ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Backend-розробка web-платформи для менеджменту ігрової соціальної структури "SOKIL"»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

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

_____ Андрій ДИКАЛО
(підпис)

Виконав: здобувач вищої освіти групи ПД-41

Андрій ДИКАЛО

Керівник: Ірина ЗАМРІЙ
д.т.н., професор

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дикалу Андрію Юрійовичу

1. Тема кваліфікаційної роботи: «Backend-розробка web-платформи для менеджменту ігрової соціальної структури "SOKIL"»
керівник кваліфікаційної роботи д.т.н., професор Ірина ЗАМРІЙ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи розробки web-платформ, опис роботи API, документація JWT, MongoDB, Docker, а також фреймворків Spring Boot, Spring Security та Lombok.

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

1. Провести аналіз предметної галузі гри та особливостей функціонування полкових структур.
2. Дослідити існуючі сервіси для управління ігровими акаунтами та технікою в онлайн-іграх.
3. Проаналізувати сучасні технології розробки безпечних web-застосунків (Spring Boot, Spring Security, MongoDB, Docker, JWT).

4. Сформувати вимоги до функціоналу та архітектури web-платформи.
5. Спроєктувати архітектуру клієнт-серверного застосунку з урахуванням обраного стеку технологій.
6. Реалізувати REST API для управління користувачами, акаунтами, технікою, командами та чорним списком.
7. Розробити механізм авторизації та автентифікації з використанням JWT і захистом CORS.
8. Провести тестування основних компонентів системи.
9. Здійснити розгортання web-платформи з використанням Docker.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма варіантів використання.
5. Діаграма діяльності.
6. Діаграма архітектури безпеки web-платформи.
7. Діаграма класів безпеки.
8. Діаграма загальної архітектури.
9. Екранні форми.
10. Апробація результатів дослідження

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Провести аналіз предметної галузі гри та особливостей функціонування полкових структур	14.03-20.03.2025	
4	Дослідити існуючі сервіси для управління ігровими акаунтами та технікою в онлайн-іграх	21.03-25.03.2025	
5	Проаналізувати сучасні технології розробки безпечних web-застосунків (Spring Boot, Spring Security, MongoDB, Docker, JWT)	26.03-30.03.2025	
6	Сформувати вимоги до функціоналу та архітектури web-платформи	31.03-04.04.2025	

7	Спроєктувати архітектуру клієнт-серверного застосунку з урахуванням обраного стеку технологій	05.04-11.04.2025	
8	Реалізувати REST API для управління користувачами, акаунтами, технікою, командами та чорним списком	12.04-26.04.2025	
9	Розробити механізм авторизації та автентифікації з використанням JWT і захистом CORS	27.04-01.05.2025	
10	Провести тестування основних компонентів системи	02.05-09.05.2025	
11	Здійснити розгортання web-платформи з використанням Docker	10.05-14.05.2025	
12	Оформлення роботи: вступ, висновки, реферат	15.05-16.05.2025	
13	Розробка демонстраційних матеріалів	17.05-18.05.2025	
14	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Андрій ДИКАЛО
(підпис)

Керівник
кваліфікаційної роботи _____ Ірина ЗАМРІЙ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: _65_ стор., __1__ табл., _15_ рис., _27_ джерел.

Мета роботи – спрощення менеджменту організації учасників ігрової онлайн-спільноти.

Об'єкт дослідження – процес функціонування ігрових онлайн-спільнот як соціальних структур у цифровому середовищі.

Предмет дослідження – web-платформа для підтримки та організації учасників ігрової спільноти.

Короткий зміст роботи: У роботі проведено аналіз предметної галузі гри War Thunder, зокрема функціонування полкових структур, що беруть участь у сезонних боях. Досліджено аналогічні сервіси для керування акаунтами гравців та ігровою технікою, виявлено їх недоліки. На основі цього сформовано функціональні та нефункціональні вимоги до системи. Розроблено архітектуру web-платформи для менеджменту акаунтів, гравців, техніки, команд, а також для моніторингу прогресу користувачів та підтримки взаємодії між ними. Реалізовано REST API з автентифікацією на базі JWT, шифруванням паролів за допомогою BCrypt і захистом CORS. Забезпечено можливість передачі доступу до акаунтів іншим користувачам, визначення метової техніки за сезоном, а також формування чорного списку гравців. В роботі використано такий стек технологій: Spring Boot, Spring Security, MongoDB, JWT, Docker. Здійснено тестування основних компонентів, а також повне розгортання проєкту в контейнеризованому середовищі.

Сферою використання застосунку є організація ігрового процесу в онлайн-спільнотах, що базуються на багатокористувацьких іграх з клановою структурою, зокрема в рамках спільноти SOKIL.

КЛЮЧОВІ СЛОВА: ІГРОВА СПІЛЬНОТА, МЕНЕДЖМЕНТ АККАУНТІВ, SPRING BOOT, MONGODB, JWT, WEB-ПЛАТФОРМА, WAR THUNDER, ІГРОВА ТЕХНІКА, ДОСТУП ДО АККАУНТІВ, ЧОРНИЙ СПИСОК

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface

JSON – JavaScript Object Notation

REST – Representational State Transfer

NoSQL – non Structured Query Language

DTO – Data Transfer Object

CORS – Cross-Origin Resource Sharing

CSRF – Cross-Site Request Forgery

CRUD – Create, Read, Update, Delete

ВСТУП

Актуальність теми зумовлена зростанням ролі онлайн-спільнот у цифровому просторі, зокрема в ігровій індустрії, де ефективна взаємодія між учасниками потребує спеціалізованих інструментів управління. В умовах активного розвитку командних ігор, таких як War Thunder, постає необхідність в оптимізації процесів організації гравців, розподілу ресурсів та керування колективною діяльністю. Відсутність централізованих платформ для менеджменту таких спільнот ускладнює організацію внутрішніх процесів, координацію подій та контроль за дотриманням правил.

Розробка web-платформи дозволяє вирішити ці проблеми, забезпечуючи гнучке, безпечне та масштабоване середовище для управління ігровими соціальними структурами. Вона дозволяє ефективно об’єднувати акаунти гравців, формувати команди, визначати метову техніку, надавати доступ до акаунтів, а також підтримує систему чорних списків для прозорості взаємодії. Враховуючи широку популярність багатокористувацьких ігор з клановою структурою, така платформа є актуальною та затребуваною як серед гравців, так і серед керівників онлайн-спільнот.

Об’єктом дослідження є процес функціонування ігрових онлайн-спільнот як соціальних структур у цифровому середовищі.

Предметом дослідження є web-платформа для підтримки та організації учасників ігрової спільноти.

Метою роботи є спрощення менеджменту організації учасників ігрової онлайн-спільноти.

Методи дослідження: першим етапом дослідження було вивчення функціональних особливостей ігрової спільноти War Thunder, зокрема полкових структур і способу організації сезонної взаємодії гравців. Далі проведено огляд існуючих сервісів для менеджменту акаунтів, ігрової техніки та взаємодії між користувачами з метою формування вимог до майбутньої системи.

На основі сформованих вимог було обрано сучасний технологічний стек для реалізації платформи: Spring Boot для серверної логіки, Spring Security для впровадження автентифікації та авторизації, MongoDB як NoSQL базу даних, JWT для токен-базованої безпеки, а також Docker для контейнеризації та зручного розгортання системи. Розробку REST API та ключових сервісів здійснено з урахуванням принципів безпеки, масштабованості й зручності користування. Тестування системи здійснювалося під час безпосередньої реалізації із застосуванням модульного та інтеграційного підходів.

Наукова новизна роботи полягає у створенні спеціалізованої web-платформи, яка поєднує засоби управління ігровими акаунтами, системою техніки, командною організацією та соціальною взаємодією в рамках клановой спільноти. Реалізовано функціонал, що дозволяє:

- централізовано керувати декількома акаунтами одного гравця;
- формувати списки доступу до акаунтів для командної гри;
- визначати метову (найефективнішу) техніку за сезонами;
- будувати гнучку модель користувацьких прав доступу;
- вести облік негативної поведінки гравців через чорні списки.

Платформа забезпечує зручну, безпечну та адаптивну структуру управління колективною ігровою діяльністю, що не має прямих аналогів у відкритому доступі в контексті гри War Thunder.

Практична значущість результатів полягає у можливості використання розробленої web-платформи для ефективного менеджменту учасників ігрової спільноти SOKIL, а також потенційної адаптації для інших ігрових організацій, що діють у подібному форматі. Система дозволяє автоматизувати організаційні процеси, спростити взаємодію між гравцями, підвищити ефективність командної координації, а також забезпечити надійний контроль над ресурсами (акаунтами, технікою) у спільноті.

Завдяки використанню сучасних технологій платформа може бути розгорнута на більшості серверів і підтримувати багатьох користувачів одночасно, що робить її зручною для масштабованого використання у сфері онлайн-геймінгу.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Упродовж останніх років традиційну індустрію спортивних змагань поступово витісняє більш молода, але динамічно зростаюча індустрія кіберспорту – галузь, що поєднує конкурентну гру, організацію турнірів і розвиток соціальних структур у межах ігрових спільнот. Завдяки технологіям будь-яка людина, маючи доступ до комп’ютера та інтернету, може стати професійним кіберспортсменом, будучи задіяною в улюбленій грі [1]. На сьогодні турніри з великими грошовими призами, у яких гравці змагаються в комп’ютерних іграх, стали буденністю, що свідчить про комерціалізацію та легітимізацію кіберспорту.

Особливої популярності набули багатокористувацькі онлайн-ігри, у яких успіх залежить не лише від індивідуальної майстерності гравця, а й від ефективної командної взаємодії. Саме тому ігрові спільноти – кланові або полкові структури – стали фундаментальним елементом сучасного геймплею. Вони формують ієрархію, розподіляють обов’язки, керують спільними ресурсами й навіть мають власні системи дисциплінарного контролю, подібно до традиційних соціальних або корпоративних організацій.

Гра War Thunder, як частина кіберспортивного простору, має розвинену ігрову екосистему. У грі реалізовано систему полків (аналогів кланів), участь у сезонних полкових боях 8 на 8, накопичення полкових очок, а також можливість гри з кількох акаунтів. Кожен акаунт має власний набір бойової техніки, яка має ступінь дослідження та вподобаності, що додає додаткової складності в менеджменті й стратегічному використанні ресурсів.

Для ефективного управління цією багаторівневою структурою необхідна спеціалізована цифрова платформа, яка б автоматизувала процеси керування акаунтами, технікою, аналітикою й формуванням команд. У цьому контексті web-платформа SOKIL розробляється як відповідь на потребу в централізованому,

гнучкому та безпечному інструменті для менеджменту спільноти гравців War Thunder. Вона забезпечує можливості:

- збереження та адміністрування акаунтів;
- керування технікою й відстеження прогресу її дослідження;
- формування команд;
- визначення метової (найбільш ефективної) техніки на основі аналітики користувацької активності.

Сучасні ігрові екосистеми потребують чіткої організації інформаційного простору, що охоплює контроль акаунтів, аналіз ігрової статистики, визначення актуальної техніки для сезону та виявлення слабких ланок у командній структурі. Згідно з дослідженням [2], саме автоматизація аналітики та управління ігровими одиницями дозволяє значно підвищити ефективність функціонування ігрових команд, зокрема в умовах конкурентного кіберспортивного середовища.

У War Thunder, де гравці часто мають по кілька акаунтів з різним складом техніки, ефективне адміністрування без відповідного інструменту є практично неможливим. Таким чином, розробка системи “SOKIL” є не просто зручним доповненням до ігрового процесу, а необхідним елементом керування спільнотою в умовах цифрової складності та високих вимог до оперативного прийняття рішень.

1.1 Роль кіберспорту та ігрових спільнот

1.1.1 Кіберспорт як культурний та соціальний феномен

Кіберспорт, або електронний спорт, перетворився з нішевого хобі на глобальне явище, що охоплює мільйони гравців і глядачів по всьому світу. Змагання з таких ігор, як League of Legends, Dota 2, Counter-Strike: Global Offensive та War Thunder, приваблюють величезні аудиторії як онлайн, так і офлайн. Це свідчить про зростаючу легітимізацію кіберспорту як форми розваги та професійної діяльності [3].

Кіберспорт також сприяє формуванню нових культурних практик. Гравці та команди стають об'єктами фанатського захоплення, подібно до традиційних

спортсменів. Фанати слідкують за їхніми досягненнями, беруть участь у фан-клубах, створюють фан-арт та інші форми творчості, що зміцнює спільноти навколо улюблених ігор та гравців [4].

1.1.2 Структура ігрових спільнот: клани, гільдії та полки

Ігрові спільноти організовуються в різні форми, такі як клани, гільдії або полки, залежно від гри та її механік. Ці об'єднання дозволяють гравцям координувати дії, розподіляти ролі та спільно досягати цілей у грі. Наприклад, у War Thunder існують полки – організовані групи гравців, які беруть участь у спільних боях та турнірах.

Структура таких спільнот часто включає лідерів, офіцерів та рядових членів, кожен з яких має свої обов’язки та ролі. Це сприяє розвитку навичок командної роботи, лідерства та стратегічного мислення серед учасників [5].

1.1.3 Соціальний вплив ігрових спільнот

Ігрові спільноти мають значний соціальний вплив. Вони сприяють формуванню почуття приналежності, особливо серед молоді, яка може відчувати себе ізольованою в реальному житті. Участь у спільнотах дозволяє гравцям знайомитися з однодумцями, обмінюватися досвідом та підтримувати один одного [6].

Крім того, ігрові спільноти часто беруть участь у благодійних ініціативах, організовують турніри для збору коштів та підтримки різних соціальних проєктів. Це демонструє потенціал таких структур як платформи для позитивних соціальних змін [4].

1.1.4 Освітній потенціал кіберспорту та ігрових спільнот

Кіберспорт та участь в ігрових спільнотах можуть мати освітній ефект. Гравці розвивають навички стратегічного мислення, швидкого прийняття рішень, комунікації та співпраці. Деякі освітні заклади вже інтегрують кіберспорт у свої

навчальні програми, визнаючи його потенціал у формуванні важливих професійних компетенцій у студентів [7].

Крім того, участь у багатонаціональних ігрових спільнотах сприяє вивченню іноземних мов, зокрема англійської, яка зазвичай використовується як мова міжнародної комунікації [8].

1.2 Особливості гри War Thunder

1.2.1 Ігрова механіка: типи техніки, режими боїв

War Thunder – це багатокористувацький онлайн-симулятор бойової техніки, що охоплює повітряні, наземні та морські бої. Гра пропонує понад 2000 одиниць техніки з різних періодів XX–XXI століть, що представляють десятки країн. Основні типи техніки включають танки, літаки, кораблі та вертольоти. Кожна одиниця має власну модель бронювання, боєкомплекту, швидкості та керованості, що моделюється з високою фізичною точністю.

Гравці можуть брати участь у трьох основних режимах боїв: аркадному, реалістичному та симуляторному. Кожен із них має власні правила, ступінь фізичної моделі, поведінки техніки та інтерфейсу. Наприклад, у симуляторному режимі відсутній інтерфейс прицілювання, що наближає досвід до реального бойового керування. Такий рівень деталізації вимагає від гравців стратегічного планування, знання техніки та вміння командної взаємодії.

1.2.2 Полки в War Thunder: принципи функціонування, ролі та очки

Полки в War Thunder є аналогом кланів або гільдій у інших MMO-іграх. Це організовані об’єднання гравців, що дозволяють брати участь у спеціальних подіях – полкових боях, змаганнях і сезонних турнірах. Гравці можуть створювати власні полки, вступати до вже існуючих, отримувати внутрішні ролі (командири, офіцери, рядові учасники) та разом накопичувати полкові очки (Squadron Activity Points), які відображають активність і успіхи полку.

Полкові бої проводяться у форматі 8 на 8, де кожна команда представляє свій полк. Перемоги приносять очки, що впливають на загальний рейтинг полку в сезоні. Сезонна система ранжування додає конкурентної мотивації та потребує від команд стратегічної підготовки, зокрема правильного підбору техніки та складу учасників.

1.2.3 Проблематика використання кількох акаунтів

У межах гри багато гравців використовують кілька акаунтів із різною технікою, прокачуванням і рівнем досліджень. Це може бути пов’язано з бажанням експериментувати з різними націями або оптимізувати участь у турнірах та полкових боях. Однак таке розділення інформації створює проблему централізованого управління: важко контролювати, який акаунт має яку техніку, яку активність демонструє гравець і як оптимально залучити його до бою.

Ця ситуація ускладнює командоутворення в полку, особливо у відповідальні моменти полкових сезонів, де кожна одиниця техніки й кожен гравець відіграють критичну роль. Відсутність системи централізованого обліку акаунтів і техніки призводить до помилок у підборі складу, дублювання ролей або неефективного використання доступних ресурсів.

1.2.4 Необхідність інструментів для організації командної гри

Для успішного функціонування полку в War Thunder потрібна добре налагоджена командна взаємодія. До ключових задач входить формування боєздатних груп, відстеження прогресу кожного гравця, аналіз його активності, а також вибір оптимальної техніки для конкретного бою. Ці задачі неможливо ефективно реалізувати вручну, особливо при великій кількості учасників.

Тому виникає потреба у цифрових інструментах, які дозволяють:

- централізовано зберігати інформацію про акаунти та техніку;
- визначати метову (найбільш ефективну) техніку для конкретного періоду сезону;
- формувати оптимальні команди під заплановані бої;

- аналізувати активність користувачів;
- забезпечувати доступ до даних керівництву полку в реальному часі.

1.3 Використання кіберспортивних ігор у веденні бойових дій

Говорячи про симулятори військової техніки, неможливо оминути таку тему, як їх використання для підготовки військових до місій. В умовах повномасштабного вторгнення російських військ, навіть в Україні широко розвинулася індустрія симуляторів для військових, таких як тренажери для екіпажів танків, обслуговування різноманітних механізмів, керування наземною та повітряною технікою та її озброєнням, а зокрема FPV-дронами.

1.3.1 Ігри як інструмент підготовки та моделювання бойових ситуацій

Використання відеоігор у військовій підготовці стало поширеною практикою у багатьох країнах світу. Наприклад, у 1990-х роках Корпус морської піхоти США адаптував гру Doom II для тренування тактичних навичок у командній роботі та прийнятті рішень у бойових умовах [9]. Ця модифікація, відома як Marine Doom, дозволяла військовим відпрацьовувати сценарії бою в безпечному віртуальному середовищі [10].

Іншим прикладом є використання гри War Thunder для тренування військових. У 2024 році український військовослужбовець, стрілець 47 ОМБр «Маґура», повідомив журналістам, що завдяки знанням про технічні характеристики, отриманим у цій грі, зумів ефективно вразити російський танк Т-90М, запам’ятавши його вразливі місця [11-12].

1.3.2 Ігри як засіб рекрутингу та популяризації військової служби

Відеоігри також використовуються як інструмент для залучення молоді до військової служби. У 2002 році армія США розробила гру America's Army, яка стала ефективним засобом рекрутингу, дозволяючи гравцям ознайомитися з військовою службою через інтерактивний досвід [13].

Крім того, військові активно використовують платформи для стрімінгу, такі як Twitch, для взаємодії з молодою аудиторією. Це дозволяє не лише популяризувати військову службу, але й демонструвати її сучасний технологічний аспект [14].

1.3.3 Етичні аспекти використання ігор у військовому контексті

Хоча використання відеоігор у військових цілях має багато переваг, воно також викликає етичні питання. Зокрема, важливо зазначити, що використання ігор для рекрутингу молоді може романтизувати війну та не відображати її реалій.

Існують побоювання щодо використання ігор для тренування навичок, пов'язаних із насильством. Наприклад, симулятор Multi-purpose Arcade Combat Simulator (MACS), розроблений армією США, був критикований за те, що він може сприяти розвитку навичок стрільби у молоді [15].

1.4 Проблематика управління ігровою спільнотою

1.4.1 Типові задачі адміністрування кланів

Ігрові спільноти, особливо в багатокористувацьких онлайн-іграх, мають складну внутрішню структуру, яка вимагає постійного управління. До типових задач адміністратора клану (полку) належать: прийом і модерація нових учасників, розподіл ролей, призначення обов’язків, організація тренувань, формування складів на турніри, контроль активності та підтримка дисципліни. У деяких іграх адміністраторам також доводиться займатися розподілом внутрішньоігрових ресурсів, моніторингом ефективності гравців та веденням внутрішньої документації [16].

Ці процеси зазвичай реалізуються вручну або з використанням сторонніх інструментів, таких як Google Таблиці, Discord або Telegram-боти, що створює фрагментованість управлінської інфраструктури [17].

1.4.2 Проблеми трекінгу активності та техніки

Однією з основних проблем адміністрування є трекинг активності користувачів. Часто немає інтегрованого способу визначити, хто з учасників є активним, хто останнім часом не з’являвся в боях, а також яка техніка є у розпорядженні гравців. Ці дані важливі для: коректного формування команд, планування боїв, аналізу ефективності окремих гравців і техніки.

У грі War Thunder, наприклад, немає детального перегляду стану дослідження техніки чи її ефективності у вигляді аналітичної візуалізації конкретного акаунта. Усе це потребує ручного збору даних або використання саморобних API-інтеграцій, що є технічно складним та часто ненадійним.

1.4.3 Потреба в централізованих, масштабованих рішеннях

Оскільки типові засоби адміністрування в ігрових спільнотах є розрізненими, виникає потреба у централізованій платформі, яка б поєднувала всі ключові функції управління: базу гравців, облік акаунтів, інвентаризацію техніки, аналітику активності, формування складів і внутрішню комунікацію. Така система повинна бути масштабованою, щоб витримувати збільшення кількості користувачів, техніки, сезонів та бойових подій.

1.4.4 Вимоги до безпеки web-платформи

З огляду на чутливість даних (акаунти, історія активності, склади команд), особливу увагу слід приділяти інформаційній безпеці. Платформа повинна підтримувати автентифікацію за допомогою JWT-токенів, захист даних через HTTPS та CORS-політики, шифрування паролів та обмеження доступу до певних функцій залежно від ролі користувача.

1.5 Аналіз аналогів

У сучасному ігровому середовищі кланові об’єднання стали ключовими елементами соціальної структури багатьох багатокористувацьких ігор.

Середовище гри War Thunder не є виключенням. Полки беруть участь у змагальних сезонах, де важливо оперативно координувати дії гравців, оптимізувати використання техніки, враховувати статистику акаунтів, обирати найефективніші стратегії. Це породжує необхідність у розробці систем, які б дозволяли автоматизувати й організовувати ці процеси. Подібні системи часто називають платформами для внутрішнього менеджменту кланів, гільдій чи полків.

Загалом, такі системи можна поділити на дві категорії:

- загальні інструменти менеджменту геймерських спільнот (Discord-боти, Google Sheets, Notion тощо);
- спеціалізовані рішення, розроблені під конкретні ігри;

Розробка таких систем вимагає врахування особливостей гри, цільової аудиторії, технічних обмежень (наявність API, формат даних, системи захисту). Проте офіційний API обмежений, тому велика частина даних має підтримуватись вручну або інтегруватись через парсинг чи неофіційні бази.

1.5.1 TS (Thunder Skill)

TS — це аналітична платформа, що надає статистику по гравцях War Thunder, включаючи K/D, кількість боїв, популярну техніку тощо [18]. Вона фокусується на оцінюванні ефективності гравців у випадкових боях. Приклад структури даного web-застосунку наведено на рисунку 1.1.

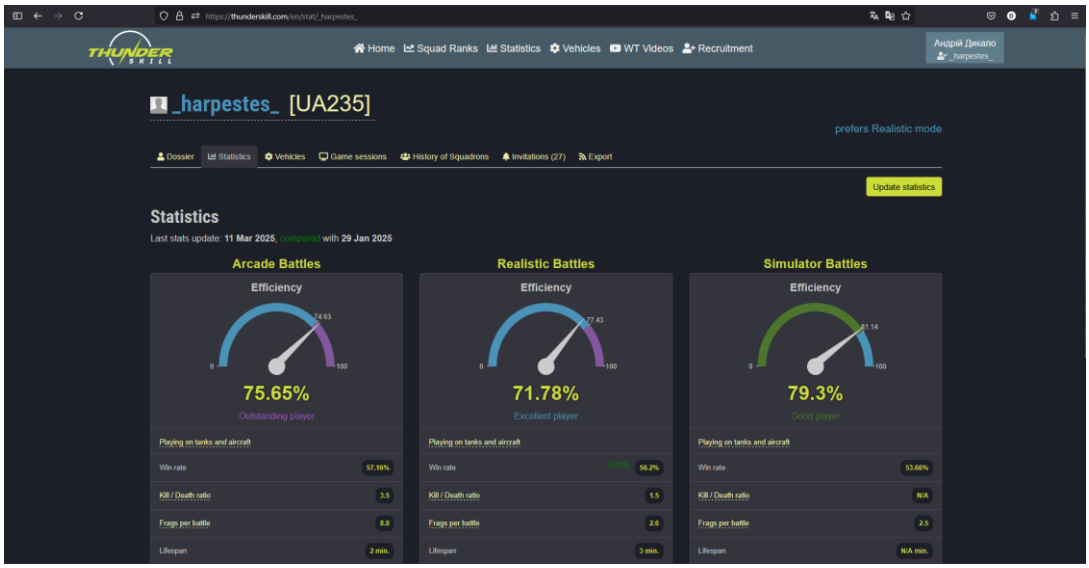


Рис. 1.1 Приклад структури web-застосунку Thunder Skill

Перевагами цієї системи є велика кількість статистичних даних, можливість порівняння статистики гравців між собою а також популярність серед гравців.

Серед недоліків можна виділити відсутність можливості керування акаунтами, полкової координації, обліку техніки на кількох акаунтах та цих акаунтів та відсутність підтримки інтерактивного керування складом команди.

1.5.2 Боти з кастомними функціями

Більшість полків у War Thunder використовують Discord як основний канал комунікації. Часто у Discord інтегрують ботів для розподілу ролей, визначення найкращої техніки та ведення статистики. Приклад боту можна переглянути на рисунку 1.2.

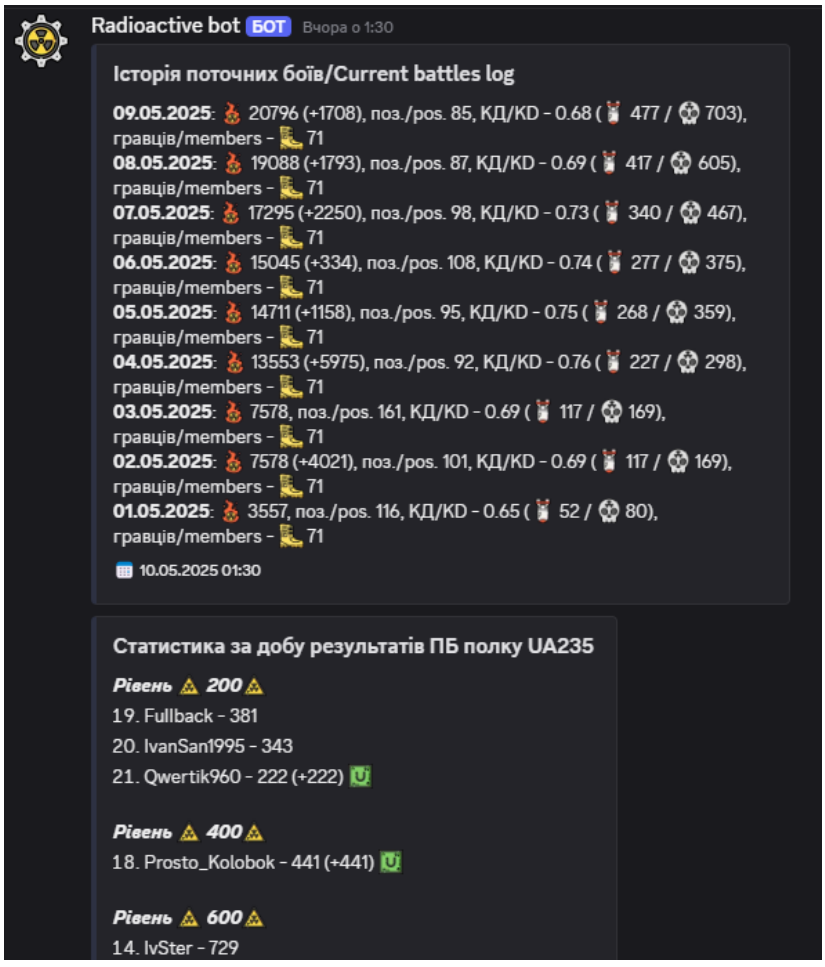


Рис. 1.2 Бот для перегляду статистики в Discord

Перевагами такого підходу є зручна інтеграція з платформою комунікації (Discord, Telegram), швидкий доступ до базових функцій.

Серед недоліків можна виділити відсутність складної бізнес-логіки або управління акаунтами та низьку гнучкість, масштабованість і обмеженість таких ботів.

1.5.3 Офіційна вікі-сторінка War Thunder

War Thunder Wiki – це офіційна вікіпедія по грі War Thunder на якій можна дізнатися про всі ігрові механіки, техніку, озброєння та інших аспектів гри [19]. Основними функціями даного сайту є відображення повних технічних характеристик техніки, опис механік та систем та пояснення новачкам специфіки проведення матчів. Структуру сайту зображено на рисунку 1.3.

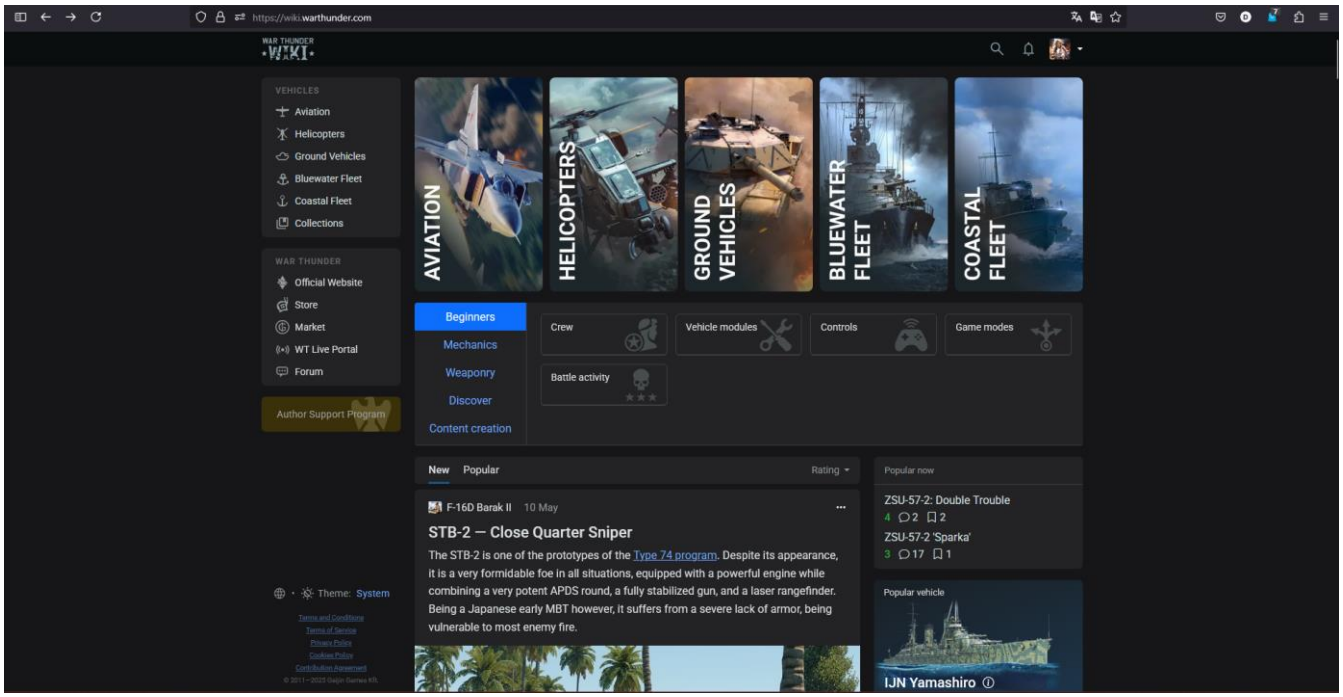


Рис. 1.3 Структура сайту War Thunder Wiki

Перевагами є підтримання актуальності інформації, деталізація технічних даних, зручна структура та відкритий доступ для всіх гравців.

Недоліками можна назвати відсутність персоналізації інформації, відсутність можливості менеджменту чи будь-якої іншої взаємодії з гравцями, відсутність полкової взаємодії і головний недолік – вікі це лише джерело інформації, а не інструмент для менеджменту.

1.5.4 Google Sheets / Notion

Полки також створюють табличні системи для обліку, в які вносять ігрові акаунти, доступну техніку, статистику та ведуть облік гравців.

Перевагами такого підходу є простота та швидкість налаштування, натомість суттєвими недоліками стають відсутність автоматизації, безпеки даних та високий ризик помилок при оновленні даних

1.5.5 Guilded

Guilded – сучасна платформа для управління геймерськими кланами, що пропонує інструменти комунікації, планування та управління спільнотою. Сервіс позиціонується як альтернатива Discord з розширеним функціоналом для командної координації [20]. Цей сервіс дозволяє створювати групи та канали для спілкування, календар подій, списки учасників та розподіл ролей, підтримує інтеграцію з іграми а також базовий менеджмент матчів. Приклад екранної форми застосунку зображено на рисунку 1.4.

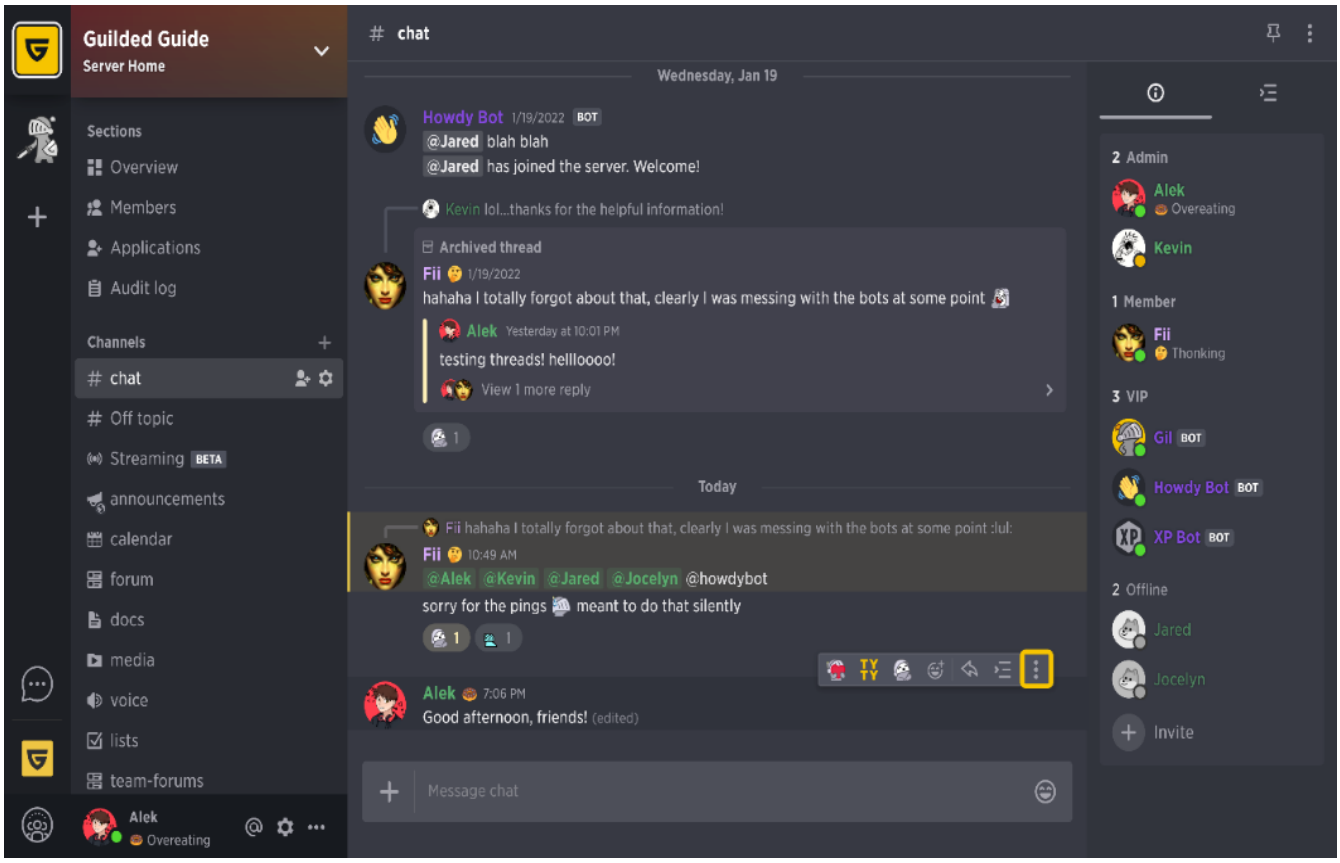


Рис. 1.4 Приклад сервера в застосунку Guilded

Перевагами є поєднання чату, голосових каналів та інструментів для планування, а також можливість використання сервісу як основного хабу для клану.

Серед недоліків варто відзначити відсутність кастомної логіки для War Thunder, обліку техніки та підтримки декількох ігрових акантів. Крім того, система не виконує роль повноцінної інформаційно-аналітичної системи.

1.5.6 Результати аналізу

Зведені результати аналізу характеристик розглянутих застосунків наведено у таблиці 1.1.

Таблиця 1.1

Зведені результати аналізу характеристик застосунків для менеджменту ігрових соціальних структур

Показник	**Thunder Skill**	**Боти**	**War Thunder Wiki**	**Google Sheets / Notion**	**Guilded**	**Web-платформа “SOKIL”**
Підтримка War Thunder	Часткова	Так (не-офіційно)	Так (офіційна база)	Так (ручне введення)	Ні	Так (підлаштована)
Керування акаунтами	Ні	Ні	Ні	Частково	Ні	Так
Керування технікою	Ні	Частково	Так	Частково	Ні	Так
Полкова координація	Ні	Так	Ні	Частково	Так	Так
Автоматизація	Так	Частково	Ні	Ні	Частково	Так
Персоналізація	Ні	Ні	Ні	Частково	Ні	Так
Платформа/ Тип	Web-застосунок	Боти	Вікі-сайт	Табличні системи	Платформа спільнот	Web-застосунок

1.6 Вимоги та обмеження

Успішна розробка інформаційної системи потребує чіткого визначення функціональних і нефункціональних вимог, а також обмежень, які впливають на архітектуру, реалізацію та використання платформи. У випадку проєкту «SOKIL» вимоги формулюються виходячи з потреб користувачів і адміністрації ігрової спільноти, особливостей гри War Thunder, а також стандартів розробки сучасних web-застосунків.

1.6.1 Функціональні вимоги

До функціональних вимог системи належать такі ключові можливості:

- реєстрація та авторизація користувачів (користувачі мають змогу створювати облікові записи лише за наданим адміністраторами реєстраційним кодом. Це дозволяє уникнути несанкціонованого доступу та забезпечити контрольовану реєстрацію нових учасників);
- розмежування прав доступу (астосунок повинен підтримувати ролі з різним рівнем доступу до функцій):
  - рядовий користувач має доступ до персональних акаунтів, може додавати техніку та змінювати параметри дослідження й уподобання, бачити чорний список а також свою статистику;
  - модератор отримує доступ до перегляду акаунтів інших користувачів, чорного списку, а також до обмеженої модерації;
  - адміністратор має повний доступ до створення, редагування та видалення техніки, керування користувачами, доступу до всіх функцій платформи.
- створення та управління ігровими акаунтами (користувачі можуть створювати ігрові акаунти, прикріплювати їх до свого профілю, переглядати техніку в акаунтах, змінювати її статуси);
- управління технікою (адміністратори мають право створювати, редагувати та видаляти техніку, що доступна для вибору. Звичайні користувачі

можуть додавати доступну техніку до свого акаунта, вказувати ступінь її дослідження та суб’єктивну преференцію – оцінку, що впливає на підбір до команд).

1.6.2 Нефункціональні вимоги

Крім основного функціоналу, платформа повинна відповідати сучасним нефункціональним вимогам:

- безпека застосунку.
  - захист від CSRF-атак та несанкціонованого доступу через реалізацію токенів авторизації (JWT);
  - використання шифрування паролів за допомогою алгоритму bcrypt, що унеможливлює зворотне декодування навіть при витоку даних.
- масштабованість (архітектура системи має передбачати можливість розширення обсягів даних та кількості користувачів без зниження продуктивності, зокрема через контейнеризацію (Docker) та використання MongoDB як бази даних, що підтримує горизонтальне масштабування);
- кросплатформенність (застосунок повинен коректно функціонувати в середовищах різних операційних систем і браузерів, з можливістю інтеграції як з десктопними, так і з мобільними клієнтами в майбутньому).

2 ЗАСОБИ РЕАЛІЗАЦІЇ WEB-ПЛАТФОРМИ ДЛЯ МЕНЕДЖМЕНТУ ІГРОВОЇ СОЦІАЛЬНОЇ СТРУКТУРИ

Розробка сучасного web-застосунку потребує ретельно підібраного технологічного стеку, здатного забезпечити реалізацію як функціональних, так і нефункціональних вимог проєкту. У межах створення серверної частини платформи для управління ігровою соціальною структурою «SOKIL» було використано низку перевірених технологій та інструментів, що сприяли побудові надійної, безпечної та масштабованої архітектури.

Основу застосунку становить фреймворк Spring Boot, який забезпечує гнучкий і швидкий спосіб створення RESTful API. Для реалізації автентифікації та авторизації використано Spring Security у поєднанні з JWT, що дозволило реалізувати розмежування прав доступу відповідно до ролей користувачів. Використання бібліотеки Lombok спростило структуру коду, зменшивши обсяг шаблонних конструкцій.

У якості системи збереження даних обрано MongoDB, що надає переваги документно-орієнтованої моделі при роботі з акаунтами, технікою та гнучкими зв’язками між сутностями. Середовище розробки побудовано в IntelliJ IDEA, а керування версіями коду здійснювалося за допомогою GitHub, що дало змогу ефективно контролювати зміни та підтримувати історію проєкту.

Для тестування API-ендпоінтів використовувався Postman, що забезпечило перевірку коректності запитів на всіх етапах реалізації бізнес-логіки. Завершальним етапом стала підготовка до розгортання, в межах якого проєкт було контейнеризовано за допомогою Docker і задеплоєно на хмарному хостингу Fly.io, що дало змогу перевірити працездатність системи в реальному середовищі.

У наступних підрозділах розглядаються кожен з компонентів стеку, а також їх роль і інтеграція у процес розробки програмного забезпечення для системи «SOKIL».

2.1 Середовище розробки

2.1.1 IntelliJ IDEA

IntelliJ IDEA – це інтегроване середовище розробки (IDE), розроблене компанією JetBrains, що широко використовується для проєктів на Java та Kotlin. У межах реалізації проєкту «SOKIL» використовувалась версія IntelliJ IDEA Ultimate з підтримкою Spring Boot, що значно прискорило розробку завдяки автоматичній конфігурації, інтеграції з Maven та зручному налагодженню.

Серед основних можливостей, що активно використовувалися:

- інтеграція з Git – дозволила керувати репозиторієм безпосередньо з IDE, виконувати коміти, переглядати історію змін та працювати з гілками;
- підтримка Spring Boot – автоматичне сканування конфігурацій, автозапуск застосунку з візуальним контролем середовища, відображення маршрутів REST-контролерів;
- Docker-плагін – хоча Docker не використовувався для локальної розробки, плагін давав можливість переглядати образи, перевіряти конфігурації;
- Lombok Plugin – для коректної підтримки анотацій Lombok (@Data, @Builder, @Getter, @Setter, тощо), що мінімізують шаблонний код;
- Spring Assistant – підсвічування властивостей у application.properties або application.yml, перевірка коректності назв.

2.1.2 GitHub

GitHub виступав основною платформою для зберігання та контролю версій коду проєкту. Для цього було створено окремий репозиторій, у якому зберігався весь код бекенд-частини разом з конфігураційними файлами, Docker-конфігурацією та тестами.

У процесі розробки використовувались такі функції:

- коміти та історія змін – кожна логічно завершена частина функціональності фіксувалась окремим комітом із коротким описом. Це дозволяло

легко відстежувати етапи реалізації та повертатися до попередніх версій при потребі;

- Гілки (branches) – у разі реалізації нових великих підфункцій створювались окремі гілки, які після перевірки об'єднувались з основною main-гілкою;
- Pull Request-и – при потребі, могли використовуватись для обговорення змін перед об’єднанням гілок;
- CI/CD процеси – у проєкті було реалізовано автоматичний деплой на платформу Fly.io. Після кожного злиття змін у гілку main, GitHub Actions автоматично запускав збірку проєкту, створював Docker-образ і розгортав оновлену версію сервера на хостинговій інфраструктурі. Такий підхід забезпечив швидке оновлення застосунку, зниження ризику людських помилок та підвищення надійності деплойменту.

2.1.3 Postman

Postman – це інструмент для тестування REST API, який відіграв важливу роль у процесі налагодження та перевірки коректності роботи ендпоінтів.

На всіх етапах реалізації платформи використовувались такі сценарії:

- реєстрація користувача з валідацією реєстраційного коду;
- авторизація користувача з отриманням JWT-токена;
- створення нового ігрового акаунта та перевірка відображення доступної техніки;
- додавання техніки на акаунт користувачем, перевірка рівня дослідження та вподобань;
- перевірка доступу до захищених маршрутів (за ролями – USER, MODERATOR, ADMIN);
- оновлення та видалення техніки (тільки для адміністратора).

Усі запити були організовані у вигляді Postman Collection, яка включала попередньо налаштовані змінні середовища, JWT-токени, параметри запитів та тіла

JSON. Це дозволяло: швидко запускати набір запитів після змін у коді, забезпечити наочну документацію для API та протестувати сценарії edge-case-ів, пов’язаних із авторизацією, валідацією та правами доступу.

2.2 Бекенд: Spring Boot та супутні модулі

2.2.1 Spring Boot

Spring Boot – це розширення класичного фреймворку Spring, призначене для спрощення процесу створення, тестування та запуску Java-застосунків. Він дозволяє уникнути складної XML-конфігурації, характерної для попередніх версій Spring, та забезпечує автоматичне налаштування компонентів за замовчуванням. Основною метою Spring Boot є забезпечення швидкого старту нового проєкту з готовими шаблонами, одночасно надаючи розробнику повну гнучкість у подальшій кастомізації [21].

Серед ключових переваг Spring Boot – автоматична конфігурація, що дозволяє мінімізувати кількість налаштувань вручну. Завдяки системі starter-ів, розробник може підключити лише ті залежності, які потрібні саме в цьому проєкті, наприклад spring-boot-starter-web, spring-boot-starter-security, spring-boot-starter-data-mongodb. Це спрощує керування залежностями та робить структуру проєкту чистішою.

2.2.2 Spring Security та JWT

Spring Security – це потужний і розширюваний фреймворк для забезпечення безпеки web-застосунків, який є стандартом у екосистемі Spring. Він забезпечує автентифікацію, авторизацію, керування сесіями, захист від атак типу CSRF (Cross-Site Request Forgery), CORS (Cross-Origin Resource Sharing) та інші засоби контролю доступу до ресурсів [22].

Для реалізації безсесійної авторизації у сучасних web-застосунках, зокрема REST API, часто використовується JWT (JSON Web Token). Це стандарт передачі інформації у вигляді токенів, які підписуються з використанням симетричного або

асиметричного ключа. JWT дозволяє клієнту після успішної автентифікації отримати токен, який надалі додається до заголовків HTTP-запитів і забезпечує ідентифікацію користувача без збереження сесії на стороні сервера [23]. Цей процес зображено на рисунку 2.1.

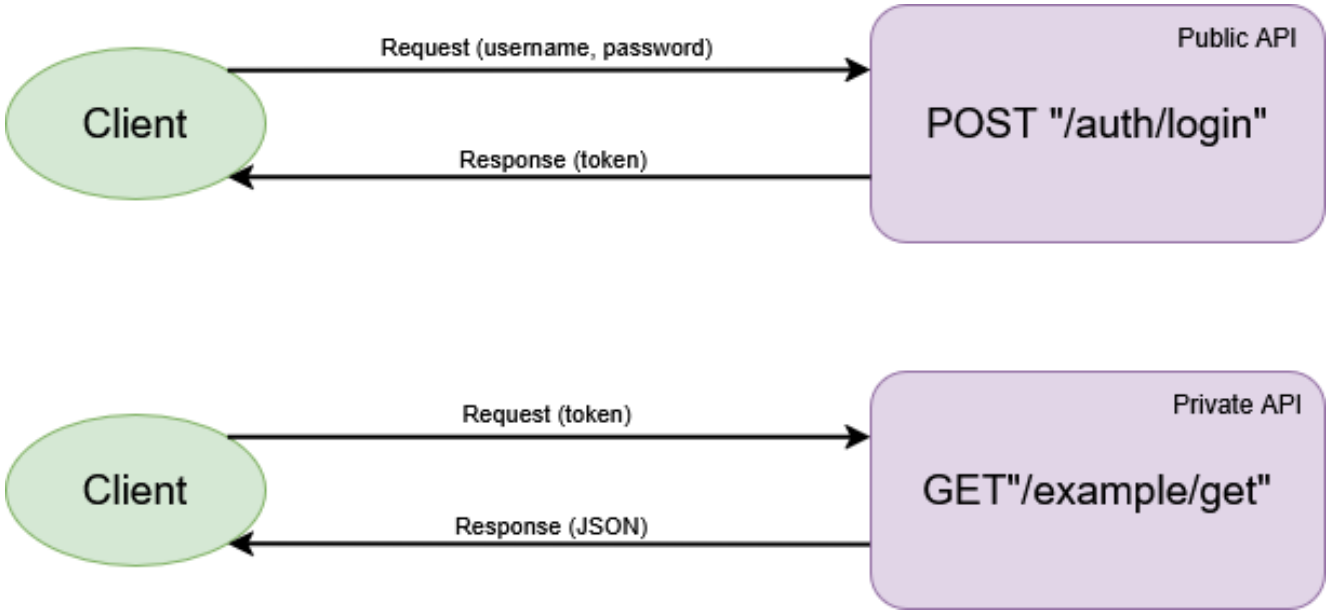


Рис. 2.1 Процес автентифікації з використанням JWT

CORS (Cross-Origin Resource Sharing) підтримується Spring Security через конфігураційні механізми, які дозволяють вказати список довірених джерел, методів і заголовків для запитів між доменами. CSRF-захист, своєю чергою, є актуальним переважно для браузерних застосунків із сесіями. У випадку використання JWT у безсесійній архітектурі його, як правило, вимикають відповідно до рекомендацій для RESTful API.

Таким чином, Spring Security у поєднанні з JWT забезпечує високий рівень контролю доступу, безпечне управління правами користувачів і гнучку інтеграцію з ролями та політиками авторизації.

2.2.3 Розмежування прав доступу

Механізм розмежування доступу є одним із ключових аспектів забезпечення безпеки в багатокористувацьких системах. У контексті Java-платформи на базі

Spring Security цей механізм реалізується через налаштування фільтрів безпеки, що дозволяють обмежити доступ до ресурсів залежно від ролі користувача.

У типових сценаріях контролю доступу використовуються методи authorizeHttpRequests() у конфігураціях безпеки, які дозволяють задати правила доступу до конкретних маршрутів залежно від авторитетів (Authorities), переданих у JWT-токені. Для цього можуть використовуватись методи hasAuthority() або hasAnyAuthority(...), які перевіряють наявність потрібної ролі.

Користувачам можуть бути призначені різні ролі (USER, MODERATOR, ADMIN), кожна з яких має власний рівень доступу до функціоналу. Такий підхід відповідає принципу мінімальних привілеїв і дозволяє гнучко налаштовувати безпеку API на основі політик.

Spring Security також дозволяє використовувати анотації для захисту методів (@PreAuthorize, @Secured), однак у сучасних конфігураціях з JWT більш поширеним є підхід, коли всі перевірки здійснюються на рівні фільтрації HTTP-запитів.

Механізм авторизації у Spring Security побудований таким чином, що може бути легко інтегрований із будь-якою роллю, схемою доступу чи зовнішнім джерелом ідентифікації, що робить його універсальним інструментом для побудови захищених web-сервісів.

2.2.4 Lombok

Lombok – це бібліотека для Java, яка дозволяє автоматично генерувати рутинний код, використовуючи прості анотації [24]. У проєкті «SOKIL» Lombok значно спростив розробку сутностей, DTO та відповідних конфігураційних класів.

Наприклад, замість ручного написання гетерів, сетерів, конструкторів і методу toString() можна просто використати:

- @Getter, @Setter для автоматичного створення гетерів і сетерів для всіх полів;
- @Data включає @Getter, @Setter, @EqualsAndHashCode, @ToString, а також конструктор;

- @NoArgsConstructor, @AllArgsConstructor – генерація конструкторів без/з усіма аргументами;
- @Builder – реалізація шаблону Builder для зручного створення об’єктів;
- @RequiredArgsConstructor використовувався для ініціалізації бінів в Spring.

Завдяки Lombok кількість шаблонного коду в проєкті зменшилася в рази, що зробило класи більш компактними, легкими для читання й обслуговування. Крім того, це позитивно вплинуло на швидкість розробки та зменшення помилок, пов’язаних із ручним дублюванням коду.

2.3 MongoDB

Для реалізації зберігання даних у проєкті «SOKIL» було обрано базу даних MongoDB, яка є однією з найпопулярніших систем NoSQL-класу. На відміну від реляційних баз даних, MongoDB використовує документно-орієнтований підхід, у якому всі дані зберігаються у вигляді JSON-подібних документів (у форматі BSON), об’єднаних у колекції. Такий підхід є ідеальним для зберігання складних, вкладених та гнучко структурованих об’єктів, що часто змінюються у процесі розробки або мають неоднорідну природу.

MongoDB було обрано з кількох причин:

- структура користувацьких акаунтів, техніки, списків – нефіксована та змінна, тому її зручно моделювати в схемі без жорсткого визначення таблиць;
- висока продуктивність читання та запису для роботи з колекціями;
- можливість горизонтального масштабування через шардинг;
- підтримка вкладених документів, що дозволяє уникати складних JOIN-операцій, властивих SQL.

Також MongoDB має чудову інтеграцію з екосистемою Spring через Spring Data MongoDB, що дозволяє автоматично створювати запити через інтерфейси репозиторіїв і працювати з класами.

2.3.1 Сутності

Архітектура бази даних web-платформи побудована відповідно до документно-орієнтованої моделі MongoDB. У системі реалізовано низку сутностей, що відображають об'єкти предметної області та логічні зв’язки між ними. Кожна з таких сутностей представлена окремим типом документа, який зберігається в відповідній колекції. Структури документів є вкладеними та можуть містити як прості поля, так і масиви об'єктів, що забезпечує гнучкість у поданні ієрархічної інформації.

Завдяки особливостям MongoDB, модель зберігання дозволяє репрезентувати складні залежності між об’єктами, зокрема багаторівневі вкладення та взаємопов’язані елементи. Це зручно для відображення, наприклад, ієрархій користувачів, пов’язаних облікових записів або наборів характеристик, які можуть змінюватися залежно від ролі чи функції в системі.

Приклад структури документа User у форматі MongoDB:


```
{
  "_id": {"$oid": "66310a8face029131efab138" },
  "discordUsername": "onikuma",
  "discordSokilNickname": "_harpestes_ (Андрій)",
  "email": "…",
  "password": "…",
  "permissions": ["USER", "ADMIN"],
  "status": "ACTIVE",
  "mainGameAccount": {"$oid": "66aa46da7748f0460c7ad641"},
  "hasAccessTo": [{"$oid": "66aa46da7748f0460c7ad641"}],
  "vehicleList": [{
     "vehicle": {"$oid": "669e6ec73fe1c928c9db8865"},
     "personalRate": 2 }],
  "preferredPosition": ["FIGHTER"],
  "name": "Андрій",
  "phoneNumber": "…",
```

```
  "city": "Київ",
  "country": "Україна",
  "discordRoles": [],
  "_class": "org.sokil.squadronwebsite.entity.User"
}
```


2.3.2 Агрегації та зв’язки

У межах побудови логічної моделі даних у документно-орієнтованій базі виникає потреба у реалізації зв’язків між об'єктами, які мають взаємозалежну або багаторівневу структуру. Однією з архітектурних задач є забезпечення ефективного представлення таких зв’язків без створення зайвих залежностей або дублювання інформації, що може призводити до ускладнення обробки та втрати цілісності даних.

Замість безпосереднього вкладення пов’язаних документів одне в одного, система використовує підхід розділення зв’язаних об’єктів з подальшим зберіганням ідентифікаторів посилання. Це дозволяє уникати циклічних залежностей та зберігати гнучкість при зміні структури окремих компонентів. Завдяки такому рішенню, кожен документ залишається атомарним, а дані оновлюються лише в одному місці.

Для забезпечення доступу до пов’язаних даних використовуються агрегаційні запити, які дозволяють об'єднувати інформацію з кількох колекцій або вкладених структур. Такий підхід підтримується через механізм Aggregation Pipeline, що дозволяє поетапно фільтрувати, трансформувати та структурувати дані у відповідності до логіки застосунку.

Застосування агрегацій є особливо ефективним при роботі з вкладеними масивами, умовною фільтрацією та динамічними запитами, які залежать від багатьох змінних. Це дає змогу гнучко формувати вихідні результати без необхідності дублювати інформацію або створювати надлишкові структури, що у свою чергу сприяє масштабованості та ефективності системи загалом.

2.4 Docker та Fly.io

Docker – це платформа контейнеризації, яка дозволяє упаковувати програмне забезпечення та його залежності у стандартизовані ізольовані середовища, звані контейнерами. На відміну від класичної віртуалізації, Docker забезпечує легковагову інфраструктуру, у якій кожен контейнер працює безпосередньо на операційній системі хост-машини, що значно знижує накладні витрати та підвищує продуктивність [25].

У межах реалізації проєкту web-платформи “SOKIL” Docker використовувався не як середовище для локальної розробки, а як інструмент підготовки до розгортання застосунку в хмарі, зокрема на платформі Fly.io. Контейнеризація дозволила запакувати серверну частину застосунку, реалізовану на Spring Boot, разом з усіма конфігураційними файлами, бібліотеками та залежностями у цілісний Docker-образ.

Основними перевагами такого підходу стали ізольованість середовища (на сервері виконання запускається саме та конфігурація, яка була протестована й зібрана локально) та відсутність залежності від системи хостингу (розгортання можливе як на локальних машинах, так і в хмарі без зміни внутрішньої структури проєкту).

Для реалізації контейнеризації було створено файл Dockerfile, у якому описано етапи побудови образу: використання базового образу OpenJDK, копіювання зібраного .jar-файлу застосунку, визначення порту для HTTP-запитів, а також команда запуску.

Для деплойменту використовувалась платформа Fly.io, яка підтримує Docker-образи без потреби у ручному конфігуруванні віртуальних машин. Після авторизації в Fly CLI було виконано розгортання за допомогою команд, які автоматично створюють інстанс, виставляють порти та забезпечують доступність сервісу в мережі.

3 ПРОЄКТУВАННЯ WEB-ПЛАТФОРМИ ДЛЯ МЕНЕДЖМЕНТУ ІГРОВОЇ СОЦІАЛЬНОЇ СТРУКТУРИ “SOKIL”

Проєктування програмного забезпечення є критичною стадією розробки, яка визначає організацію системи, принципи її функціонування та взаємодії компонентів. Формування архітектури web-платформи для ігрової спільноти “SOKIL” передбачає узгодження технічних рішень із вимогами до функціональності, безпеки, масштабованості та підтримуваності. У контексті реалізації серверної частини системи для менеджменту ігрової спільноти важливо забезпечити стабільність роботи сервісу, підтримку розмежування прав доступу, зручність у роботі з динамічною структурою даних і здатність до адаптації в умовах змін.

Побудова внутрішньої структури платформи відбувалася з урахуванням особливостей обраного стеку технологій. Компонентна організація застосунку, поділ відповідальності між логічними модулями, вибір шаблонів взаємодії між ними базуються на практиках побудови розширюваних web-систем. Обрана модель проєктування підтримує незалежність шарів, спрощує тестування, забезпечує чіткий розподіл функціональності та дозволяє масштабувати систему без конфлікту залежностей.

3.1 Архітектурні принципи

3.1.1 Архітектура системи

У реалізації серверної частини web-платформи було застосовано архітектурний стиль, що базується на багатошаровій архітектурі, зокрема її класичному варіанті – трьохшаровій моделі. Цей підхід передбачає чіткий поділ системи на незалежні функціональні рівні, кожен з яких відповідає за свою окрему логіку та взаємодіє лише з безпосередньо сусіднім шаром. Така структура дозволяє

побудувати систему, що легко масштабуються, тестується, підтримується та розширюється [26].

У межах проєкту виокремлено такі шари:

- шар представлення (Presentation Layer) – представлений REST-контролерами, які відповідають за взаємодію з клієнтом через HTTP-запити. Контролери приймають вхідні дані, ініціюють відповідні сервіси та повертають результат у вигляді JSON-відповідей. На цьому рівні реалізуються також елементи валідації запитів та серіалізація даних;
- шар логіки застосунку (Application/Business Logic Layer) – реалізований у вигляді сервісних класів. Саме на цьому рівні визначається поведінка системи відповідно до бізнес-вимог, наприклад, створення ігрових акаунтів, додавання техніки, перевірка прав доступу тощо. Сервіси використовують інтерфейси репозиторіїв для доступу до даних і забезпечують обробку та трансформацію інформації для клієнта;
- шар доступу до даних (Data Access Layer) – відповідає за взаємодію з базою даних MongoDB. У ньому використовуються інтерфейси Spring Data MongoDB (MongoRepository), які дозволяють здійснювати CRUD-операції без потреби писати Mongo-запити вручну. Цей рівень ізольований від бізнес-логіки, що дозволяє гнучко змінювати модель зберігання без зміни логіки застосунку.

Відповідно до принципів [26], така структура дозволяє:

- інкапсулювати зміни в межах конкретного шару, не впливаючи на решту системи;
- забезпечити повторне використання компонентів, зокрема сервісів або об’єктів DTO;
- полегшити тестування, оскільки кожен шар може бути протестований окремо з використанням моків;
- забезпечити масштабованість, оскільки шари можуть розгортатися окремо (наприклад, у мікросервісній архітектурі або при вертикальному поділі).

Ще однією перевагою трьохшарового підходу є підтримка ієрархічного управління залежностями: залежності завжди спрямовані вниз, що відповідає принципу інверсії залежностей. У системі «SOKIL» жоден контролер не взаємодіє безпосередньо з репозиторіями, а виключно через сервіси, що забезпечує чіткий контроль за потоками даних та централізацію логіки. Наочно цей підхід демонструє діаграма архітектури web-платформи, зображена на рисунку 3.1.

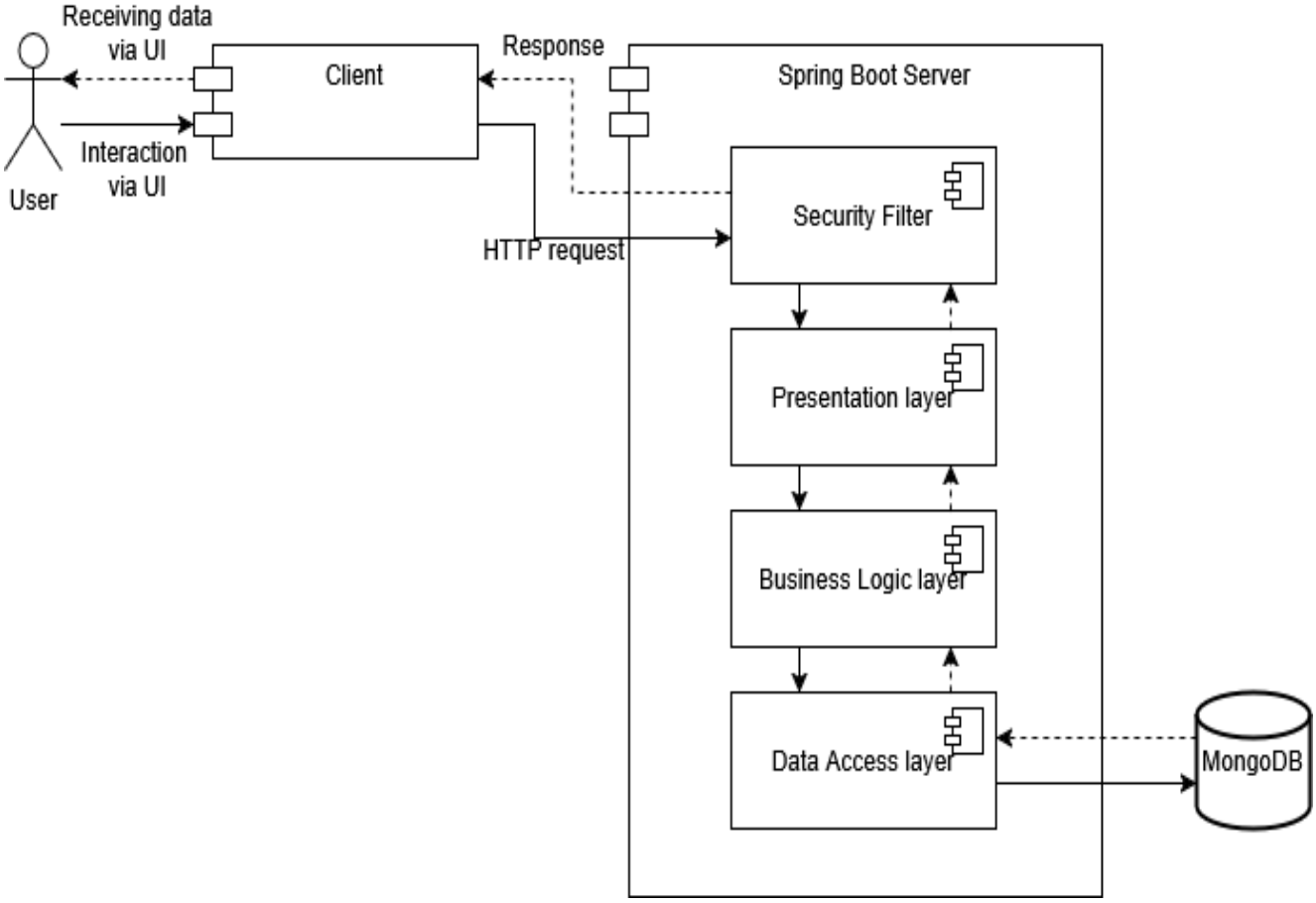


Рис. 3.1 Архітектура web-платформи “SOKIL”

Застосування трьохшарової архітектури дозволяє ефективно масштабувати платформу, адаптувати її до зміни бізнес-вимог, зменшити ризики при модифікації компонентів і підвищити загальну якість коду. Саме тому такий стиль є стандартним підходом при створенні корпоративних та ігрових бекенд-систем.

3.1.2 Дотримання принципів SOLID

Для підтримки структурованості, надійності й еволюційної гнучкості програмного коду в системі застосовано принципи, об'єднані у концепцію SOLID – набір з п’яти загальновизнаних принципів об'єктно-орієнтованого програмування, сформульованих Робертом Мартіном. Ці принципи допомагають уникати надмірної зв’язаності компонентів, сприяють побудові легко масштабованої архітектури та забезпечують підтримуваність проєкту на всіх етапах його життєвого циклу.

Перші літери назв цих принципів і формують назву SOLID, а саме:

- принцип єдиної відповідальності (Single Responsibility Principle; SRP) – визначає, що певний модуль має бути відповідальний за одне завдання, або ж модуль має мати лише одну причину для його зміни;
- принцип відкритості/закритості (Open-Closed Principle; OCP) – визначає, що сутності мають бути відкриті до розширення та закриті для змін;
- принцип підстановки Лісков (Liskov Substitution Principle; LSP) – визначає, що екземпляри батьківського класу мають мати змогу замінюватися на дочірні класи без внесення змін в програму;
- принцип розподілу інтерфейсів (Interface Segregation Principle; ISP) – визначає, що актори не мають бути залежними від методів, що вони не використовують;
- принцип інверсії залежності (Dependency Inversion Principle; DIP) – визначає, що високорівневі компоненти програм не мають бути залежними від низькорівневих [27].

Дотримуючись цих принципів, можливе створення великих складних систем, які не будуть викликати труднощі в розширенні та підтримці, а їх недотримання призводить до недокументованих, незапланованих та випадкових рішень, що руйнує підтримуваність, а з нею і системи повністю через недбалість розробників [27]. Наочно переваги використання SOLID може продемонструвати діаграма витрат часу на зміни на рисунку 3.2.

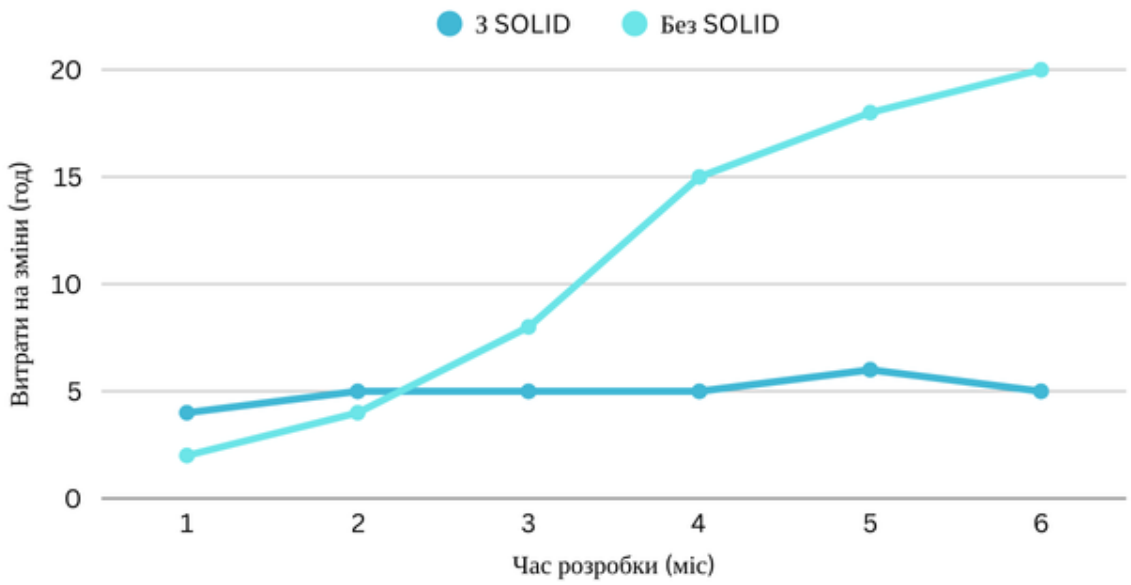


Рис. 3.2 Діаграма витрат часу на зміни в програмному коді

3.1.3 Розширюваність і підтримуваність

Гнучкість системи реалізується через використання інтерфейсів, DTO-класів та конфігураційних компонентів. DTO використовуються для переносу даних між шарами, абстрагуючи внутрішні структури від зовнішнього API, що дозволяє незалежно змінювати моделі зберігання й логіку обробки без впливу на клієнтські застосунки.

Конфігурація безпеки, CORS, авторизації та шифрування паролів винесена в окремі класи, що забезпечує централізоване управління змінами. Це спрощує адаптацію під зміну вимог без потреби перегляду всього проєкту.

Архітектура системи дозволяє легко адаптувати платформу до змін у внутрішніх процесах ігрової спільноти. Наприклад, у разі появи нових категорій техніки, змін у правилах розподілу ролей чи потреби в нових формах статистики, систему можна розширити, не порушуючи існуючу логіку та з мінімальним впливом на користувачів.

3.2 Структура модулів

3.2.1 Опис логічних компонентів

Архітектура застосунку розділена на низку логічно незалежних модулів, кожен з яких виконує окрему роль у системі. Такий підхід дозволяє досягти високої модульності, ізольованості функціональності та можливості повторного використання компонентів у різних частинах проєкту.

Контролери відповідають за прийом і обробку HTTP-запитів. Вони виконують роль шлюзу між зовнішнім клієнтом і внутрішньою логікою застосунку, здійснюють початкову валідацію вхідних даних і делегують виконання операцій відповідним сервісам.

Сервіси реалізують бізнес-логіку системи. Вони координують обробку запитів, здійснюють взаємодію з репозиторіями, обробляють помилки, виконують трансформацію даних. Сервіси не мають залежності від представлення (контролерів) і працюють через інтерфейси, що полегшує їх тестування.

DTO використовуються для перенесення даних між шарами системи. DTO забезпечують відокремлення внутрішніх моделей даних від API, мінімізуючи ризик витоку зайвої інформації та забезпечуючи адаптацію структури відповіді до потреб клієнта.

Моделі (Entity/Document-класи) представляють об'єкти збереження, що відповідають схемам документів у базі даних. Вони містять поля, які підлягають персистенції, а також анотації, необхідні для взаємодії з MongoDB (наприклад, @Document, @Id, індекси, вкладені об’єкти).

Репозиторії – абстрактний шар для доступу до бази даних. Реалізуються як інтерфейси, які наслідують MongoRepository або інші репозиторні контракти Spring Data. Вони дозволяють виконувати CRUD-операції, пошук за критеріями, а також підтримують побудову запитів за допомогою імен методів або @Query.

Поділ на ці модулі забезпечує чітке розділення відповідальностей та дає змогу масштабувати систему, розширювати її логіку без порушення існуючих компонентів.

3.2.2 Залежності від сторонніх бібліотек

Для реалізації основної функціональності та спрощення розробки було використано низку сторонніх бібліотек і фреймворків, які інтегруються в систему через Maven.

Spring Boot – основа застосунку, що забезпечує автоматичну конфігурацію, запуск вбудованого web-сервера, REST API, залежності між модулями.

Spring Security – використовується для реалізації авторизації та автентифікації. Забезпечує механізми контролю доступу до ендпоінтів на основі ролей, а також базові засоби шифрування.

JSON Web Token (JWT) – механізм безсесійної авторизації. Використовується в поєднанні з Spring Security для підтвердження особи користувача та передачі ролей у запитах.

Spring Data MongoDB – надає інтеграцію з базою даних MongoDB. Дозволяє працювати з колекціями як зі звичайними Java-об’єктами, використовуючи репозиторії та автоматичне формування запитів.

Lombok – бібліотека для скорочення шаблонного коду. Забезпечує автоматичну генерацію гетерів, сетерів, конструкторів, equals(), hashCode(), toString(), а також реалізацію шаблону "будівельник".

Кожна з цих бібліотек виконує свою чітку функцію в межах відповідного модуля, не порушуючи загальної ізольованості компонентів. Їх інтеграція сприяє дотриманню принципів SOLID, підвищує продуктивність розробки та забезпечує стабільність роботи системи.

3.3 Моделювання функціональності

3.3.1 Варіанти використання

Функціональність web-платформи для менеджменту ігрової соціальної структури реалізується через взаємодію кількох типів користувачів із системою. Основними акторами в контексті варіантів використання є:

User – базовий користувач, який може реєструватися або входити до системи, керувати власними акаунтами та технікою, створювати команди, переглядати статистику та чорний список;

Admin – адміністратор із розширеними повноваженнями для модерації чорного списку, додавання/редагування техніки, перегляду всіх акаунтів і користувачів;

System – внутрішній технічний актор, який відповідає за обробку JWT-токенів, валідацію та безпечну авторизацію користувачів.

На діаграмі варіантів використання на рисунку 3.3 зображено основні дії, доступні для кожного актора, з урахуванням принципу розмежування доступу. Варіанти використання згруповані за рівнем доступу до ресурсу.

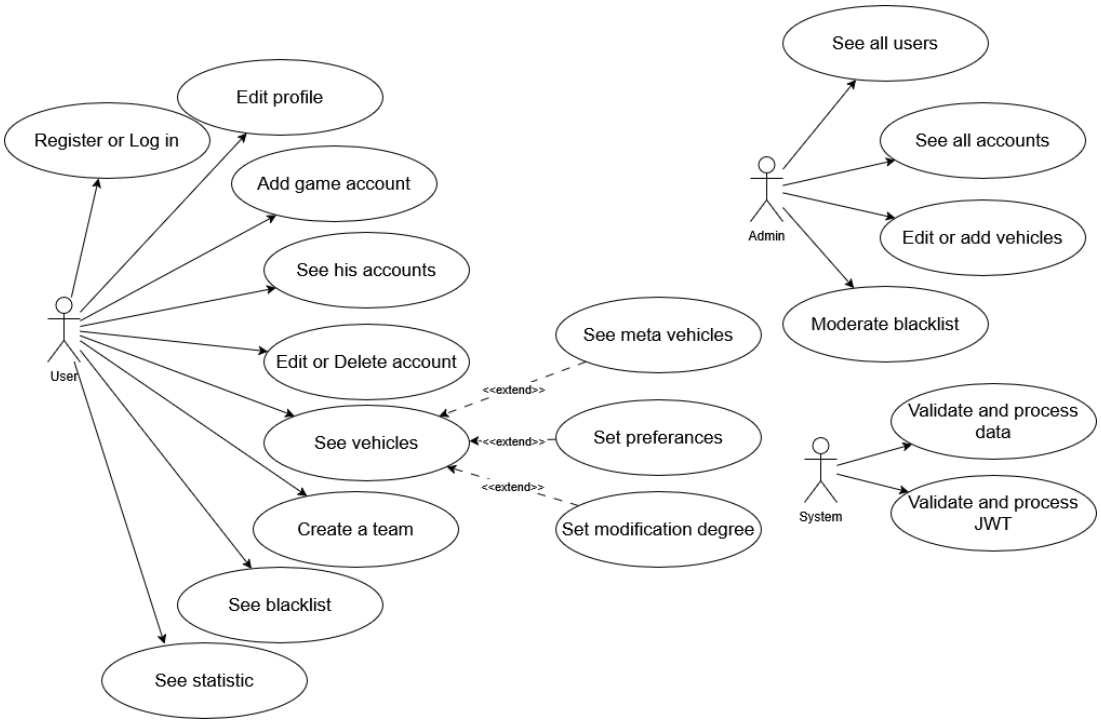


Рис. 3.3 Діаграма варіантів використання

3.3.2 Сценарій взаємодії з системою

Один із поширених сценаріїв взаємодії – користувач оновлює ступінь персональної вподобаності техніки, що дозволяє позначити ті одиниці, які гравець вважає найбільш підходящими до поточного сезону чи командної стратегії.

Користувач надсилає HTTP-запит з новими значеннями вподобань (наприклад, для певних ID техніки). Контролер отримує запит та передає його до сервісу. Сервіс завантажує поточного користувача, знаходить відповідну техніку у списку або створює новий об’єкт техніки, оновлює її персональну оцінку та зберігає оновлений об’єкт користувача. Діаграма цього процесу зображена на рисунку 3.4.

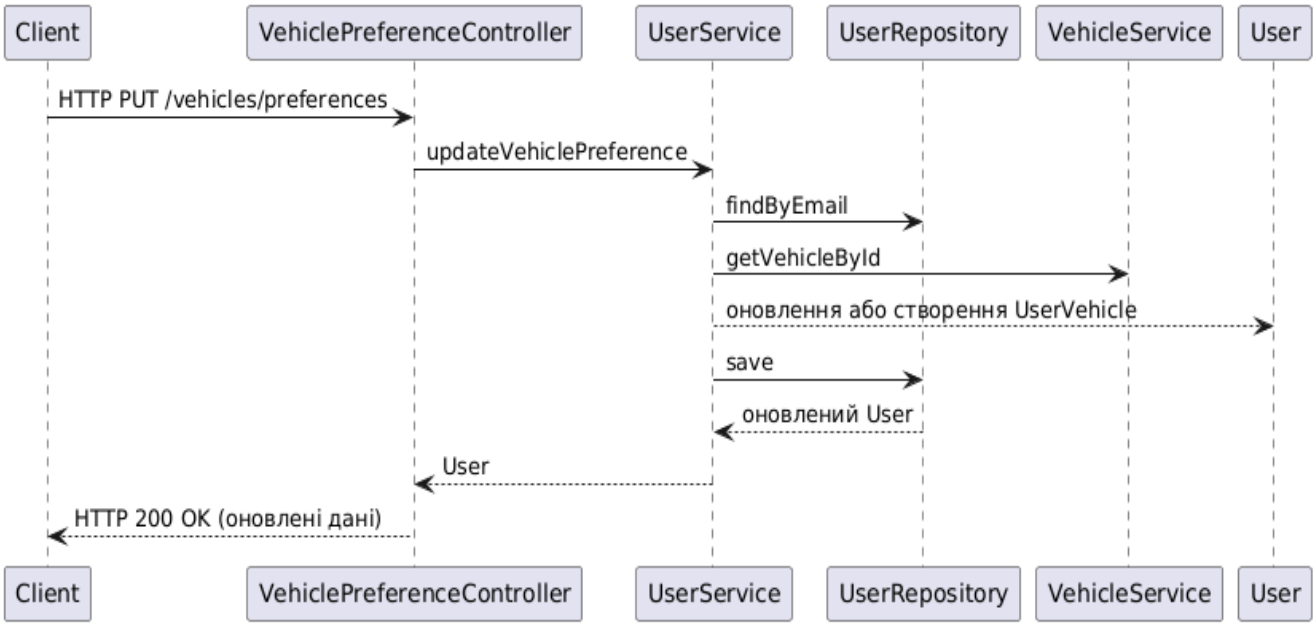


Рис. 3.4 Діаграма послідовності оновлення ступені персональної вподобаності техніки

3.4 Проєктування інформаційної моделі

3.4.1 Структура основних об’єктів

Інформаційна модель платформи базується на структурі предметної області, у якій ключовими елементами є користувачі, ігрові акаунти, техніка а також записи чорного списку. Кожна з цих сутностей відображає окрему логічну одиницю системи та має власну поведінку й набір атрибутів.

Користувач (User) – центральна сутність, яка репрезентує зареєстрованого учасника платформи. Має атрибути ідентифікації (email, пароль), роль доступу, а також пов’язаний перелік ігрових акаунтів, персональних налаштувань техніки та інші параметри.

Ігровий акаунт (GameAccount) – об'єкт, що належить користувачу та відповідає одному з його акаунтів у грі. Містить інформацію про платформу, нікнейм, перелік техніки, яка доступна на цьому акаунті.

Техніка (Vehicle) – об'єкт, що представляє одиницю військової техніки. Має властивості, такі як назва, тип, нація, рейтинг бою тощо. Один і той самий запис техніки може бути доступний на кількох акаунтах і мати різні ступені дослідження та вподобання залежно від користувача.

Чорний список (BlacklistUser) – запис, який фіксує обмеження або блокування певних користувачів.

Між сутностями встановлено логічні зв’язки:

- один користувач може мати декілька акаунтів (1:N);
- один акаунт може мати декілька одиниць техніки, причому техніка є загальною для всієї системи, а її параметри – специфічні для кожного користувача або акаунта (N:M);
- одна техніка може бути пов’язана з багатьма користувачами, але персоналізовані характеристики зберігаються у вкладеній структурі;
- команди можуть включати кілька акаунтів (N:M).

Ця логіка була покладена в основу формування схеми бази даних, що забезпечує гнучку адаптацію під потреби ігрової спільноти, збереження персональних даних техніки та масштабованість структури. Схему бази даних наведено на рисунку 3.5.

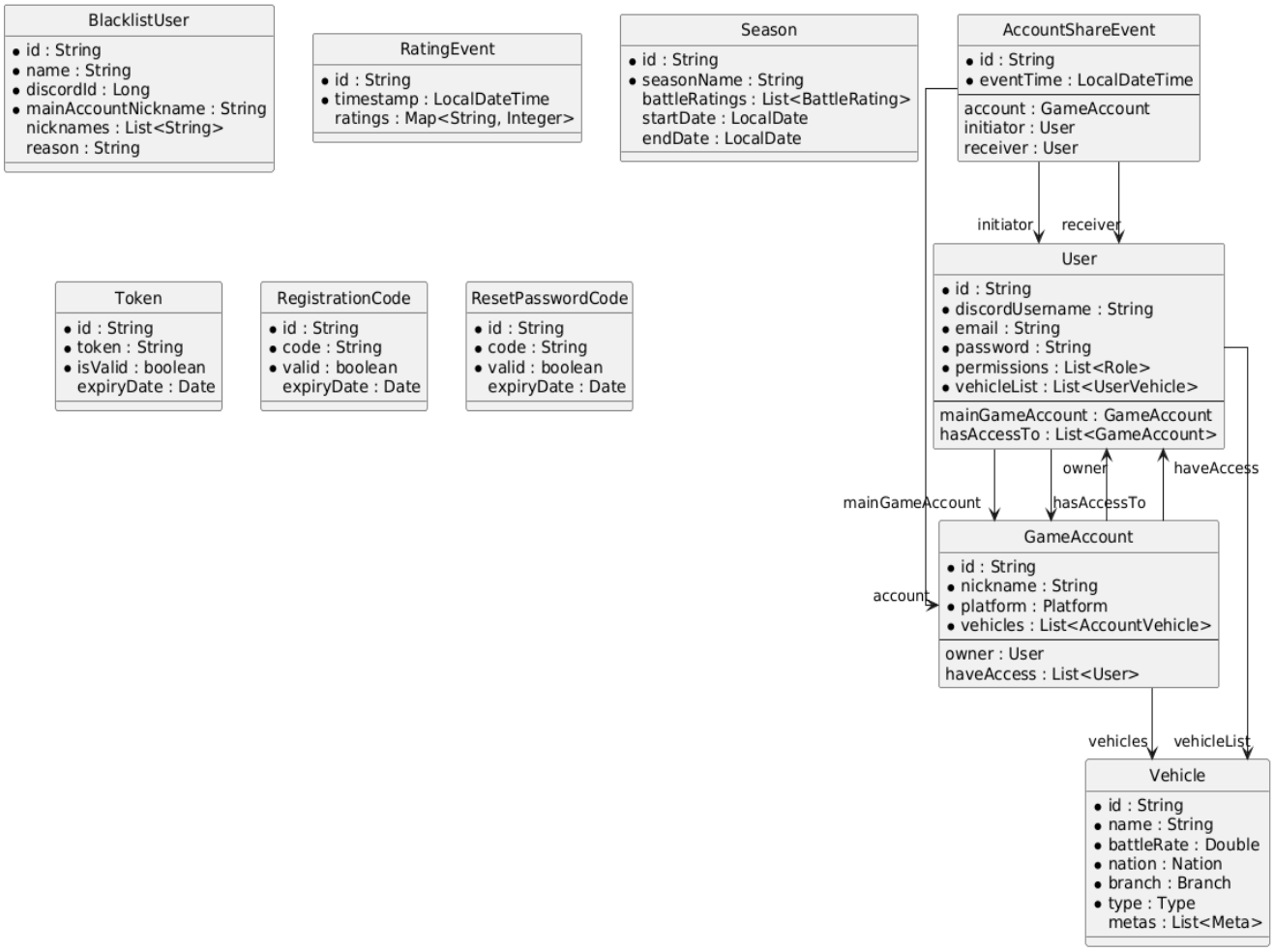


Рис. 3.5 Схема бази даних

3.4.2 Модель зберігання в MongoDB

З огляду на вибір MongoDB як бази даних, структура зберігання інформації у web-платформі побудована відповідно до принципів документно-орієнтованої моделі. Усі об’єкти системи зберігаються у вигляді BSON-документів у відповідних колекціях. Документи є самодостатніми одиницями, що можуть містити як прості поля, так і вкладені структури, масиви, мапи та посилання на інші документи.

Для моделювання взаємозв’язків між сутностями застосовуються два ключові підходи – вбудовані структури (денормалізація) та посилання через @DocumentReference (референційна нормалізація). Така комбінована стратегія

дозволяє зберегти баланс між продуктивністю, гнучкістю моделі та простотою запитів.

Ключові особливості зберігання:

- користувачі (Users) зберігаються як окремі документи, що містять вкладені об’єкти (наприклад, список обраної техніки vehicleList або список уподобаних позицій). Водночас взаємозв’язки з акаунтами (mainGameAccount, hasAccessTo) реалізуються через посилання, що дозволяє уникнути дублювання та забезпечити гнучкість при оновленнях;
- ігрові акаунти (Accounts) також представлені окремими документами. Вони містять дані про ігрову активність, історію рейтингу, список прив’язаних користувачів та список техніки. Список техніки реалізований у вигляді масиву вбудованих об’єктів AccountVehicle, що включають посилання на глобальну техніку (Vehicles) і ступінь модифікації;
- техніка (Vehicles) зберігається як незалежна сутність з усіма параметрами. До неї посилаються як акаунти, так і користувачі (через AccountVehicle або UserVehicle відповідно), що дає змогу централізовано керувати змінами характеристик техніки без потреби в масових оновленнях;
- події передання акаунтів (AccountShareEvent) містять посилання на акаунт і користувачів-учасників (ініціатора та отримувача). Це дозволяє вести історію взаємодій без зміни основної моделі Account;
- події зміни рейтингу (RatingEvent) реалізуються у вигляді окремих документів, які зберігають дату та мапу рейтингових значень. Це дозволяє зручно агрегувати або візуалізувати історію змін без дублювання об’єктів акаунтів;
- сезони (Seasons) зберігають дані про періоди активності, бойові рейтинги та часові межі. Вони не мають жорстких зовнішніх зв’язків, але використовуються як база для відбору мета-техніки;
- коди (Codes, ResetPasswordCodes) зберігаються в окремих колекціях, забезпечуючи обробку одноразових токенів, наприклад, при реєстрації або скиданні пароля;

- токени (Tokens), що використовуються для JWT-автентифікації, зберігаються з мітками валідності та терміном дії, що дозволяє реалізувати механізм відкликання токенів;
- чорний список (BlacklistUsers) зберігає інформацію про заблокованих користувачів або акаунти з деталізацією причин, історією нікнеймів, Discord ID та основним акаунтом. Оскільки ці записи не є активними суб'єктами системи, вони реалізовані як автономні документи без зовнішніх посилань.

Денормалізація використовується у випадках, коли вкладені структури тісно пов'язані з документом-носієм (наприклад, vehicleList в User, vehicles в Account). Це дозволяє швидко отримати повний набір пов'язаних даних одним запитом без складних агрегацій.

Нормалізація з посиланнями застосовується там, де сутності мають самостійне значення, використовуються у кількох контекстах і можуть часто змінюватися (наприклад, Vehicle, User, Account). Завдяки використанню @DocumentReference ці сутності зберігаються окремо та зв'язуються через ID.

Обрана модель зберігання у MongoDB забезпечує гнучке, масштабоване та логічно чисте представлення ігрової соціальної структури. Поєднання вбудованих об'єктів і референційних зв'язків дозволяє ефективно балансувати між швидкістю доступу до даних та нормалізацією, зберігаючи цілісність, простоту обробки і гнучкість адаптації до майбутніх змін.

3.5 Проєктування безпеки

Архітектура безпеки web-платформи передбачає багаторівневу модель захисту, побудовану на основі сучасних практик у сфері розробки RESTful-систем. Основною метою є забезпечення цілісності, конфіденційності та контрольованості доступу до ресурсів системи.

Безпековий модуль інтегрується у загальну архітектуру платформи як окремий компонент, що обслуговує всі запити на вхідному рівні. Його структура

включає механізми автентифікації, авторизації, захисту від несанкціонованого доступу, а також підтримку базових принципів безпечного програмування.

3.5.1 Модель автентифікації

У системі для реалізаціїї була обрана безсесійна модель автентифікації, що ґрунтується на токенах. Кожен користувач після підтвердження особи отримує цифровий маркер, який надалі використовується для доступу до ресурсів. Такий підхід дозволяє повністю відмовитись від зберігання сесій на сервері, забезпечити масштабованість та просту інтеграцію з клієнтськими застосунками.

Захист від CSRF враховується при виборі моделі автентифікації. У системах з токен-автентифікацією такі загрози частково нейтралізуються відсутністю автоматичної передачі облікових даних у запитах браузера.

Верифікація токенів здійснюється на початковому етапі обробки запиту, до передавання управління основному застосунку. Таким чином, система фільтрує неавторизовані запити та ізолює доступ до функціоналу. На рисунку 3.6 представлено архітектуру підсистеми безпеки.

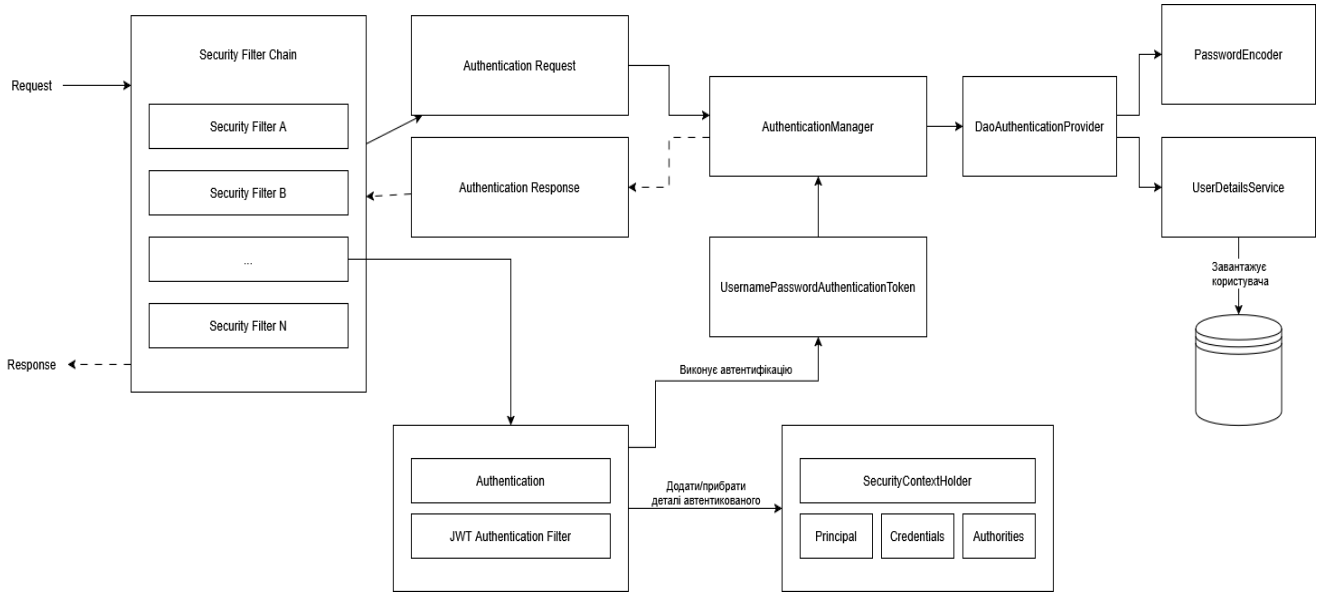


Рис. 3.6 Архітектура підсистеми безпеки

3.5.2 Захист даних

Для захисту персональної інформації було обрано такі базові засоби:

- шифрування чутливих даних, зокрема паролів, із використанням криптографічно стійкого алгоритму bcrypt;
- фільтрація запитів за джерелом походження (CORS), що забезпечує контрольовану міждоменну взаємодію;
- обмеження HTTP-методів і маршрутів на основі типу користувача та метаданих запиту;
- ізоляція критично важливих маршрутів і захист адміністративного функціоналу через окремі політики.

3.5.3 Принцип мінімальних привілеїв

Фундаментальним підходом до організації доступу є дотримання принципу мінімальних привілеїв. Це означає, що кожному користувачеві надається лише той обсяг прав, який необхідний для виконання конкретних завдань. Такий підхід дозволяє мінімізувати наслідки у випадку скомпрометованого облікового запису та підвищує загальний рівень стійкості системи до атак.

Розмежування доступу до функціоналу враховує тип ролі, тип операції (читання, запис, оновлення), а також рівень критичності об’єкта.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ WEB-ПЛАТФОРМИ ДЛЯ МЕНЕДЖМЕНТУ ІГРОВОЇ СОЦІАЛЬНОЇ СТРУКТУРИ “SOKIL”

Розробка серверної частини web-платформи передбачала послідовну реалізацію функціональних модулів відповідно до поставлених вимог. Усі компоненти були створені з урахуванням сучасних підходів до архітектури web-застосунків, дотриманням принципів модульності, розширюваності та безпеки.

Особливу увагу було приділено побудові чіткої логічної структури коду, що дозволяє ізолювати бізнес-логіку від зовнішніх шарів системи, забезпечити прозору взаємодію між сервісами та зменшити зв’язність між модулями. У результаті реалізовано систему, яка є стійкою до змін, легко тестується та масштабується без потреби суттєвого рефакторингу.

Розробка відбувалася ітераційно, з поетапною перевіркою функціональності, що дозволило своєчасно виявляти і усувати недоліки. Для забезпечення надійності результатів були застосовані різні рівні тестування, а також засоби автоматизації розгортання і перевірки роботи платформи в цільовому середовищі.

4.1 Структура проєкту

Побудова структури коду в платформі “SOKIL” ґрунтується на принципах розділення відповідальностей і трирівневої архітектури. У проєкті не використовуються окремі модулі, однак логічна організація забезпечується поділом коду на пакети, кожен з яких відповідає за окремий аспект функціонування системи.

Пакетна структура дотримується таких загальних підходів:

- controller – обробка HTTP-запитів, взаємодія з користувачем через REST API. У цьому пакеті розміщено класи-контролери, які приймають запити, проводять базову валідацію та делегують бізнес-логіку відповідним сервісам;

- service – реалізація бізнес-логіки. Класи сервісів виконують основні дії з даними, координують взаємодію між репозиторіями, зовнішніми джерелами та утилітами, а також відповідають за виконання операцій згідно з правилами доменної моделі;
- repository – доступ до бази даних MongoDB. Репозиторії побудовані на основі інтерфейсів Spring Data MongoDB і відповідають за виконання CRUD-операцій, пошук за параметрами, а також роботу з агрегованими запитами;
- entity – опис структур даних, які зберігаються в базі даних. Тут розташовано всі класи-документи, а також вкладені об’єкти, що використовуються як вбудовані структури або посилання на інші документи за допомогою анотацій (@Document, @DocumentReference, @Id, тощо);
- dto – об’єкти передавання даних, які використовуються для формування запитів і відповідей у контролерах. DTO дозволяють розмежувати внутрішню структуру даних та формат API, підвищуючи безпеку та гнучкість інтерфейсу;
- security – конфігурація механізмів автентифікації та авторизації, реалізація JWT-фільтрації, налаштування доступу до маршрутів, управління сесією. Компоненти цього пакету взаємодіють з Spring Security і забезпечують захист усіх частин системи;
- config – загальна конфігурація Spring Boot-застосунку: конфігураційні класи, бін-конфігурації, компоненти ініціалізації. Це забезпечує централізоване керування параметрами системи;
- utils, mapper тощо – допоміжні пакети, які містять утилітні класи, кастомні конвертери, перетворювачі DTO↔Entity, обробники винятків та інші функціональні елементи, що не належать до основних логічних шарів, але використовуються в різних частинах проєкту;

Така структура дозволяє розробникам легко орієнтуватися в кодовій базі, пришвидшує підтримку та дає змогу масштабувати застосунок без порушення вже реалізованих компонентів. Крім того, вона сприяє дотриманню SOLID-принципів,

адже кожен пакет обмежується своєю зоною відповідальності, а зв’язки між компонентами чітко регламентовані. Всі пакети зображені на рисунку 4.1.

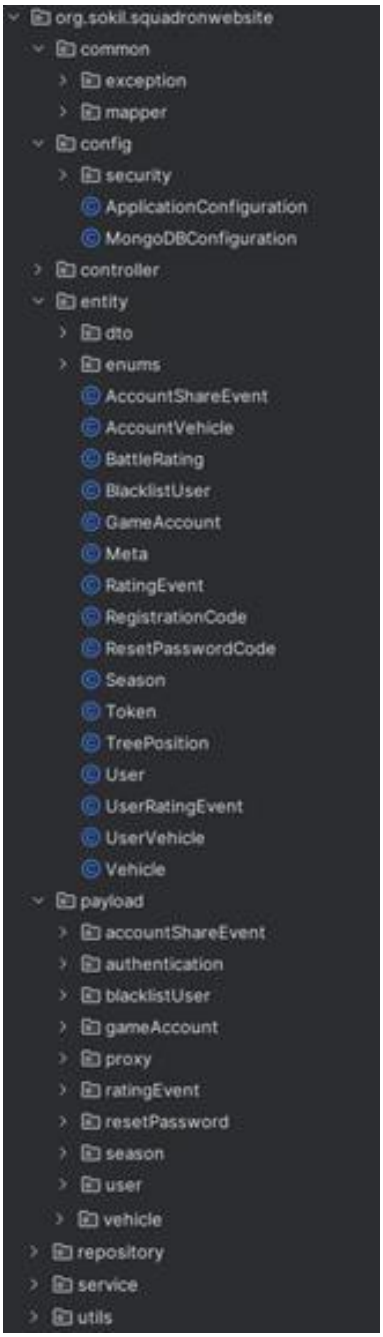


Рис. 4.1 Пакети проєкту

Підхід до організації пакунків повністю відповідає практикам трирівневої архітектури, де взаємодія здійснюється строго по вертикалі: контролери → сервіси → репозиторії → база даних.

4.2 Реалізація основного функціоналу

4.2.1 Реєстрація, автентифікація та керування сесією

Система автентифікації реалізована на основі безсесійного підходу з використанням JWT. Реєстрація нового користувача можлива лише за наявності дійсного реєстраційного коду, що дозволяє обмежити доступ до платформи та контролювати процес приєднання нових учасників.

Користувач проходить автентифікацію, надсилаючи логін і пароль на відповідний endpoint. У разі успішної перевірки облікових даних система генерує токен, який клієнт зберігає на стороні браузера. Цей токен додається до кожного запиту у вигляді заголовка авторизації, що дозволяє серверу автентифікувати користувача на основі вбудованої інформації.

Сторона сервера містить JWT-фільтр, що перехоплює запити до захищених маршрутів, верифікує підпис, перевіряє термін дії та дійсність токена. Завдяки цьому забезпечується повна безсесійність та масштабованість архітектури. В програмному коді цей процес виглядає так:

```
protected void doFilterInternal(
        @NonNull HttpServletRequest request,
        @NonNull HttpServletResponse response,
        @NonNull FilterChain filterChain
    ) throws ServletException, IOException {
        final String authHeader = request.getHeader("Authorization");
        final String jwt;
        final String email;

        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }
        jwt = authHeader.substring(7);
```

```
            email = jwtService.extractEmail(jwt);

            if (email != null && SecurityContextHolder.getContext().getAuthentication()
== null) {
                UserDetails userDetails =
userDetailsService.loadUserByUsername(email);
                if(jwtService.isTokenValid(jwt, userDetails)) {
                    UsernamePasswordAuthenticationToken authenticationToken = new
UsernamePasswordAuthenticationToken(
                            userDetails,
                            null,
                            userDetails.getAuthorities()
                    );
                    authenticationToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(authenticationToken);
                }
            }
            filterChain.doFilter(request, response);
        }
```

4.2.2 Управління користувачами

Користувачі мають змогу керувати власними профілями через відповідні REST API. Передбачено зміну персональних даних, таких як місто, країна, дата народження, номер телефону тощо. Крім того, реалізована можливість оновлення вподобань користувача – наприклад, у контексті улюблених бойових позицій, персональних рейтингів техніки, пріоритетів у грі.

Окрему роль відіграє логіка формування списку техніки, прив’язаного до користувача, який може відрізнятись від техніки на конкретному акаунті. Ця

техніка зберігається як UserVehicle зі своїм ступенем уподобання personalRate, що використовується при створенні команд та аналітиці.

Окрім базових можливостей редагування профілю, користувач може налаштовувати індивідуальні вподобання бойової техніки. Зокрема, реалізовано можливість задавати ступінь персонального пріоритету для певних одиниць техніки, а саме її вподобання від 1 до 5, де 5 – найвищий пріоритет. Ці вподобання зберігаються у вкладеній структурі об'єкта користувача і враховуються при побудові команд та визначенні бойового потенціалу. Таким чином, система підтримує персоналізований підхід до оцінки наявних ресурсів без потреби змінювати базові характеристики техніки.

4.2.3 Робота з акаунтами

Один із базових функціональних блоків платформи – це управління ігровими акаунтами. Користувачі можуть створювати, редагувати та видаляти акаунти, додаючи необхідну інформацію про нікнейм, платформу, тип акаунта (основний, другорядний) та інші характеристики. Кожен акаунт може бути пов'язаний із кількома користувачами – наприклад, для тимчасового надання доступу до техніки.

Система дозволяє користувачу (у межах наданих прав) передавати доступ до акаунтів іншим учасникам. Це здійснюється через відповідні API-запити, які вносять зміни до поля haveAccess у документі GameAccount. Детальніше зрозуміти алгоритм даної процедури допоможе даний відрізок коду та діаграма діяльності, зображена на рисунку 4.2:

```
public GameAccountDto giveAccessToAccount(Principal principal, GiveAccessRequest request) {
        GameAccount gameAccount = gameAccountMapper.fromDto(getGameAccountByNickname(request.accountNickname()));
        User user = userService.getUserByEmail(request.userEmail());
        if (gameAccount.isShareable()) {
           if (principal.getName().equals(gameAccount.getOwner().getEmail())) {
```

```
                gameAccount.getHaveAccess().add(user);
                user.getHasAccessTo().add(gameAccount);
                userRepository.save(user);
                accountShareEventService.createAccountShareEvent(
                        new CreateAccountShareEventRequest(
                                gameAccount,
                                userService.getUserByEmail(principal.getName()),
                                user
                        )
                );
                return
gameAccountMapper.toDto(gameAccountRepository.save(gameAccount));
            }
            throw new GameAccountSharingForbiddenException(String.format("You
haven't permission to share game account with nickname [%s]!",
gameAccount.getNickname()));
        }
        throw new GameAccountNotSharableException(String.format("Game
account with nickname [%s] is not shareable!", gameAccount.getNickname()));
    }
```

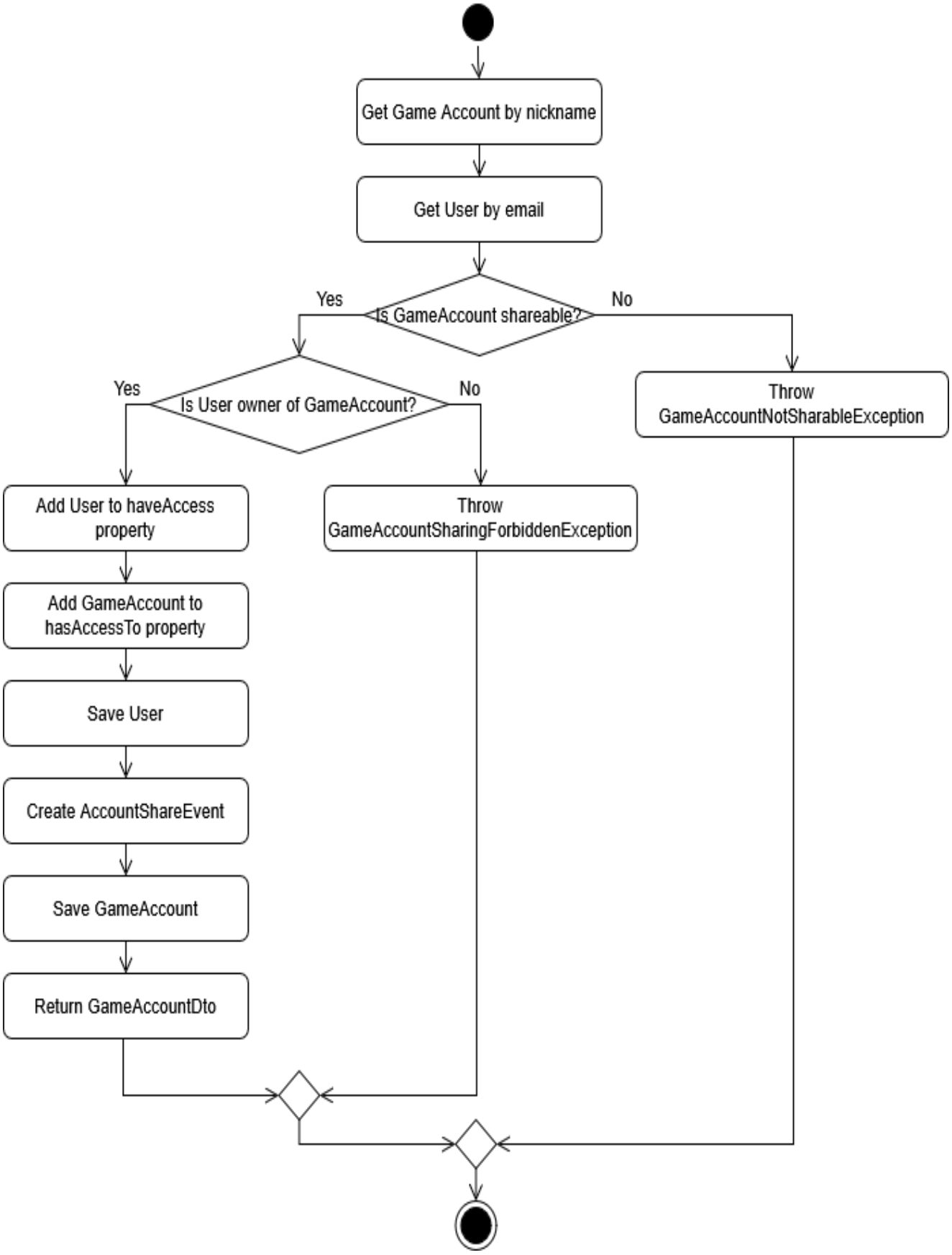


Рис. 4.2 Діаграма діяльності передачі акаунта

Крім того, акаунти можуть містити історію активності, рейтингові значення, дати входу/виходу з полку, що використовується для аналізу динаміки гри. Всі ці поля формують комплексну картину про стан і використання акаунта в рамках сезону чи командного складу.

4.2.4 Робота з технікою

Ключовою складовою функціоналу платформи є можливість керування бойовою технікою, яка закріплена за ігровими акаунтами. Система дозволяє користувачам додавати техніку до власних акаунтів, вибираючи її з централізованої бази одиниць техніки, збережених у відповідній колекції. Завдяки цьому виключається дублювання записів, забезпечується цілісність даних та спрощується оновлення загальних параметрів (тип, нація, гілка, бойовий рейтинг тощо).

Для кожної одиниці техніки в межах акаунта користувач має змогу вказати ступінь дослідження – рівень її ігрової доступності на конкретному обліковому записі. Це дозволяє точно відображати стан акаунта в контексті боєздатності та готовності до командних боїв.

Окремо реалізована функція встановлення метових параметрів техніки, що дозволяє фіксувати найбільш ефективні одиниці на кожному етапі сезону. Ці параметри формуються адміністраторами на основі бойових рейтингів, результатів активності гравців та сезонної статистики. Вони використовуються для оптимального підбору команд і розрахунку рейтингу техніки у поточній мета-структурі гри.

4.2.5 Управління командами та аналітикою

Платформа підтримує функціонал формування бойових підрозділів – команд, що складаються з обраних акаунтів. Створення команди передбачає вибір користувачем наявних акаунтів із власного доступу та закріплення за ними техніки, що відповідає бойовим завданням і метовим критеріям сезону. Сформована в системі команда показана на рисунку 4.3.

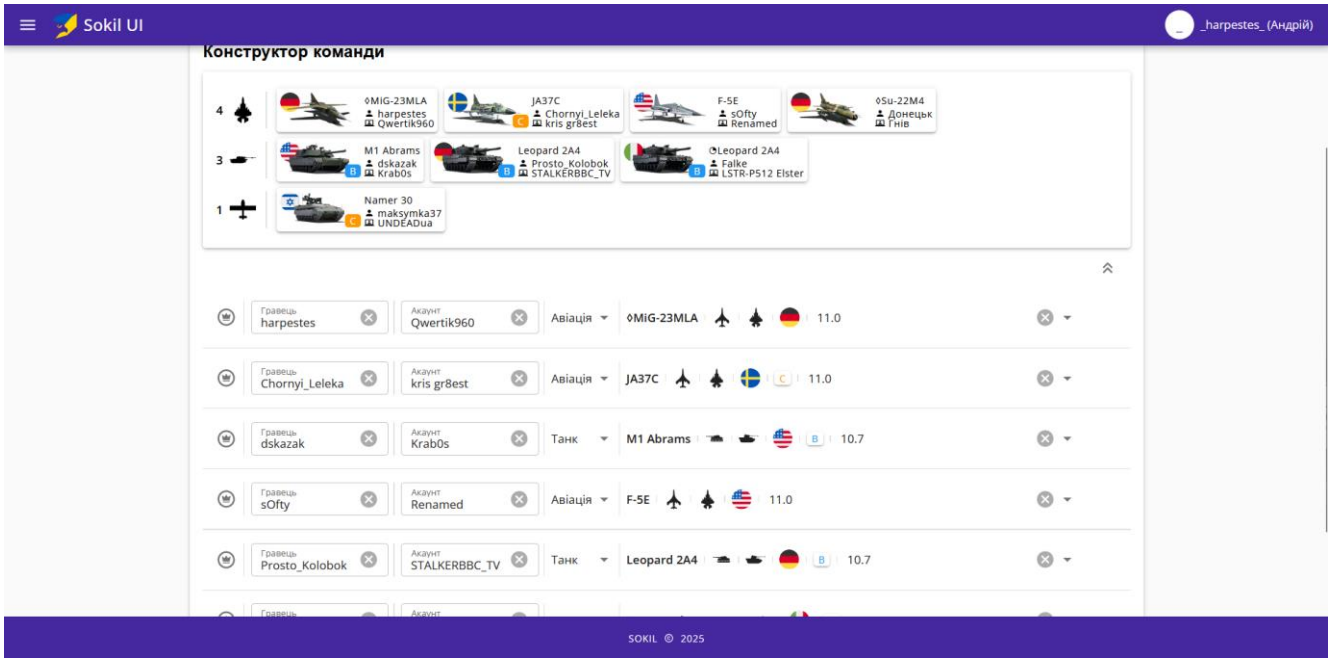


Рис. 4.3 Сформована в системі команда

Однією з важливих функцій є фільтрація техніки за бойовим рейтингом (Battle Rating), що дозволяє ефективно підбирати техніку під конкретні умови бою. Ця можливість значно спрощує процес формування збалансованих складів команд, орієнтованих на оптимальні BR-значення, які задаються як у глобальних параметрах сезону, так і у налаштуваннях конкретного бою.

Користувачі також можуть переглядати зведену статистику – як персональну (прогрес у дослідженні, техніка, вподобання), так і загальну (активність полку, популярність техніки, бойові результати). Всі ці аналітичні дані генеруються автоматично та є ключовими при прийнятті управлінських рішень у межах спільноти.

4.2.6 Модерація та адміністративні можливості

Платформа передбачає розширений функціонал для адміністраторів та модераторів, які відповідають за загальний нагляд за активністю користувачів та станом ігрової спільноти. Одним із ключових інструментів адміністративного керування є перегляд списку всіх зареєстрованих користувачів та акаунтів. Для

цього реалізовано окремі API-запити, доступ до яких відкрито лише ролі ADMIN. Це реалізовано за допомогою перевірки:

```
public List<?> getAllGameAccounts(Principal principal, boolean vehiclesCondensed) {
    User user = userService.getUserByEmail(principal.getName());
    List<Role> userRoles = user.getPermissions();
    if (userRoles.contains(Role.ADMIN)) {
       return aggregateGameAccounts(vehiclesCondensed);
    } else {
       return user.getHasAccessTo().stream().map(gameAccountMapper::toDto).toList();
    }
  }
```

У даному коді при запиті рядового користувача, метод поверне акаунти, що належать цьому користувачу, а при запиті адміністратора – всі акаунти.

Чорний список реалізовано як окрему сутність у системі. Модератори мають можливість додавати нові записи до списку, вказуючи інформацію про порушника, основний акаунт, причину блокування та пов’язаний Discord ID. Також реалізовано механізми редагування та перегляду історії змін, що дозволяє ефективно управляти санкційною політикою спільноти.

Ще одним важливим компонентом адміністративного інтерфейсу є система реєстраційних кодів, які створюються адміністраторами для контрольованої реєстрації нових користувачів. Кожен код має термін дії, статус валідності та є унікальним. Аналогічно реалізовано функціонал скидання паролів, який ґрунтується на механізмі одноразових токенів. Адміністратори мають змогу переглядати, видаляти або деактивувати такі коди у разі потреби.

Усі адміністративні запити обмежені в доступі через механізми авторизації, які перевіряють відповідність ролі та маршруту. Це дозволяє ізолювати критичні точки керування від потенційних загроз з боку звичайних користувачів.

Розмежування цього доступу реалізовано в конфігурації SecurityFilterChain на основі методу authorizeHttpRequests(), з подальшим використанням методу hasAnyAuthority(...). Ці механізми інтегруються з JWT-токеном, який містить інформацію про роль користувача. Це дозволяє захищати будь-який маршрут або бізнес-функцію без зайвого дублювання логіки у коді.

4.3 Взаємодія з базою даних

База даних платформи реалізована на основі MongoDB, що дозволяє ефективно працювати з гнучкими структурами даних та вкладеними об’єктами. З огляду на динамічну природу предметної області, документно-орієнтований підхід був обраний як найбільш відповідний.

Робота з базою даних включає класичні CRUD-операції, а також підтримку MongoDB Aggregation Pipeline для побудови складних запитів. Наприклад, агрегації використовуються для отримання списку техніки на всіх акаунтах користувача або для формування статистики активності за сезоном.

У платформі застосовано комбінований підхід до структурування документів:

- вбудовані структури використовуються там, де об’єкти тісно пов’язані з батьківським документом, наприклад: UserVehicle всередині користувача, AccountVehicle всередині акаунта. Така денормалізація дозволяє швидко отримувати всю потрібну інформацію без додаткових запитів;
- посилання через @DocumentReference застосовуються для зв’язку з глобальними сутностями, що змінюються незалежно: техніка (Vehicle), користувачі (User), акаунти (GameAccount). Це знижує дублювання даних та спрощує централізоване оновлення.

Зміни в базі даних відбуваються без дублювання інформації: наприклад, при оновленні персональної оцінки техніки змінюється лише вкладена структура UserVehicle, без втручання в сам об’єкт Vehicle. Так само зміна списку доступу до акаунта не зачіпає інші його властивості.

Завдяки використанню Spring Data MongoDB, взаємодія з базою даних відбувається у вигляді інтерфейсів-репозиторіїв з автоматично згенерованими запитами, що значно спрощує розробку та супровід застосунку.

4.4 Розгортання та конфігурація середовища

Для забезпечення ефективного розгортання та підтримки проєкту “SOKIL” застосовано сучасні інструменти контейнеризації, хмарного деплою та автоматизації CI/CD. Це дозволило досягти гнучкості, повторюваності й стабільності в процесі розгортання серверної частини web-платформи.

Проєкт упаковано в Docker-контейнер, що містить повне серверне середовище: Java-рантайм, Spring Boot-застосунок, усі залежності та системні налаштування. Для цього створено файл Dockerfile, у якому описано базовий образ, етапи збирання проєкту та команду запуску. Використання Docker гарантує однакову поведінку застосунку на всіх етапах життєвого циклу – від локальної розробки до продакшн-середовища.

Хостинг застосунку здійснюється за допомогою платформи Fly.io, яка забезпечує масштабоване середовище для виконання контейнеризованих застосунків. Застосунок публікується безпосередньо з Docker-контейнера, який автоматично збирається і завантажується через CLI Fly.

Процес розгортання автоматизовано так, що при пуші в основну гілку репозиторію GitHub (main) Fly.io отримує оновлену версію контейнера та автоматично перезапускає інстанс.

Для реалізації безперервної інтеграції та доставки (CI/CD) у проєкті налаштовано GitHub Actions. Основні задачі, які виконуються через пайплайни:

- перевірка коректності збірки;
- запуск автоматичних тестів;
- збірка Docker-образу;
- деплой у середовище Fly.io після мерджу в main гілку.

Це забезпечує повну автоматизацію розгортання та дозволяє швидко вносити зміни до проєкту без втручання у процес доставки. Ця процедура конфігурується спеціальним файлом fly.yml:

```
name: Fly Deploy
on:
  push:
    branches:
      - main
env:
  MONGODB_URI: mongodb+srv://${{ secrets.MONGODB_USERNAME }}:${{ secrets.MONGODB_PASSWORD }}@sokildb.q3aevuz.mongodb.net/?retryWrites=true&w=majority&appName=${{ secrets.MONGDB_APP_NAME }}
  DB_NAME: ${{ secrets.MONGODB_DB_NAME }}
  TOKEN_GENERATION_KEY: ${{ secrets.TOKEN_GENERATION_KEY }}
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up JDK 21
        uses: actions/setup-java@v3
        with:
          java-version: '21'
          distribution: 'adopt'
          cache: maven
      - name: Build with Maven
        run: mvn clean package
      - name: Deploy our Spring Boot app to fly.io
        uses: superfly/flyctl-actions/setup-flyctl@master
```

```
- run: flyctl deploy --remote-only -e MONGODB_URI=$MONGODB_URI -e
DB_NAME=$DB_NAME                                                    -e
TOKEN_GENERATION_KEY=$TOKEN_GENERATION_KEY
        env:
          FLY_ACCESS_TOKEN: ${{ secrets.FLY_TOKEN }}
```

4.5 Тестування системи

Надійність і коректність функціонування web-платформи забезпечується через багаторівневе тестування, яке охоплює модульні, інтеграційні та ручні перевірки. Тестування є невід'ємною частиною процесу розробки, спрямованою на виявлення логічних помилок, порушень валідації, некоректної обробки виняткових ситуацій та перевірки доступу до маршрутів відповідно до ролей користувачів.

4.5.1 Модульне тестування

Модульне тестування охоплює окремі сервіси та їхню бізнес-логіку. Для ізоляції залежностей використовувалися бібліотеки мокування, зокрема Mockito. Це дозволило протестувати поведінку сервісів незалежно від бази даних чи зовнішніх компонентів.

Кожен сервіс перевіряється на наявність правильних відповідей у типових і граничних сценаріях. Наприклад, тестування методу оновлення техніки включає пошук техніки для оновлення та її отримання, оновлення параметрів і збереження оновленого об’єкта. Тестування метода полягає в перевірці чи збігається результат методу з очікуваним та чи потрібні вкладені методи викликалися потрібну кількість разів:

```
void testUpdateVehicle() {
        UpdateVehicleRequest request = new UpdateVehicleRequest(…);

        Vehicle vehicle = Vehicle.builder()
                …
```

```
                .build();


when(vehicleRepository.findById(VALID_ID)).thenReturn(Optional.of(vehicle));
        when(vehicleRepository.save(any(Vehicle.class))).thenReturn(vehicle);

        Vehicle result = vehicleService.updateVehicle(request);

        assertNotNull(result);
        assertEquals(result.getName(), vehicle.getName());
        assertEquals(result.getBattleRate(), vehicle.getBattleRate());
        assertEquals(result.getNation(), vehicle.getNation());
        assertEquals(result.getBranch(), vehicle.getBranch());
        assertEquals(result.getType(), vehicle.getType());
        assertEquals(result.getMetas(), vehicle.getMetas());
        assertEquals(vehicle.getTreePosition(), result.getTreePosition());
        assertEquals(vehicle.getBannerUrl(), result.getBannerUrl());
        assertEquals(vehicle.getIconUrl(), result.getIconUrl());
        assertEquals(vehicle.getAdvicePageLink(), result.getAdvicePageLink());
        assertEquals(vehicle.getWikiLink(), result.getWikiLink());
        verify(vehicleRepository, times(1)).findById(anyString());
        verify(vehicleRepository, times(1)).save(any(Vehicle.class));
    }
```

Таким чином, тестування підтверджує правильність змін у внутрішніх структурах без доступу до мережі або сховища, що забезпечує швидкість та ізоляцію перевірки.

4.5.2 Інтеграційне тестування

Інтеграційне тестування було спрямоване на перевірку коректної взаємодії між компонентами системи, а саме – між REST API, сервісною логікою та базою даних. Для цього використовувався Postman.

Postman застосовувався для ручної перевірки API-ендпоінтів у середовищі, максимально наближеному до реального. Було створено спеціальну колекцію запитів (Postman Collection), що включала: реєстрацію користувача, логін, додавання акаунта, керування технікою, модерацію чорного списку, генерацію реєстраційного коду тощо. Кожен запит містив відповідні токени автентифікації, необхідні тіла запитів і перевірки відповідей. Приклад запиту зображено на рисунку 4.4.

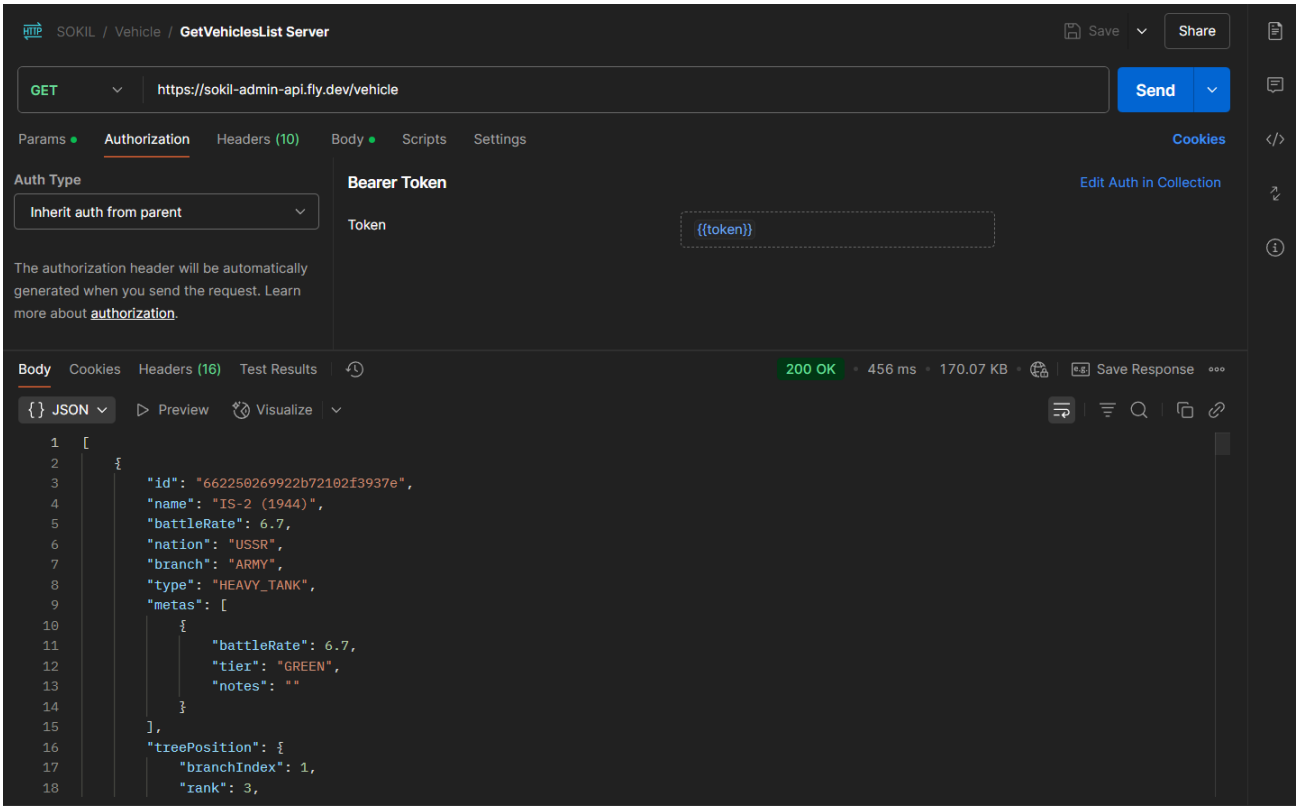


Рис. 4.4 Приклад виконаного запиту в Postman

З запиту видно, що для його успішного виконання потребується автентифікаційний токен, а також видно результат виконання цього запиту, що повертає список всієї техніки у форматі JSON.

4.5.3 Ручне тестування системи

Ручне тестування проводилось для перевірки узгодженості поведінки інтерфейсу API із очікуваними функціональними вимогами. Тестування охоплювало основні сценарії користування системою, зокрема реєстрацію, створення акаунтів, додавання техніки, передавання доступу, роботу з чорним списком тощо.

У процесі ручного тестування були виявлені незначні помилки в логіці перевірки прав доступу до деяких маршрутів. Наприклад, на початковому етапі тестування модератор мав змогу викликати адміністративні ендпоінти. Це було виправлено шляхом уточнення маршрутів для ролей у конфігурації безпеки.

Результати тестування підтвердили відповідність реалізованого функціоналу технічним вимогам, а виявлені недоліки були усунуті до фінального деплою.

4.6 Аналіз результатів реалізації

У результаті реалізації серверної частини web-платформи для менеджменту ігрової соціальної структури “SOKIL” вдалося досягти повного покриття попередньо сформульованих функціональних та нефункціональних вимог. Система забезпечує централізоване управління ігровими акаунтами, технікою, ролями, рейтингами та взаємодією між користувачами, що є критично важливим для організації внутрішніх процесів спільноти в грі War Thunder.

З технічного боку реалізовано всі основні сценарії: реєстрація й автентифікація користувачів з використанням реєстраційного коду; розмежування доступу до функціоналу відповідно до ролі; створення й модифікація акаунтів; керування технікою й уподобаннями; формування команд; модерація чорного списку; адміністрування реєстраційних та відновлювальних кодів. Платформа дозволяє працювати як з особистими акаунтами, так і з тими, до яких користувач має спільний доступ, при цьому зберігаючи контроль за історією передавання та активністю.

Систему побудовано з урахуванням архітектурних принципів, які забезпечують її гнучкість, надійність і масштабованість. Завдяки трирівневій структурі (контролер — сервіс — репозиторій) досягнуто чіткого розділення відповідальностей, що спрощує підтримку та розширення проєкту. Впроваджене авторизаційне рішення на основі JWT дозволяє безпечно обслуговувати запити без необхідності збереження сесій, що, у свою чергу, покращує продуктивність і масштабованість. Базу даних MongoDB обрано не лише через її гнучкість щодо структури документів, а й через зручну підтримку вкладених масивів, денормалізованих даних і зв’язків через @DocumentReference, що дало змогу зберегти баланс між читабельністю, продуктивністю та узгодженістю.

Автоматизація процесу збирання та розгортання проєкту реалізована через GitHub Actions у поєднанні з Docker-контейнеризацією та платформою Fly.io. Завдяки цьому було забезпечено стабільне оновлення продакшн-версії, що знижує ймовірність людських помилок і пришвидшує процес інтеграції нових функцій.

Особливу роль у стабільності роботи системи відіграє модуль тестування, який охоплює як модульні перевірки окремих сервісів, так і інтеграційні тести REST API, зокрема з перевіркою прав доступу, валідації, обробки виключень і типових сценаріїв взаємодії. Тестування продемонструвало відповідність очікуваній логіці роботи та дало змогу оперативно усунути критичні помилки до публічного розгортання.

Переваги застосованого підходу:

- гнучкість структури даних – завдяки NoSQL-підходу реалізовано динамічну модель акаунтів, техніки та користувачів із можливістю швидкого розширення;
- безпека – реалізовано повноцінну систему контролю доступу з перевіркою ролей, шифруванням паролів, обмеженням доступу до маршрутів та CORS-конфігурацією;
- масштабованість – завдяки безсесійності та використанню хмарної інфраструктури система може обслуговувати зростаючу кількість користувачів без значного рефакторингу;

- прозорість і підтримуваність – архітектура платформи забезпечує зрозумілу структуру та легкість у внесенні змін без ризику для інших компонентів.

Напрямки подальшого розвитку:

- інтеграція з офіційними API гри War Thunder (за умови надання доступу), що дозволить автоматично підтягувати техніку, рейтинги, історію боїв;
- розширення аналітики – реалізація модуля глибокого аналізу активності користувачів, техніки, популярності гілок прокачування тощо;
- мікросервісна декомпозиція – поділ на окремі сервіси (користувачі, техніка, аналітика) з незалежним розгортанням;
- інтеграція з Discord – автоматичне оновлення ролей на сервері, сповіщення про нові реєстрації або зміни в чорному списку.

Таким чином, розроблена система відповідає сучасним вимогам до архітектури, безпеки та масштабованості web-застосунків і має потенціал для використання не лише в межах одного полку чи гри, а й як база для подібних платформ у сфері менеджменту ігрових спільнот.

ВИСНОВКИ

Результатом виконаної роботи стала web-платформа для централізованого управління структурою ігрової спільноти в межах гри War Thunder. Реалізація системи забезпечила автоматизацію багатьох процесів, що раніше виконувалися вручну, зокрема реєстрації гравців, керування акаунтами, технікою, доступами, формування команд, модерації користувачів і технічної аналітики. Це дозволяє значно підвищити ефективність функціонування спільноти, зменшити адміністративне навантаження та забезпечити прозору взаємодію між її учасниками.

У ході розробки були вирішені наступні задачі:

1. Проведено аналіз предметної галузі гри та особливостей функціонування полкових структур. Встановлено, що в грі War Thunder полкові формування мають складну ієрархію, залежність від бойової техніки та акаунтів гравців, а також потребують цифрових інструментів для керування командною взаємодією.

2. Дослідження існуючих сервісів засвідчило відсутність повноцінних рішень для гнучкого управління ігровими акаунтами та технікою. Більшість доступних платформ є надто загальними або обмеженими функціонально, що обґрунтувало доцільність створення власної web-платформи.

3. Аналіз сучасних технологій розробки web-застосунків дозволив обрати інструменти, які забезпечили надійність, безпеку та масштабованість системи. Стек на базі Spring Boot, MongoDB, JWT і Docker підтвердив свою ефективність у створенні розширюваної платформи з гнучкою структурою даних і безсесійною архітектурою доступу.

4. Сформовані вимоги стали основою для подальшої архітектурної та функціональної реалізації системи. Чітке визначення ролей, сценаріїв використання й обмежень дозволило уникнути суперечностей у реалізації та забезпечити відповідність очікуванням кінцевих користувачів.

5. Проєктування архітектури на основі принципів SOLID та трирівневої моделі забезпечило логічну структуру коду, що спростило підтримку системи. Таке

архітектурне рішення дозволяє легко масштабувати застосунок і впроваджувати нові функціональні модулі без ризику порушення існуючих залежностей.

6. Реалізовано REST API для управління користувачами, акаунтами, технікою, командами та чорним списком, що дозволило автоматизувати процеси менеджменту ігрової спільноти.

7. Використання JWT у поєднанні зі Spring Security забезпечило захищеність системи та контроль доступу без втрати продуктивності. Такий підхід дозволив ефективно захистити маршрути, виключити несанкціонований доступ і забезпечити масштабованість за рахунок безсесійної обробки запитів.

8. Комплексне тестування компонентів системи підтвердило її стабільність і відповідність технічним вимогам. Модульні, інтеграційні та ручні тести дозволили виявити та усунути критичні помилки до розгортання застосунку в продакшн-середовищі.

9. Застосунок розгорнуто за допомогою Docker, що спростило подальше розгортання та обслуговування. Автоматизація процесів CI/CD забезпечила стабільність, відтворюваність середовища і швидке оновлення продакшн-версії без залучення додаткових ручних операцій.

Таким чином, поставлені в межах роботи задачі виконано в повному обсязі. Система “SOKIL” є життєздатним технічним рішенням, що відповідає вимогам ігрової спільноти, забезпечує зручність керування соціальною структурою полку та має потенціал для подальшого розвитку: впровадження інтерфейсу користувача, мікросервісної декомпозиції, інтеграції з Discord та ігровими API.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Дикало А.Ю., Замрій І.В. Важливість дотримання архітектурних принципів у проєктуванні веб-застосунків на базі Spring Boot. // Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 24.04.2025, ДУІКТ, Київ, Україна, С. 61-64.

2. Дикало А.Ю., Замрій І.В. Перспективи підвищення безпеки у веб-додатках на базі Spring Boot з використанням JWT. // Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025». 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, С. 279-281.

ПЕРЕЛІК ПОСИЛАНЬ

1. Block S., Haack F. eSports: a new industry. *SHS Web of Conferences*. 2021. Vol. 92. P. 04002. URL: https://doi.org/10.1051/shsconf/20219204002 (дата звернення: 14.05.2025).
2. Werder K. Esport. *Business & Information Systems Engineering*. 2022. URL: https://doi.org/10.1007/s12599-022-00748-w (дата звернення: 15.05.2025).
3. Hamari J., Sjöblom M. What is eSports and why do people watch it?. *Internet Research*. 2017. Vol. 27, no. 2. P. 211–232. URL: https://doi.org/10.1108/intr-04-2016-0085 (дата звернення: 15.05.2025).
4. Virtual(ly) Athletes: Where eSports Fit Within the Definition of “Sport” / S. E. Jenny et al. *Quest*. 2016. Vol. 69, no. 1. P. 1–18. URL: https://doi.org/10.1080/00336297.2016.1144517 (дата звернення: 15.05.2025).
5. Taylor T. L. Raising the Stakes: E-Sports and the Professionalization of Computer Gaming. MIT Press, 2012. 336 p.
6. Carter M., Gibbs M., Harrop M. Metagames, paragames and orthogames. *the International Conference*, Raleigh, North Carolina, 29 May – 1 June 2012. New York, New York, USA, 2012. URL: https://doi.org/10.1145/2282338.2282346 (дата звернення: 15.05.2025).
7. Funk D. C., Pizzo A. D., Baker B. J. eSport management: Embracing eSport education and research opportunities. *Sport Management Review*. 2018. Vol. 21, no. 1. P. 7–13. URL: https://doi.org/10.1016/j.smr.2017.07.008 (дата звернення: 15.05.2025).
8. The ingredients of Twitch streaming: Affordances of game streams / M. Sjöblom et al. *Computers in Human Behavior*. 2019. Vol. 92. P. 20–28. URL: https://doi.org/10.1016/j.chb.2018.10.012 (дата звернення: 16.05.2025).
9. 6 Video Games (Sorry, ‘Simulators’) the US Military Used for Training. *Military.com*. URL: https://www.military.com/off-duty/games/6-video-games-sorry-simulators-us-military-used-training.html (дата звернення: 16.05.2025).

10. Riddell R. Doom Goes To War. *WIRED*. URL: https://www.wired.com/1997/04/ff-doom (дата звернення: 16.05.2025).

11. Przała M. Tanker in Ukraine Reports His Job Easier Having Played Video Games. *Gamepressure.com*. URL: https://www.gamepressure.com/newsroom/tanker-in-ukraine-reports-his-job-easier-having-played-video-game/z867f5 (дата звернення: 16.05.2025).

12. Ed Scarce. Bradley Crew Credits Gamer Training On 'War Thunder' To Take Out T-90. *Crooks and Liars*. URL: https://crooksandliars.com/2024/01/bradley-crew-credits-gamer-training-war (дата звернення: 16.05.2025).

13. Katwala A. The military is turning to Twitch to fix its recruitment crisis. *WIRED*. URL: https://www.wired.com/story/british-army-video-game-recruitment (дата звернення: 16.05.2025).

14. Schwartzburg R. The US military is embedded in the gaming world. Its target: teen recruits. *the Guardian*. URL: https://www.theguardian.com/us-news/2024/feb/14/us-military-recruiting-video-games-targeting-teenagers (дата звернення: 16.05.2025).

15. Kent S. L. For military, games hone combat skills. *Newspapers.com*. URL: https://www.newspapers.com/article/usa-today-for-military-games-hone-comba/169784408/ (дата звернення: 17.05.2025).

16. Developer's Dilemma: The Secret World of Videogame Creators / T. Pinch et al. MIT Press, 2014. 352 p.

17. Taylor T. L. Play Between Worlds: Exploring Online Game Culture. MIT Press, 2009. 206 p.

18. ThunderSkill | War Thunder Player Stats & Squadron. *ThunderSkill*. URL: https://thunderskill.com/en (дата зверенення: 17.05.2025).

19. New | War Thunder Wiki. *New | War Thunder Wiki*. URL: https://wiki.warthunder.com/ (дата зверенення: 17.05.2025).

20. Guilded - Chat for Gaming Communities. *Guilded - Chat for Gaming Communities*. URL: https://www.guilded.gg/ (дата зверенення: 17.05.2025).

21. Walls C. Spring Boot in Action. Manning Publications, 2016. 264 p.

22. Spilca L. Spring Security in Action. Manning Publications Co. LLC, 2020.

23. RFC 7519. JSON Web Token (JWT). [Effective from 2015-05-01]. Official edition. Internet Engineering Steering Group (IESG), 2015. 2 p. URL: https://doi.org/10.17487/RFC7519 (дата зверенення 17.05.2025).

24. Project Lombok. *Project Lombok*. URL: https://projectlombok.org/ (дата звернення: 19.05.2025).

25. Merkel D. Docker: lightweight Linux containers for consistent development and deployment. *Linux J.* 2014. Vol. 2014, no. 239.

26. IBM. What Is Three-Tier Architecture? | IBM. *IBM - United States*. URL: https://www.ibm.com/think/topics/three-tier-architecture (дата звернення: 19.05.2025).

27. Ramachandrappa N. C. SOLID design principles in software engineering. International journal of computer trends and technology. 2024. Vol. 72, no. 9. P. 18–23. URL: https://doi.org/10.14445/22312803/ijctt-v72i9p104.

ДОДАТОК А. ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Backend-розробка web-платформи для менеджменту ігрової соціальної структури "SOKIL"

Виконав студент 4 курсу
групи ПД-41
Дикало Андрій Юрійович
Керівник роботи
д.т.н., професор, зав. кафедри ІПЗ Замрій Ірина Вікторівна

Київ – 2025

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи** – спрощення менеджменту організації учасників ігрової онлайн-спільноти
- **Об'єкт дослідження** – процес функціонування ігрових онлайн-спільнот як соціальних структур у цифровому середовищі
- **Предмет дослідження** – web-платформа для підтримки та організації учасників ігрової спільноти

2

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз предметної галузі гри та особливостей функціонування полкових структур.
2. Дослідити існуючі сервіси для управління ігровими акаунтами та технікою в онлайн-іграх.
3. Проаналізувати сучасні технології розробки безпечних web-застосунків (Spring Boot, Spring Security, MongoDB, Docker, JWT).
4. Сформувати вимоги до функціоналу та архітектури web-платформи.
5. Спроєктувати архітектуру клієнт-серверного застосунку з урахуванням обраного стеку технологій.
6. Реалізувати REST API для управління користувачами, акаунтами, технікою, командами та чорним списком.
7. Розробити механізм авторизації та автентифікації з використанням JWT і захистом CORS.
8. Провести тестування основних компонентів системи.
9. Здійснити розгортання web-платформи з використанням Docker.

3

АНАЛІЗ АНАЛОГІВ

Показник	Thunder Skill	Боти	War Thunder Wiki	Google Sheets / Notion	Guilded	Web-платформа “SOKIL”
Підтримка War Thunder	Часткова	Так (неофіційно)	Так (офіційна база)	Так (ручне введення)	Ні	Так (підлаштована)
Керування акаунтами	Ні	Ні	Ні	Частково	Ні	Так
Керування технікою	Ні	Частково	Так	Частково	Ні	Так
Полкова координація	Ні	Так	Ні	Частково	Так	Так
Автоматизація	Так	Частково	Ні	Ні	Частково	Так
Персоналізація	Ні	Ні	Ні	Частково	Ні	Так
Платформа/Тип	Web-застосунок	Боти	Вікі-сайт	Табличні системи	Платформа спільнот	Web-застосунок

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Реєстрація та авторизація користувачів (реєстрація нового користувача відбувається лише за реєстраційним кодом, наданим адміністратором).
2. Розмежування прав доступу(рядовий користувач, модератор, адміністратор).
3. Створення та управління ігровими акаунтами (реєстрація нових ігрових акаунтів, керування списком техніки акаунта).
4. Управління технікою (користувачі з адміністраторською роллю можуть створювати, редагувати та видаляти техніку; звичайні користувачі можуть додавати техніку на свій ігровий акаунт та виставляти ступінь її дослідження та преференції).

Нефункціональні вимоги:

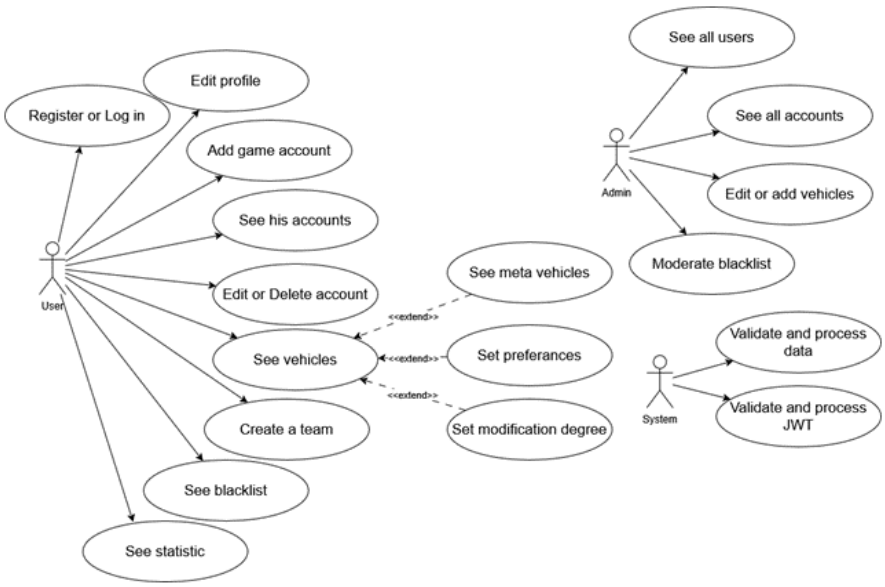
1. Безпека застосунку (захист від CSRF, шифрування паролів за допомогою bcrypt)
2. Масштабованість
3. Кросплатформенність

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

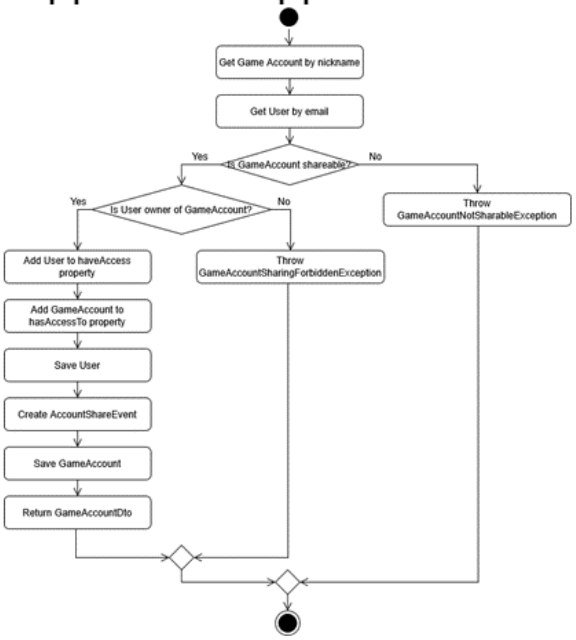
6

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



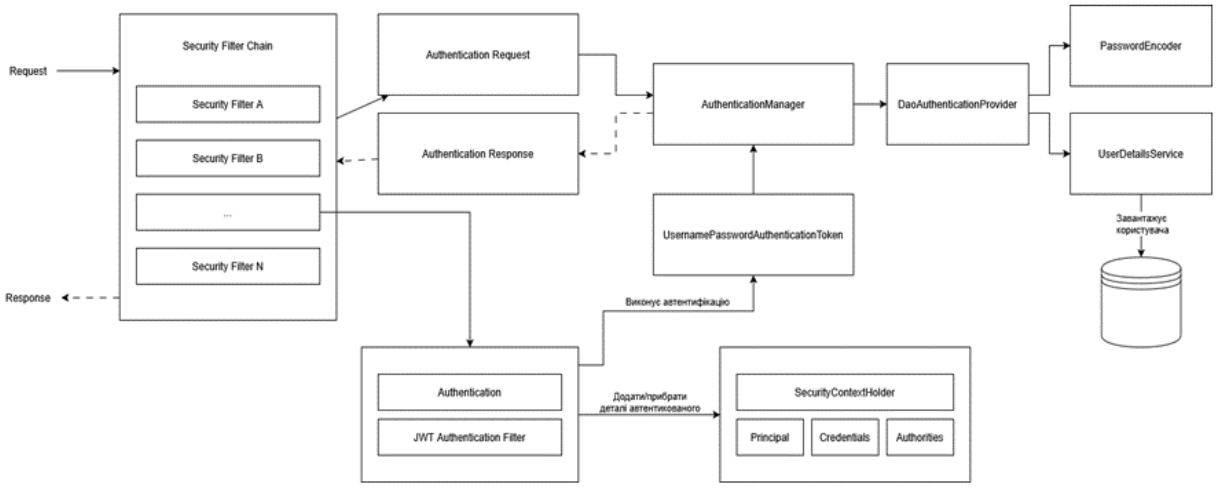
7

ДІАГРАМА ДІЯЛЬНОСТІ



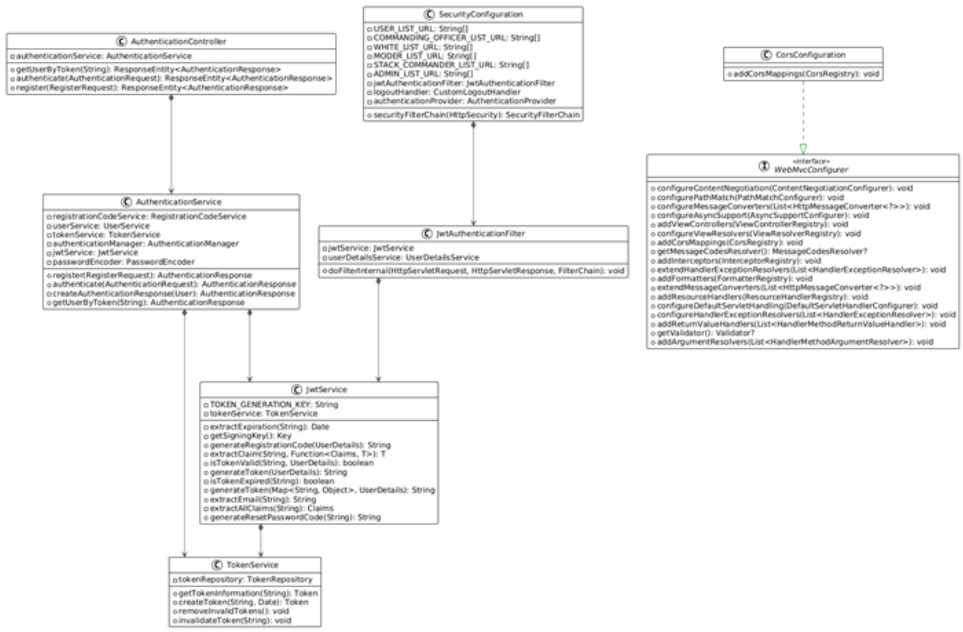
8

ДІАГРАМА АРХІТЕКТУРИ БЕЗПЕКИ WEB-ПЛАТФОРМИ



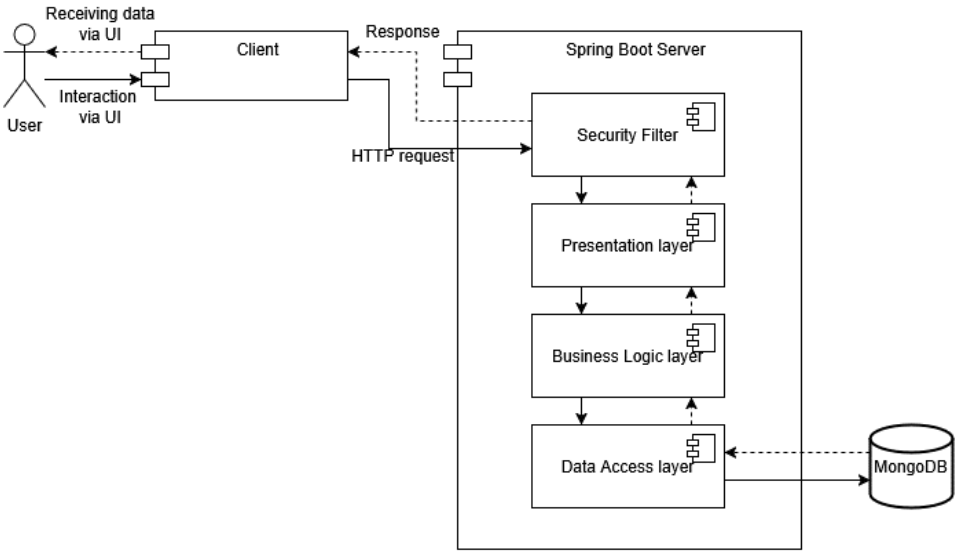
9

ДІАГРАМА КЛАСІВ БЕЗПЕКИ WEB-ПЛАТФОРМИ



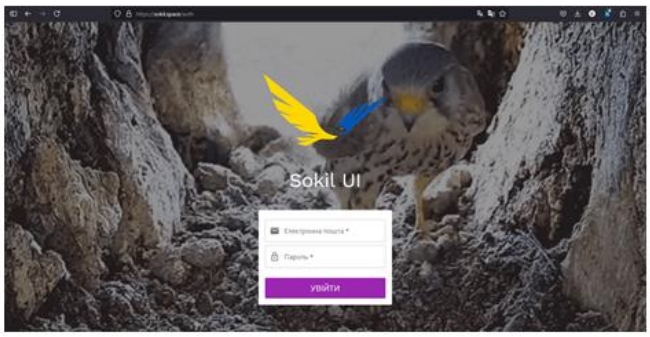
10

ДІАГРАМА ЗАГАЛЬНОЇ АРХІТЕКТУРИ

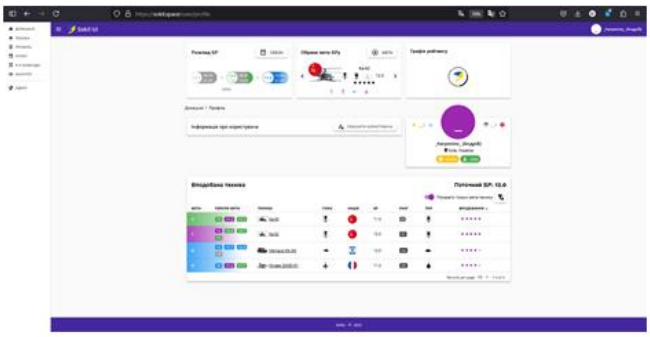


11

ЕКРАННІ ФОРМИ



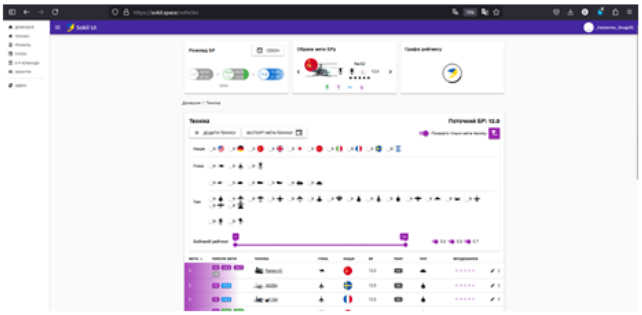
Сторінка авторизації



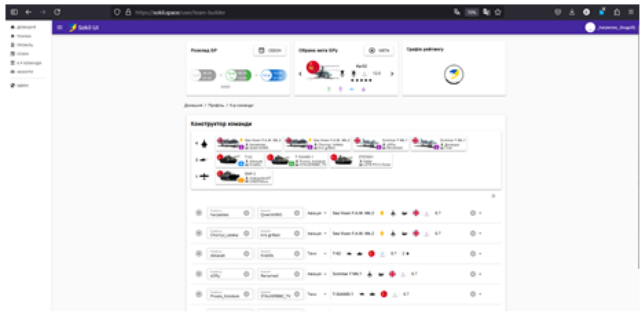
Сторінка профілю користувача

12

ЕКРАННІ ФОРМИ



Сторінка техніки



Сторінка конструктора команди

13

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Дикало А.Ю., Замрій І.В. ВАЖЛИВІСТЬ ДОТРИМАННЯ АРХІТЕКТУРНИХ ПРИНЦИПІВ У ПРОЄКТУВАННІ ВЕБ-ЗАСТОСУНКІВ НА БАЗІ SPRING BOOT. // Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 24.04.2025, ДУІКТ, Київ, Україна, С. 61-64.
2. Дикало А.Ю., Замрій І.В. ПЕРСПЕКТИВИ ПІДВИЩЕННЯ БЕЗПЕКИ У ВЕБ-ДОДАТКАХ НА БАЗІ SPRING BOOT З ВИКОРИСТАННЯМ JWT. // Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025». 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, С. 279-281.

14

ВИСНОВКИ

1. Проведення аналізу гри War Thunder та полкових структур дало змогу врахувати специфіку взаємодії акаунтів і техніки.
2. Дослідження аналогічних сервісів виявило їх обмеження, що підтвердило актуальність розробки власної платформи.
3. Обґрунтовано вибір сучасного стеку технологій для створення безпечного та масштабованого застосунку.
4. Сформовано вимоги до функціоналу та архітектури платформи відповідно до потреб користувачів, які були використані при проєктуванні.
5. Спроєктовано архітектуру клієнт-серверного застосунку, що дозволило спростити розробку.
6. Реалізовано REST API для управління користувачами, акаунтами, технікою, командами та чорним списком, що дозволило автоматизувати процеси менеджменту ігрової спільноти.
7. Створення системи авторизації та автентифікації з JWT, CORS-захистом та шифруванням паролів забезпечило захищеність даних користувачів.
8. Проведено тестування, яке підтвердило стабільність та працездатність компонентів.
9. Застосунок розгорнуто за допомогою Docker, що спростило подальше обслуговування.

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

```
package
org.sokil.squadronwebsite.controller;

import lombok.RequiredArgsConstructor;
import
org.sokil.squadronwebsite.entity.AccountShareEve
nt;
import
org.sokil.squadronwebsite.service.AccountShareEv
entService;
import
org.springframework.http.ResponseEntity;
import
org.springframework.stereotype.Controller;
import
org.springframework.web.bind.annotation.GetMapp
ing;
import
org.springframework.web.bind.annotation.Request
Mapping;
import
org.springframework.web.bind.annotation.RequestP
aram;

import java.util.List;

@Controller
@RequiredArgsConstructor
@RequestMapping("/account-share-event")
public class AccountShareEventController {

    private final AccountShareEventService
accountShareEventService;

    @GetMapping
    public
ResponseEntity<List<AccountShareEvent>>
getRatingEventsByTimestampIn(@RequestParam
String timestamps) {
        return
ResponseEntity.ok(accountShareEventService.getA
ccountShareEventsByTimestampIn(timestamps));
    }
}
```

```
package org.sokil.squadronwebsite.controller;

import lombok.RequiredArgsConstructor;
import
org.sokil.squadronwebsite.payload.authentication.A
uthenticationRequest;
import
org.sokil.squadronwebsite.payload.authentication.A
uthenticationResponse;
import
org.sokil.squadronwebsite.payload.authentication.R
egisterRequest;
import
org.sokil.squadronwebsite.service.AuthenticationSe
rvice;
import org.springframework.http.HttpHeaders;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

@Controller
@RequiredArgsConstructor
@RequestMapping("/auth")
public class AuthenticationController {

  private final AuthenticationService
authenticationService;

  @PostMapping("/register")
  public ResponseEntity<AuthenticationResponse>
register(
      @RequestBody RegisterRequest request
  ) {
    return
ResponseEntity.ok(authenticationService.register(re
quest));
  }

  @PostMapping("/authenticate")
  public ResponseEntity<AuthenticationResponse>
authenticate(
      @RequestBody AuthenticationRequest
request
  ) {
    return
ResponseEntity.ok(authenticationService.authentica
te(request));
  }

  @GetMapping("/user")
  public ResponseEntity<AuthenticationResponse>
getUserByToken(

@RequestHeader(HttpHeaders.AUTHORIZATION
) String authHeader
  ) {
```

```
        return
ResponseEntity.ok(authenticationService.getUserB
yToken(authHeader));
    }
}
package org.sokil.squadronwebsite.controller;

import lombok.RequiredArgsConstructor;
import
org.sokil.squadronwebsite.entity.BlacklistUser;
import
org.sokil.squadronwebsite.payload.blacklistUser.Cr
eateBlacklistUserRequest;
import
org.sokil.squadronwebsite.payload.blacklistUser.Up
dateBlacklistUserRequest;
import
org.sokil.squadronwebsite.service.BlacklistUserSer
vice;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@Controller
@RequiredArgsConstructor
@RequestMapping("/blacklist")
public class BlacklistUserController {

    private final BlacklistUserService
blacklistUserService;

    @PostMapping
    public ResponseEntity<BlacklistUser>
createBlacklistUser(@RequestBody
CreateBlacklistUserRequest request) {
        return
ResponseEntity.ok(blacklistUserService.createBlac
klistUser(request));
    }

    @GetMapping
    public ResponseEntity<List<BlacklistUser>>
getAllBlacklistUsers() {
        return
ResponseEntity.ok(blacklistUserService.getAllBlac
klistUsers());
    }

    @GetMapping("/{id}")
    public ResponseEntity<BlacklistUser>
getBlacklistUserById(@PathVariable String id) {
        return
ResponseEntity.ok(blacklistUserService.getBlacklis
tUserById(id));
    }

    @GetMapping(params = "discordId")
    public ResponseEntity<BlacklistUser>
getBlacklistUserByDiscordId(@RequestParam
Long discordId) {
        return
ResponseEntity.ok(blacklistUserService.getBlacklis
tUserByDiscordId(discordId));
    }

    @GetMapping(params =
"mainAccountNickname")
    public ResponseEntity<BlacklistUser>
getBlacklistUserByMainAccountNickname(@Requ
estParam String mainAccountNickname) {
        return
ResponseEntity.ok(blacklistUserService.getBlacklis
tUserByMainAccountNickname(mainAccountNick
name));
    }

    @PutMapping
    public ResponseEntity<BlacklistUser>
updateBlacklistUser(@RequestBody
UpdateBlacklistUserRequest request) {
        return
ResponseEntity.ok(blacklistUserService.updateBlac
klistUser(request));
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteBlacklistUser(@PathVariable
String id) {

blacklistUserService.deleteBlacklistUserById(id);
    }
}
package org.sokil.squadronwebsite.controller;

import lombok.RequiredArgsConstructor;
import
org.sokil.squadronwebsite.entity.dto.GameAccount
Dto;
import
org.sokil.squadronwebsite.entity.dto.UserDto;
import
org.sokil.squadronwebsite.payload.gameAccount.*;
import
org.sokil.squadronwebsite.service.GameAccountSe
rvice;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;
```

```
import java.security.Principal;
import java.util.List;

@Controller
@RequiredArgsConstructor
@RequestMapping("/game-account")
public class GameAccountController {

    private final GameAccountService gameAccountService;

    @GetMapping("/list")
    public ResponseEntity<List<?>> getGameAccounts(Principal principal, @RequestParam(required = false) boolean vehiclesCondensed) {
        return ResponseEntity.ok(gameAccountService.getAllGameAccounts(principal, vehiclesCondensed));
    }

    @GetMapping("/{nickname}")
    public ResponseEntity<GameAccountDto> getGameAccountByNickname(@PathVariable String nickname) {
        return ResponseEntity.ok(gameAccountService.getGameAccountByNickname(nickname));
    }

    @GetMapping
    public ResponseEntity<GameAccountDto> getMainGameAccount(Principal principal) {
        return ResponseEntity.ok(gameAccountService.getMainGameAccount(principal));
    }

    @PostMapping("/create")
    public ResponseEntity<GameAccountDto> createGameAccount(
            Principal principal,
            @RequestBody CreateGameAccountRequest request
    ) {
        return ResponseEntity.ok(gameAccountService.createGameAccount(principal, request));
    }

    @PatchMapping("/vehicles")
    public ResponseEntity<GameAccountDto> updateAccountVehicles(@RequestBody UpdateAccountVehiclesRequest request) {
        return ResponseEntity.ok(gameAccountService.updateVehicles(request));
    }

    @PatchMapping("/update")
    public ResponseEntity<GameAccountDto> updateGameAccount(@RequestBody UpdateGameAccountRequest request) {
        return ResponseEntity.ok(gameAccountService.updateGameAccount(request));
    }

    @PostMapping("/give-access")
    public ResponseEntity<GameAccountDto> giveAccessToAccount(Principal principal, @RequestBody GiveAccessRequest request) {
        return ResponseEntity.ok(gameAccountService.giveAccessToAccount(principal, request));
    }

    @GetMapping("/has-access/{hasAccessEmail}")
    public ResponseEntity<List<GameAccountDto>> getOwnerGameAccount(@PathVariable String hasAccessEmail) {
        return ResponseEntity.ok(gameAccountService.getAccountsByUserEmail(hasAccessEmail));
    }

    @GetMapping("/{accountNickname}/users")
    public ResponseEntity<List<UserDto>> getUsersWithAccessTo(@PathVariable String accountNickname) {
        return ResponseEntity.ok(gameAccountService.getUsersWithAccessTo(accountNickname));
    }

    @PatchMapping("/link")
    public HttpStatus linkGameAccount(Principal principal, LinkGameAccountRequest request) {

        gameAccountService.linkMainGameAccount(principal, request);
        return HttpStatus.OK;
    }
}
package org.sokil.squadronwebsite.controller;


import lombok.RequiredArgsConstructor;
import org.sokil.squadronwebsite.payload.proxy.ProxyRequest;
import org.sokil.squadronwebsite.service.ProxyService;
```

```
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import
org.springframework.web.bind.annotation.PatchMa
pping;
import
org.springframework.web.bind.annotation.Request
Body;
import
org.springframework.web.bind.annotation.Request
Mapping;

@Controller
@RequiredArgsConstructor
@RequestMapping("/proxy")
public class ProxyController {

    private final ProxyService proxyService;

    @PatchMapping
    public                ResponseEntity<String>
handleApiRequest(@RequestBody    ProxyRequest
proxyRequest) {
        return
ResponseEntity.ok(proxyService.handleApiRequest
(proxyRequest));
    }
}
package org.sokil.squadronwebsite.controller;

import lombok.RequiredArgsConstructor;
import
org.sokil.squadronwebsite.entity.RatingEvent;
import
org.sokil.squadronwebsite.entity.UserRatingEvent;
import
org.sokil.squadronwebsite.payload.ratingEvent.Cre
ateRatingEventRequest;
import
org.sokil.squadronwebsite.service.RatingEventServ
ice;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@Controller
@RequiredArgsConstructor
@RequestMapping("/rating-event")
public class RatingEventController {

    private          final          RatingEventService
ratingEventService;

    @PostMapping
    public                ResponseEntity<RatingEvent>
createRatingEvent(@RequestBody
CreateRatingEventRequest
createRatingEventRequest) {
        return
ResponseEntity.ok(ratingEventService.createRating
Event(createRatingEventRequest));
    }

    @GetMapping
    public        ResponseEntity<List<RatingEvent>>
getRatingEventsByTimestampIn(@RequestParam
String timestamps) {
        return
ResponseEntity.ok(ratingEventService.getRatingEv
entsByTimestampIn(timestamps));
    }

    @GetMapping("/user-ratings")
    public  ResponseEntity<List<UserRatingEvent>>
getUserRatingEventsByTimestampInAndUsername
(
            @RequestParam String timestamps,
            @RequestParam String username
    ) {
        return
ResponseEntity.ok(ratingEventService.getUserRati
ngEventsByTimestampInAndUsername(timestamps
, username));
    }
}
package org.sokil.squadronwebsite.controller;

import lombok.RequiredArgsConstructor;
import
org.sokil.squadronwebsite.entity.RegistrationCode;
import
org.sokil.squadronwebsite.entity.dto.InitiatorDto;
import
org.sokil.squadronwebsite.service.RegistrationCode
Service;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

import java.security.Principal;
import java.util.List;

@Controller
@RequiredArgsConstructor
@RequestMapping("/registration-code")
public class RegistrationCodeController {

    private          final          RegistrationCodeService
registrationCodeService;
```

```
    @GetMapping("/create")
    public ResponseEntity<String>
generateRegistrationCode(Principal principal) {
        return
ResponseEntity.ok(registrationCodeService.generat
eRegistrationCode(principal));
    }

    @GetMapping("/initiator")
    public ResponseEntity<InitiatorDto>
getInitiatorByCode(@RequestParam String code) {
        return
ResponseEntity.ok(registrationCodeService.getIniti
atorByCode(code));
    }

    @DeleteMapping("/purge")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void removeInvalidCodes() {

registrationCodeService.removeInvalidCodes();
    }

    @GetMapping
    public ResponseEntity<List<RegistrationCode>>
getAvailableCodes() {
        return
ResponseEntity.ok(registrationCodeService.getAva
ilableCodes());
    }
}
package org.sokil.squadronwebsite.controller;

import lombok.RequiredArgsConstructor;
import
org.sokil.squadronwebsite.entity.ResetPasswordCo
de;
import
org.sokil.squadronwebsite.payload.resetPassword.R
esetPasswordRequest;
import
org.sokil.squadronwebsite.service.ResetPasswordC
odeService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@Controller
@RequiredArgsConstructor
@RequestMapping("/reset-password")
public class ResetPasswordCodeController {

    private final ResetPasswordCodeService
resetPasswordCodeService;

    @GetMapping("/create-code/{userEmail}")
    public ResponseEntity<String>
generateResetPasswordCode(@PathVariable String
userEmail) {
        return
ResponseEntity.ok(resetPasswordCodeService.gene
rateResetPasswordCode(userEmail));
    }

    @DeleteMapping("/purge")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void removeInvalidCodes() {

resetPasswordCodeService.removeInvalidCodes();
    }

    @GetMapping
    public
ResponseEntity<List<ResetPasswordCode>>
getAvailableCodes() {
        return
ResponseEntity.ok(resetPasswordCodeService.getA
vailableCodes());
    }

    @PatchMapping
    public ResponseEntity<Boolean>
resetPassword(@RequestBody
ResetPasswordRequest request) {
        return
ResponseEntity.ok(resetPasswordCodeService.reset
Password(request));
    }
}
package org.sokil.squadronwebsite.controller;

import lombok.RequiredArgsConstructor;
import
org.sokil.squadronwebsite.entity.BattleRating;
import org.sokil.squadronwebsite.entity.Season;
import
org.sokil.squadronwebsite.payload.season.CreateSe
asonRequest;
import
org.sokil.squadronwebsite.payload.season.UpdateS
easonRequest;
import
org.sokil.squadronwebsite.service.SeasonService;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@Controller
@RequiredArgsConstructor
```

```
@RequestMapping("/season")
public class SeasonController {

    private final SeasonService seasonService;

    @PostMapping("/create")
    public ResponseEntity<Season> createSeason(@RequestBody CreateSeasonRequest request) {
        return ResponseEntity.ok(seasonService.createSeason(request));
    }

    @GetMapping("/br")
    public ResponseEntity<List<BattleRating>> getCurrentSeasonBattleRatings() {
        return ResponseEntity.ok(seasonService.getCurrentSeasonBattleRatings());
    }

    @GetMapping("/br/current")
    public ResponseEntity<BattleRating> getCurrentBattleRating() {
        return ResponseEntity.ok(seasonService.getCurrentBattleRating());
    }

    @GetMapping("/br/{seasonName}")
    public ResponseEntity<Season> getSeasonByName(@PathVariable String seasonName) {
        return ResponseEntity.ok(seasonService.getSeasonByName(seasonName));
    }

    @GetMapping("/current")
    public ResponseEntity<Season> getCurrentSeason() {
        return ResponseEntity.ok(seasonService.getCurrentSeason());
    }

    @PatchMapping("/update")
    public ResponseEntity<Season> updateSeason(@RequestBody UpdateSeasonRequest request) {
        return ResponseEntity.ok(seasonService.updateSeason(request));
    }
}
package org.sokil.squadronwebsite.controller;

import lombok.RequiredArgsConstructor;
import org.sokil.squadronwebsite.service.TokenService;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;

@Controller
@RequiredArgsConstructor
@RequestMapping("/token")
public class TokenController {

    private final TokenService tokenService;

    @DeleteMapping("/purge")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void removeInvalidTokens() {
        tokenService.removeInvalidTokens();
    }
}
package org.sokil.squadronwebsite.controller;

import lombok.RequiredArgsConstructor;
import org.sokil.squadronwebsite.entity.User;
import org.sokil.squadronwebsite.entity.dto.ExtendedUserDto;
import org.sokil.squadronwebsite.payload.user.UpdateUserRequest;
import org.sokil.squadronwebsite.payload.user.UpdateVehiclePreferenceRequest;
import org.sokil.squadronwebsite.service.UserService;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

import java.security.Principal;
import java.util.List;

@Controller
@RequiredArgsConstructor
@RequestMapping("/user")
public class UserController {
```

```
    private final UserService userService;

    @PatchMapping("/preference")
    public ResponseEntity<User>
updateVehiclePreference(Principal principal,
@RequestBody
List<UpdateVehiclePreferenceRequest> request) {
        return
ResponseEntity.ok(userService.updateVehiclePrefe
rence(principal, request));
    }

    @PatchMapping("/update")
    public ResponseEntity<User>
updateUser(Principal principal, @RequestBody
UpdateUserRequest request) {
        return
ResponseEntity.ok(userService.updateUser(princip
al, request));
    }

    @GetMapping("/all")
    public
ResponseEntity<List<ExtendedUserDto>>
getAllUsers() {
        return
ResponseEntity.ok(userService.getAllUsers());
    }

    @GetMapping("/{id}")
    public ResponseEntity<ExtendedUserDto>
getUserById(@PathVariable String id) {
        return
ResponseEntity.ok(userService.getUserById(id));
    }
}
package org.sokil.squadronwebsite.controller;

import lombok.RequiredArgsConstructor;
import org.sokil.squadronwebsite.entity.Vehicle;
import
org.sokil.squadronwebsite.payload.vehicle.CreateV
ehicleRequest;
import
org.sokil.squadronwebsite.payload.vehicle.UpdateV
ehicleRequest;
import
org.sokil.squadronwebsite.service.VehicleService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Map;

@Controller
@RequiredArgsConstructor
@RequestMapping("/vehicle")
public class VehicleController {

    private final VehicleService vehicleService;

    @GetMapping("/{id}")
    public ResponseEntity<Vehicle>
getVehicleById(@PathVariable String id) {
        return
ResponseEntity.ok(vehicleService.getVehicleById(
id));
    }

    @PostMapping
    public ResponseEntity<Vehicle>
createVehicle(@RequestBody
CreateVehicleRequest request) {
        return
ResponseEntity.ok(vehicleService.createVehicle(re
quest));
    }

    @PutMapping
    public ResponseEntity<Vehicle>
updateVehicle(@RequestBody
UpdateVehicleRequest request) {
        return
ResponseEntity.ok(vehicleService.updateVehicle(re
quest));
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteVehicle(@PathVariable String
id) {
        vehicleService.deleteVehicleById(id);
    }

    @GetMapping
    public ResponseEntity<List<Vehicle>>
getVehiclesList(
        @RequestParam Map<String, String>
requestParams
    ) {
        return
ResponseEntity.ok(vehicleService.getVehiclesList(
requestParams));
    }
}
```