```
    const steamId = BigInt(steamId64) -
BigInt("76561197960265728");
    return `[U:1:${steamId}]`;
}

passport.use(
    new SteamStrategy({
            returnURL:
`${process.env.HOSTNAME}:${process.env.PORT}/api/auth/steam/retu
rn`,
            realm:
`${process.env.HOSTNAME}:${process.env.PORT}`,
            apiKey: process.env.STEAM_KEY,
        },
        async (identifier, profile, done) => {
            try {
                const steamId = steamId64ToSteamId3(profile.id);
                const playerData = {
                    p_steam_id: steamId,
                    p_name: profile.displayName,
                };

                const player = await
playerService.findOrCreatePlayer(playerData);
                return done(null, player);
            } catch (err) {
                return done(err);
            }
        }
    )
);

passport.serializeUser((user, done) => {
    done(null, user.p_id);
});

passport.deserializeUser(async (id, done) => {
    try {
        const player = await playerService.getCurrentPlayer(id);
        done(null, player);
    } catch (err) {
        done(err);
    }
});

module.exports = passport;
```

Б5 models/MatchMap.js

```
const { DataTypes } = require('sequelize');
const { sequelize } = require('../db/sequelize');
const MapStatus = require('../constants/mapStatus');
```

```
const MatchMap = sequelize.define('MatchMap', {
    mm_match_map_id: {
        type: DataTypes.BIGINT,
        primaryKey: true,
        autoIncrement: true,
    },
    mm_match: {
        type: DataTypes.BIGINT,
        allowNull: false,
    },
    mm_map: {
        type: DataTypes.BIGINT,
        allowNull: false,
    },
    mm_map_order: {
        type: DataTypes.TINYINT,
        allowNull: false,
    },
    mm_score_team_a: {
        type: DataTypes.SMALLINT,
        allowNull: true,
    },
    mm_score_team_b: {
        type: DataTypes.SMALLINT,
        allowNull: true,
    },
    mm_team_a_start_side: {
        type: DataTypes.ENUM('CT', 'T'),
        allowNull: true,
    },
    mm_team_b_start_side: {
        type: DataTypes.ENUM('CT', 'T'),
        allowNull: true,
    },
    mm_status: {
        type: DataTypes.ENUM(
            MapStatus.NOT_STARTED,
            MapStatus.STARTING,
            MapStatus.WARMUP,
            MapStatus.KNIFE_ROUND,
            MapStatus.WAITING_FOR_SIDE_CHOICE,
            MapStatus.FIRST_HALF,
            MapStatus.SECOND_HALF,
            MapStatus.FINISHED,
        ),
        allowNull: true,
        defaultValue: MapStatus.NOT_STARTED,
    },
    mm_winner_team: {
        type: DataTypes.BIGINT,
        allowNull: true,
    },
}, {
```

```
    tableName: 'm2m_match_map',
    timestamps: false,
});

module.exports = MatchMap;
```

Б6 services/rconService.js

```
const Rcon = require('rcon-srcds').default;
const { decryptPassword } = require('../utils/encryption');
const ApiError = require('../exceptions/apiError');

class RconService {
    /**
     * @param {Object} server
     * @returns {Promise<Rcon>}
     */
    async createRconClient(server) {
        if (!server) {
            throw ApiError.BadRequest('Game server not found');
        }

        const decryptedRconPassword =
decryptPassword(server.s_rcon_password);
        const [host, portString] =
server.s_ip_address.split(':');
        const port = portString ? parseInt(portString, 10) :
27015;

        const rcon = new Rcon({
            host: host,
            port: port,
            encoding: 'ascii',
            maximumPacketSize: 0,
            timeout: 50000,
        });

        try {
            await rcon.authenticate(decryptedRconPassword);
            return rcon;
        } catch (error) {
            console.error('Error during RCON authentication:',
error);
            throw ApiError.Internal('Помилка підключення до
ігрового сервера');
        }
    }

    /**
     * @param {Rcon} rcon
     * @param {Array<string>} commands
     */
    async executeCommands(rcon, commands) {
```

```
        try {
            const MAX_COMMAND_LENGTH = 510;

            let commandBatch = '';
            for (const command of commands) {
                if (typeof command !== 'string') {
                    console.error(`Invalid command type:
${typeof command}, command: ${command}`);
                    continue;
                }

                if ((commandBatch.length + command.length +
(commandBatch.length > 0 ? 2 : 0)) > MAX_COMMAND_LENGTH) {
                    if (commandBatch.length > 0) {
                        console.log(`ComandBatch:\n${commandBatc
h}\n`);
                        try {
                            await rcon.execute(commandBatch);
                        } catch (error) {
                            console.error('Error executing
batch:', error);
                            throw error;
                        }
                    }
                    commandBatch = command;
                } else {
                    if (commandBatch.length > 0) {
                        commandBatch += '; ';
                    }
                    commandBatch += command;
                }
            }

            if (commandBatch.length > 0) {
                console.log(`ComandBatch:\n${commandBatch}\n`);
                try {
                    await rcon.execute(commandBatch);
                } catch (error) {
                    console.error('Error executing final
batch:', error);
                    throw error;
                }
            }
        } catch (error) {
            console.error('Error when executing RCON commands:',
error);
            throw ApiError.Internal('Error when executing
commands on the game server');
        }
    }

    /**
     * @param {Rcon} rcon
```

```
     * @param {string} command
     * @returns {Promise<string>}
     */
    async executeCommand(rcon, command) {
        try {
            const response = await rcon.execute(command);
            return response;
        } catch (error) {
            console.error(`Error executing RCON command
"${command}":`, error);
            throw ApiError.Internal(`Error executing RCON
command: ${command}`);
        }
    }

    /**
     * @param {Object} server
     * @param {string} message
     * @returns {Promise<void>}
     */
    async sendSay(server, message) {
        if (!message || typeof message !== 'string') {
            throw ApiError.BadRequest('Invalid message');
        }

        const rcon = await this.createRconClient(server);

        try {
            await rcon.execute(`say ${message}`);
        } catch (e) {
            throw e;
        } finally {
            await rcon.disconnect();
        }
    }


}

module.exports = new RconService();
```

Б7 services/logService.js

```
const fs = require('fs');
const path = require('path');

class LogService {
    constructor() {
        this.listeners = {};
        this.logStreams = {};

        this.logsDirectory = path.join(__dirname, '..', 'logs');
```

```
        if (!fs.existsSync(this.logsDirectory)) {
            fs.mkdirSync(this.logsDirectory, { recursive: true
});
        }
    }

    addListener(matchId, callback) {
        if (!this.listeners[matchId]) {
            this.listeners[matchId] = [];
            this.createLogStream(matchId);
        }
        this.listeners[matchId].push(callback);
    }

    removeListener(matchId, callback) {
        if (this.listeners[matchId]) {
            this.listeners[matchId] =
this.listeners[matchId].filter(cb => cb !== callback);
            if (this.listeners[matchId].length === 0) {
                delete this.listeners[matchId];
                this.closeLogStream(matchId);
            }
        }
    }

    handleLog(matchId, log) {
        if (this.listeners[matchId]) {
            for (const callback of this.listeners[matchId]) {
                // console.log(`[Match ${matchId}] ${log}`);
                callback(log);
            }
            this.writeLogToFile(matchId, log);
        }
    }

    createLogStream(matchId) {
        const logFilePath = path.join(this.logsDirectory,
`match_${matchId}.log`);
        const stream = fs.createWriteStream(logFilePath, {
flags: 'a' });
        this.logStreams[matchId] = stream;
    }

    closeLogStream(matchId) {
        if (this.logStreams[matchId]) {
            this.logStreams[matchId].end();
            delete this.logStreams[matchId];
        }
    }

    writeLogToFile(matchId, log) {
        if (this.logStreams[matchId]) {
            const timestamp = new Date().toISOString();
```

```
            this.logStreams[matchId].write(`[${timestamp}]
${log}\n`);
        }
    }
}

module.exports = new LogService();
```

Б8 services/gameServerService.js

```
const { GameServer, Match } = require('../models');
const ApiError = require('../exceptions/apiError');
const RconService = require('./rconService');
const { encryptPassword, decryptPassword } =
require('../utils/encryption');
const { validateIpAddress } = require('../utils/validation');
const { warmupConfig, undoWarmupConfig, knifeConfig,
undoKnifeConfig, competitiveConfig } =
require('../constants/serverConfigs');

class ServerService {

    async createServer(serverData) {
        const {
            s_ip_address,
            s_rcon_password,
            s_hostname
        } = serverData;

        if (!s_ip_address || !s_rcon_password) {
            throw ApiError.BadRequest('IP address and RCON
password are required');
        }

        validateIpAddress(s_ip_address);

        const encryptedPassword =
encryptPassword(s_rcon_password);

        const server = await GameServer.create({
            s_ip_address,
            s_rcon_password: encryptedPassword,
            s_hostname,
        });

        return {
            s_id: server.s_id,
            s_ip_address: server.s_ip_address,
            s_hostname: server.s_hostname,
        };
    }

    async checkServerStatus(serverId) {
```

```
        const server = await GameServer.findByPk(serverId);

        if (!server) {
            throw ApiError.NotFound('Server not found');
        }

        const rcon = await RconService.createRconClient(server);

        try {
            const response = await rcon.execute('status');
            await rcon.disconnect();

            return {
                online: true,
                response,
            };
        } catch (error) {
            return {
                online: false,
                error: error.message,
            };
        }

    }

    async getAllServers(page = 1, limit = 5) {
        const offset = (page - 1) * limit;
        const { count, rows } = await
GameServer.findAndCountAll({
            attributes: ['s_id', 's_ip_address', 's_hostname',
's_rcon_password'],
            limit,
            offset,
            order: [['s_id', 'ASC']],
        });

        const servers = rows.map(server => {
            return {
                s_id: server.s_id,
                s_ip_address: server.s_ip_address,
                s_hostname: server.s_hostname,
                s_rcon_password:
decryptPassword(server.s_rcon_password),
            };
        });

        return {
            servers,
            total: count,
            page,
            totalPages: Math.ceil(count / limit),
        };
    }
```

```
    async getServerById(serverId) {
        const server = await GameServer.findByPk(serverId, {
            attributes: ['s_id', 's_ip_address', 's_hostname',
's_port'],
        });
        if (!server) {
            throw ApiError.NotFound('Server not found.');
        }
        return server;
    }

    async updateServer(serverId, serverData) {
        const server = await GameServer.findByPk(serverId);
        if (!server) {
            throw ApiError.NotFound('Server not found.');
        }

        if (serverData.s_ip_address) {
            validateIpAddress(serverData.s_ip_address);
        }

        if (serverData.s_rcon_password) {
            serverData.s_rcon_password =
encryptPassword(serverData.s_rcon_password);
        }

        await server.update(serverData);
        return {
            s_id: server.s_id,
            s_ip_address: server.s_ip_address,
            s_hostname: server.s_hostname,
        };
    }

    async deleteServer(serverId) {
        const server = await GameServer.findByPk(serverId);
        if (!server) {
            throw ApiError.NotFound('Server not found.');
        }

        await server.destroy();
        return {
            message: 'Server deleted successfully'
        };
    }

    async prepareServerForMatch(rcon, matchId) {
        const match = await Match.findByPk(matchId, {
            include: [{
                model: GameServer,
                as: 'server',
            }, ],
```

```
        });

        if (!match) {
            throw ApiError.NotFound('Match not found');
        }

        const server = match.server;

        if (!server) {
            throw ApiError.BadRequest('Game server is not
assigned for the match');
        }

        try {
            const commands = [
                'log on',
                `logaddress_add_http
"${process.env.LOG_SERVER_ADDRESS}/${matchId}"`,
            ];

            await RconService.executeCommands(rcon, commands);
            console.log('Server prepared\n');
        } catch (error) {
            throw error;
        }
    }

    async executeCompetitiveConfig(rcon, maxRounds) {
        try {
            await RconService.executeCommands(rcon,
competitiveConfig(maxRounds));
            console.log('Competitive config executed\n');
        } catch (error) {
            throw error;
        }
    }

    async executeWarmupConfig(rcon) {
        try {
            await RconService.executeCommands(rcon,
warmupConfig);
            console.log('Warmup config executed\n');
        } catch (error) {
            throw error;
        }
    }

    async undoWarmupConfig(rcon) {
        try {
            await RconService.executeCommands(rcon,
undoWarmupConfig);
        } catch (error) {
            throw error;
```

```
        }
    }

    async executeKnifeConfig(rcon) {
        try {
            await RconService.executeCommands(rcon,
knifeConfig);
            console.log('Knife config executed\n');
        } catch (error) {
            throw error;
        }
    }

    async undoKnifeConfig(rcon) {
        try {
            await RconService.executeCommands(rcon,
undoKnifeConfig);
        } catch (error) {
            throw error;
        }
    }
}

module.exports = new ServerService();
```

Б9 App.js

```
const { GameServer, Match } = require('../models');
const ApiError = require('../exceptions/apiError');
const RconService = require('./rconService');
const { encryptPassword, decryptPassword } =
require('../utils/encryption');
const { validateIpAddress } = require('../utils/validation');
const { warmupConfig, undoWarmupConfig, knifeConfig,
undoKnifeConfig, competitiveConfig } =
require('../constants/serverConfigs');

class ServerService {

    async createServer(serverData) {
        const {
            s_ip_address,
            s_rcon_password,
            s_hostname
        } = serverData;

        if (!s_ip_address || !s_rcon_password) {
            throw ApiError.BadRequest('IP address and RCON
password are required');
        }

        validateIpAddress(s_ip_address);
```

```
        const encryptedPassword =
encryptPassword(s_rcon_password);

        const server = await GameServer.create({
            s_ip_address,
            s_rcon_password: encryptedPassword,
            s_hostname,
        });

        return {
            s_id: server.s_id,
            s_ip_address: server.s_ip_address,
            s_hostname: server.s_hostname,
        };
    }

    async checkServerStatus(serverId) {
        const server = await GameServer.findByPk(serverId);

        if (!server) {
            throw ApiError.NotFound('Server not found');
        }

        const rcon = await RconService.createRconClient(server);

        try {
            const response = await rcon.execute('status');
            await rcon.disconnect();

            return {
                online: true,
                response,
            };
        } catch (error) {
            return {
                online: false,
                error: error.message,
            };
        }

    }

    async getAllServers(page = 1, limit = 5) {
        const offset = (page - 1) * limit;
        const { count, rows } = await
GameServer.findAndCountAll({
            attributes: ['s_id', 's_ip_address', 's_hostname',
's_rcon_password'],
            limit,
            offset,
            order: [['s_id', 'ASC']],
        });
```

```
        const servers = rows.map(server => {
            return {
                s_id: server.s_id,
                s_ip_address: server.s_ip_address,
                s_hostname: server.s_hostname,
                s_rcon_password:
decryptPassword(server.s_rcon_password),
            };
        });

        return {
            servers,
            total: count,
            page,
            totalPages: Math.ceil(count / limit),
        };
    }

    async getServerById(serverId) {
        const server = await GameServer.findByPk(serverId, {
            attributes: ['s_id', 's_ip_address', 's_hostname',
's_port'],
        });
        if (!server) {
            throw ApiError.NotFound('Server not found.');
        }
        return server;
    }

    async updateServer(serverId, serverData) {
        const server = await GameServer.findByPk(serverId);
        if (!server) {
            throw ApiError.NotFound('Server not found.');
        }

        if (serverData.s_ip_address) {
            validateIpAddress(serverData.s_ip_address);
        }

        if (serverData.s_rcon_password) {
            serverData.s_rcon_password =
encryptPassword(serverData.s_rcon_password);
        }

        await server.update(serverData);
        return {
            s_id: server.s_id,
            s_ip_address: server.s_ip_address,
            s_hostname: server.s_hostname,
        };
    }

    async deleteServer(serverId) {
```

```
        const server = await GameServer.findByPk(serverId);
        if (!server) {
            throw ApiError.NotFound('Server not found.');
        }

        await server.destroy();
        return {
            message: 'Server deleted successfully'
        };
    }

    async prepareServerForMatch(rcon, matchId) {
        const match = await Match.findByPk(matchId, {
            include: [{
                model: GameServer,
                as: 'server',
            }, ],
        });

        if (!match) {
            throw ApiError.NotFound('Match not found');
        }

        const server = match.server;

        if (!server) {
            throw ApiError.BadRequest('Game server is not
assigned for the match');
        }

        try {
            const commands = [
                'log on',
                `logaddress_add_http
"${process.env.LOG_SERVER_ADDRESS}/${matchId}"`,
            ];

            await RconService.executeCommands(rcon, commands);
            console.log('Server prepared\n');
        } catch (error) {
            throw error;
        }
    }

    async executeCompetitiveConfig(rcon, maxRounds) {
        try {
            await RconService.executeCommands(rcon,
competitiveConfig(maxRounds));
            console.log('Competitive config executed\n');
        } catch (error) {
            throw error;
        }
    }
```

```
    async executeWarmupConfig(rcon) {
        try {
            await RconService.executeCommands(rcon,
warmupConfig);
            console.log('Warmup config executed\n');
        } catch (error) {
            throw error;
        }
    }

    async undoWarmupConfig(rcon) {
        try {
            await RconService.executeCommands(rcon,
undoWarmupConfig);
        } catch (error) {
            throw error;
        }
    }

    async executeKnifeConfig(rcon) {
        try {
            await RconService.executeCommands(rcon,
knifeConfig);
            console.log('Knife config executed\n');
        } catch (error) {
            throw error;
        }
    }

    async undoKnifeConfig(rcon) {
        try {
            await RconService.executeCommands(rcon,
undoKnifeConfig);
        } catch (error) {
            throw error;
        }
    }
}

module.exports = new ServerService();
```