МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: «**Розробка ігрового сервера Minecraft»**

Виконав: студент 4 курсу, групи КН-41
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Дужий Артур Андрійович

Керівник: *викладач кафедри інформаційних технологій та аналітики даних*
Мацевич Денис Володимирович

Рецензент: *кандидат технічних наук, доцент, доцент кафедри прикладної математики Донецького національного університету імені Василя Стуса*
Загоруйко Любов Василівна

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних _____
(проф., д.е.н. Кривицька О.Р.)

Протокол № 11 від «20» травня 2026 р.

Острог, 2026

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1. Проаналізувати архітектурні підходи до побудови масштабованих ігрових систем та обґрунтувати вибір серверного ядра Paper як високоефективної основи для організації багатокористувацької взаємодії.
2. Спроєктувати гібридну розподілену архітектуру платформи, поєднавши монолітне ігрове ядро з інфраструктурними мікросервісами, та реалізувати централізований комунікаційний TCP-сервер із підтримкою Protocol Buffers.
3. Розробити систему управління обліковими записами з підтримкою авторизації за протоколом OAuth 2.0, двофакторної автентифікації (TOTP) та інтегрованим S3-сумісним модулем керування скінами гравців.
4. Реалізувати сервіс ігрових сесій, сумісний зі специфікацією Yggdrasil Session Server, та допоміжні сервіси динамічної генерації аватарів і асинхронних електронних повідомлень на базі RabbitMQ.
5. Створити настільний кросплатформений лаунчер на базі фреймворку Wails (Go, React 19) та модифікований клієнт Minecraft із впровадженням ігрових модулів і кастомної системи автентифікації.
6. Налаштувати автоматизовані процеси CI/CD на базі GitHub Actions, контейнеризацію сервісів у Docker, а також забезпечити мережеву безпеку платформи за допомогою Cloudflare Tunnel та реверс-проксі Caddy.

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: *Розробка ігрового сервера Minecraft*

Автор: *Дужий Артур Андрійович*

Науковий керівник: *викладач кафедри інформаційних технологій та аналітики даних Мацевич Денис Володимирович*

Захищена «……»…… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: *88 (кількість сторінок роботи) с., 10 (кількість рисунків) рис., 1 (кількість таблиць) табл., 0 (кількість додатків) додатків, 14 (кількість джерел) джерел.*

Ключові слова: *MINECRAFT, МІКРОСЕРВІСНА АРХІТЕКТУРА, GOFIBER, OAUTH 2.0, RABBITMQ, PAPER, CLOUDFLARE, ІГРОВИЙ СЕРВЕР, ЛАУНЧЕР, БЕКЕНД, ІГРОВА СЕСІЯ, GITHUB ACTIONS, CLEAN ARCHITECTURE.*

Короткий зміст праці: *Кваліфікаційну роботу присвячено розробці ігрового сервера Minecraft «Toweray» із застосуванням мікросервісної архітектури та сучасних інструментів серверного програмування. У теоретичній частині проведено аналіз Minecraft як платформи для багатокористувацької взаємодії, розглянуто архітектурні підходи до побудови масштабованих ігрових систем, здійснено порівняльний аналіз серверних ядер та обґрунтовано вибір Paper як основи серверної частини. На етапі проєктування визначено загальну концепцію системи та спроєктовано її ключові функціональні компоненти: центральний комунікаційний сервіс на базі TCP з підтримкою Protocol Buffers, сервіс управління обліковими записами з OAuth 2.0 авторизацією, сервіс ігрових сесій, сумісний із протоколом Yggdrasil, сервіс генерації аватарів, асинхронний сервіс електронних повідомлень на базі RabbitMQ, а також клієнтська частина та лаунчер. Програмна реалізація виконана мовами Go та Java з дотриманням принципів чистої архітектури. Розгортання системи здійснено із застосуванням Docker-контейнеризації, CI/CD-пайплайнів на основі GitHub Actions, реєстру образів ZotRegistry та Portainer для оперативного управління інфраструктурою. Мережева безпека забезпечена через Cloudflare Tunnel та реверс-проксі Caddy. Практична перевірка архітектурних рішень підтверджена на прикладі реалізованого ігрового режиму GunGame.*

(підпис автора)

ABSTRACT
of the Bachelor's thesis
for the award of a Bachelor's degree

Subject: *Development of a Minecraft game server*

Author: *Arthur Duzhyy*

Academic Supervisor: *Lecturer of the Department of Information Technologies and Data Analytics Denys Volodymyrovych Matsevych*

Defended on «……»…… 20__

Explanatory Note to the Thesis: *88 (number of pages) p., 10 (number of figures) figs., 1 (number of tables) tables, 0 (number of appendices) appendices, 14 (number of references) references.*

Keywords: *MINECRAFT, MICROSERVICE ARCHITECTURE, GOFIBER, OAUTH 2.0, RABBITMQ, PAPER, CLOUDFLARE, GAME SERVER, LAUNCHER, BACKEND, GAME SESSION, GITHUB ACTIONS, CLEAN ARCHITECTURE.*

Summary: *This thesis is dedicated to the development of a Minecraft game server "Toweray" using a microservice architecture and modern server-side programming tools. The theoretical part analyzes Minecraft as a platform for multiplayer interaction, examines architectural approaches to building scalable gaming systems, compares existing server cores, and justifies the selection of Paper as the server foundation. During the design phase, the overall system concept was defined and its key functional components were architected: a central communication service based on TCP with Protocol Buffers support, a user account management service with OAuth 2.0 authorization, a game session service compatible with the Yggdrasil protocol, an avatar generation service, an asynchronous email notification service built on RabbitMQ, as well as a custom game client and launcher. The software implementation was carried out in Go and Java, following clean architecture principles. System deployment was accomplished using Docker containerization, CI/CD pipelines built with GitHub Actions, a ZotRegistry image registry, and Portainer for operational infrastructure management. Network security is ensured through Cloudflare Tunnel and the Caddy reverse proxy. The practical validity of the architectural decisions was confirmed through the implementation of the GunGame game mode.*

(author's signature)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

Скорочення	Розшифрування
ACID	Atomicity, Consistency, Isolation, Durability (атомарність, узгодженість, ізоляція, довговічність)
API	Application Programming Interface (інтерфейс програмування застосунків)
CI/CD	Continuous Integration / Continuous Delivery (безперервна інтеграція та доставлення)
DDoS	Distributed Denial of Service (розподілена відмова в обслуговуванні)
DLX	Dead Letter Exchange (обмінник недоставлених повідомлень)
DSL	Domain Specific Language (предметно-орієнтована мова)
HTTP	HyperText Transfer Protocol (протокол передачі гіпертексту)
HTTPS	HyperText Transfer Protocol Secure (захищений протокол передачі гіпертексту)
IPC	Instructions Per Clock (інструкцій за такт)
JSON	JavaScript Object Notation (формат обміну даними)
JVM	Java Virtual Machine (віртуальна машина Java)
MCP	Minecraft Coder Pack
OAuth	Open Authorization (відкрита авторизація)
RFC 6238	Request for Comments 6238 (стандарт алгоритму одноразових паролів за часом)
SMTP	Simple Mail Transfer Protocol (простий протокол передавання електронної пошти)
TCP	Transmission Control Protocol (протокол керування передачею)
TOTP	Time-Based One-Time Password (одноразовий пароль за часом)
TPS	Ticks Per Second (тиків за секунду)

TLS	Transport Layer Security (протокол захисту транспортного рівня)
URL	Uniform Resource Locator (уніфікований локатор ресурсу)
UX/UI	User Experience / User Interface (досвід користувача / інтерфейс користувача)

ВСТУП

Сучасні комп’ютерні ігри дедалі частіше використовують багатокористувацький режим, що потребує стабільної та продуктивної серверної інфраструктури. Однією з найпопулярніших платформ для онлайн-взаємодії є Minecraft, який надає широкі можливості для створення власних ігрових серверів із унікальним функціоналом, системами керування та модифікаціями. Розробка ігрового сервера Minecraft є актуальною задачає, оскільки потребує застосування сучасних інформаційних технологій, засобів мережевого програмування, систем адміністрування та забезпечення безпеки даних. Крім того, створення власного сервера дозволяє реалізувати індивідуальні механіки гри, оптимізувати продуктивність та покращити взаємодію між користувачами.

Метою дослідження є розробка ігрового сервера Minecraft із реалізацією функціоналу для забезпечення стабільної багатокористувацької взаємодії, адміністрування та підтримки ігрового процесу.

Для досягнення поставленої мети необхідно виконати такі задачі дослідження:

- провести аналіз предметної області та особливостей функціонування ігрових серверів Minecraft;
- здійснити проєктування архітектури інформаційної системи та визначити її основні компоненти;
- обрати інструментальні засоби, технології та методи реалізації програмного продукту;
- розробити серверну інфраструктуру з підтримкою автентифікації, управління сесіями та ігровими даними гравців;
- Реалізувати міжсервісну комунікацію для забезпечення взаємодії між компонентами системи;
- налаштувати процес збирання, контейнеризації та розгортання програмного продукту.

Об’єктом дослідження є ігровий сервер Minecraft як інформаційна система для організації багатокористувацької взаємодії.

Предметом дослідження є методи, програмні засоби та інформаційні технології розробки ігрових серверів Minecraft, включаючи управління аутентифікацією, сесіями, базами даних, кешуванням, міжсервісною комунікацією та контейнеризацією.

У роботі використовуються методи аналізу предметної області, об’єктно-орієнтованого проєктування, мережевого програмування та моделювання клієнт-серверної взаємодії.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ ІГРОВОГО СЕРВЕРА MINECRAFT

1.1 Minecraft та його роль у сучасній індустрії ігор

Minecraft є однією з найбільш відомих та унікальних ігор у жанрі sandbox, яка за час свого існування змогла перетворитися з простої ігрової концепції на повноцінну багатофункціональну платформу для створення та організації багатокористувацької взаємодії. Починаючи з моменту виходу першої публічної версії у 2009 році, гра поступово набула глобального поширення та стала одним із наймасовіших цифрових продуктів у сфері інтерактивних розваг. Сьогодні Minecraft об’єднує величезну кількість користувачів по всьому світу, формуючи активну та постійно зростаючу спільноту, яка взаємодіє через мережеві серверні рішення.

Важливою особливістю Minecraft як ігрової системи є його висока гнучкість та можливість створення власних серверних середовищ, які можуть суттєво відрізнятися між собою як за правилами гри, так і за внутрішньою логікою, механіками та загальною концепцією взаємодії гравців. Саме ця властивість стала ключовим чинником у формуванні великої кількості різноманітних проєктів, починаючи від класичних режимів виживання, де основна увага приділяється базовим механікам гри, і завершуючи складними багатокористувацькими системами, що включають економічні моделі, рівневу систему розвитку, спеціалізовані ігрові режими та індивідуальні механіки взаємодії між користувачами.

Завдяки такій архітектурній гнучкості Minecraft почав використовуватися не лише як розважальний продукт, а й як своєрідне середовище для експериментів у сфері багатокористувацьких онлайн-систем. У межах серверної інфраструктури гравці отримують можливість взаємодіяти в реальному часі, спільно виконувати ігрові завдання, будувати складні конструкції, брати участь у змаганнях або формувати власні ігрові сценарії. Усе це робить серверну частину гри критично важливою складовою всієї екосистеми, оскільки саме вона забезпечує стабільність, узгодженість та коректність ігрового процесу.

Як програмна система, ігровий сервер Minecraft можна розглядати як складний багатокористувацький програмний комплекс, основною задачею якого є забезпечення взаємодії великої кількості клієнтів у межах спільного віртуального середовища. Така система відповідає за обробку подій, що надходять від користувачів, синхронізацію їхніх дій, підтримку актуального стану ігрового світу, а також передачу інформації між усіма учасниками взаємодії.

Функціонування подібної системи потребує врахування значної кількості вимог, які безпосередньо впливають на її стабільність та продуктивність. Зокрема, сервер повинен бути здатним обробляти велику кількість одночасних підключень без суттєвого зниження продуктивності. Крім цього, важливим є забезпечення стабільної роботи системи без втрати або пошкодження даних, що особливо критично у випадках тривалої роботи серверів. Значну роль також відіграє точна синхронізація дій гравців, оскільки навіть незначні затримки або розсинхронізація можуть суттєво впливати на загальний ігровий досвід. Окремо слід звертати увагу на питання безпеки, які включають контроль доступу користувачів, захист від несанкціонованих дій та запобігання використанню сторонніх переваг у грі.

З точки зору внутрішньої структури, серверна частина Minecraft складається з низки взаємопов’язаних функціональних компонентів, кожен з яких виконує окрему роль у загальному процесі обробки даних. До таких компонентів належать модуль обробки ігрової логіки, який відповідає за основні правила та механіки гри, мережевий модуль, що забезпечує взаємодію між сервером та клієнтами, підсистема управління ігровим світом, яка контролює стан блоків і простору, система обліку гравців та їхніх даних, а також додаткові інструменти для виконання команд та розширення функціональності сервера.

У класичному підході до реалізації подібних систем усі зазначені компоненти зазвичай об’єднуються в межах одного програмного середовища, що з одного боку спрощує запуск та базову підтримку системи, але з іншого боку створює певні обмеження щодо масштабування та подальшого розвитку. Саме тому у сучасних підходах до проєктування подібних систем дедалі частіше застосовується принцип

розділення функціоналу на окремі логічні модулі, які можуть взаємодіяти між собою через мережеві протоколи та незалежно виконувати свої задачі.

Таким чином, Minecraft як платформа для багатокористувацької взаємодії представляє не просто гру, а складну багаторівневу екосистему, що поєднує в собі елементи клієнт-серверної архітектури, мережевих технологій, розподілених систем та програмних рішень реального часу. Усе це формує високу складність розробки серверної частини та робить створення власного ігрового сервера актуальною, сучасною та технічно цікавою задачею, яка потребує комплексного підходу до проєктування, реалізації та подальшої оптимізації.

1.2 Архітектурні підходи до розробки масштабованих ігрових серверів

У процесі створення сучасних багатокористувацьких ігрових систем особливого значення набуває вибір архітектурного підходу, оскільки саме він визначає загальну організацію програмної системи, принципи її масштабування та спосіб взаємодії між її компонентами. У випадку ігрових серверів Minecraft це питання є особливо важливим, оскільки система має забезпечувати стабільну роботу при одночасній взаємодії великої кількості користувачів у режимі реального часу.

1.2.1 Монолітна архітектура

Монолітна архітектура – один із традиційних підходів до створення програмних систем, за якого всі ключові функціональні частини працюють у межах одного застосунку. У такій моделі система існує як єдине ціле: внутрішні модулі взаємодіють напряму, без винесення логіки в окремі зовнішні сервіси чи потреби в розподіленій інфраструктурі.

У світі Minecraft-серверів моноліт історично є стандартним способом реалізації серверної частини. Більшість популярних ядер на кшталт Paper або Spigot побудовані саме так: один серверний процес одночасно відповідає за ігрову логіку, стан світу, синхронізацію дій гравців і виконання плагінів.

Характерна риса цього підходу – концентрація великої кількості функцій у

межах одного процесу або навіть одного плагіна. На практиці один плагін нерідко поєднує обробку івентів, взаємодію з базою даних, збереження даних про гравців, економічні механіки, систему покарань та інші допоміжні можливості.

Такий формат робить структуру системи простішою: компоненти залишаються щільно інтегрованими та працюють в одному середовищі виконання. Завдяки цьому зменшується потреба у додатковій інфраструктурі, а розгортання сервера зазвичай зводиться до запуску на одному хості.

Серед основних плюсів монолітної архітектури можна виділити:

- відносну легкість розробки та адміністрування;
- відсутність складної мережевої взаємодії між частинами системи;
- швидку внутрішню роботу модулів без мережевих затримок;
- простіший контроль цілісності даних;
- зручне розгортання на одному сервері.

Для малих і середніх Minecraft-проєктів цього часто достатньо: система може стабільно працювати без побудови складної багаторівневої інфраструктури, а відсутність мережевих “прокладок” позитивно впливає на швидкодію внутрішніх процесів.

Разом із тим моноліт має відчутні мінуси. Оскільки все працює в одному процесі, серйозне навантаження або критична помилка здатні вплинути на весь сервер. Це створює проблему “єдиної точки відмови” і ускладнює підтримку масштабних багатокористувацьких платформ.

Окремо стоїть питання масштабування: найчастіше продуктивність моноліту підвищують вертикально – додаючи ресурси одному серверу. Однак такий шлях має фізичні межі й не завжди вигідний з точки зору витрат.

Крім того, зі зростанням функціоналу та ускладненням логіки система стає менш гнучкою для розвитку: сильні зв’язки між частинами ускладнюють виділення підсистем у незалежні модулі та підвищують складність підтримки коду.

Попри ці обмеження, монолітна архітектура й надалі широко використовується в Minecraft-проєктах завдяки простоті, високій швидкості внутрішньої взаємодії та зручній інтеграції з плагінною екосистемою серверних ядер.

1.2.2 Мікросервісна архітектура

Мікросервісна архітектура передбачає побудову системи у вигляді набору окремих незалежних компонентів, кожен з яких виконує власну функцію та може функціонувати автономно від інших частин системи. На відміну від монолітного підходу, у такій архітектурі функціональність розподіляється між декількома сервісами, які взаємодіють між собою через мережеві протоколи або програмні інтерфейси.

У сфері Minecraft-серверів мікросервісний підхід використовується переважно для побудови допоміжної інфраструктури навколо основного ігрового сервера. При цьому безпосередня ігрова логіка, обробка подій, взаємодія між гравцями та управління ігровим світом продовжують виконуватись у межах серверного ядра та плагінів.

Отже, мікросервіси у Minecraft-проєктах не замінюють серверну частину гри, а виступають як зовнішній інфраструктурний рівень, який забезпечує виконання додаткових задач.

У сучасних Minecraft-проєктах подібний підхід найчастіше використовується для реалізації:

- систем авторизації та управління обліковими записами;
- централізованих сервісів збереження даних;
- веб-платформ та особистих кабінетів користувачів;
- систем синхронізації між ігровими серверами;
- обробки внутрішніх платежів та донатних операцій;
- аналітики та статистики;
- допоміжних сервісів роботи з профілями та текстурами користувачів.

У випадку використання власного клієнта гри або лаунчера можуть також додатково застосовуватись окремі сервіси ігрових сесій, централізованої автентифікації та управління зовнішнім виглядом персонажів.

Важливою особливістю подібної архітектури є те, що основна взаємодія із зовнішніми сервісами здійснюється через серверні плагіни. Саме плагіни виступають проміжною ланкою між ігровим середовищем та інфраструктурними компонентами системи. Через них виконується отримання користувацьких даних, синхронізація інформації між серверами та інтеграція допоміжних механізмів у межах ігрового процесу.

Однією з головних переваг мікросервісної архітектури є можливість винесення частини навантаження за межі ігрового сервера. Це дозволяє зменшити кількість додаткових задач, які виконуються безпосередньо у серверному процесі, а також забезпечує незалежний розвиток окремих компонентів системи.

До основних переваг такого підходу можна віднести:

- незалежне масштабування окремих сервісів;
- підвищену стійкість інфраструктури до відмов;
- можливість використання різних технологій для різних компонентів;
- спрощення розвитку окремих підсистем;
- зниження навантаження на ігровий сервер;
- гнучкість побудови серверної інфраструктури.

Окремою перевагою є ізоляція компонентів системи. У випадку тимчасової недоступності одного сервісу інші частини інфраструктури можуть продовжувати роботу незалежно від нього. Це дозволяє підвищити загальну надійність платформи.

Водночас подібний підхід значно ускладнює загальну структуру системи. Для забезпечення коректної взаємодії між сервісами необхідно реалізовувати механізми передачі повідомлень, синхронізації даних та контролю стану підключених компонентів.

До основних недоліків мікросервісної архітектури належать:

- складність побудови та підтримки інфраструктури;
- необхідність організації стабільної мережевої взаємодії;
- затримки під час передачі даних між сервісами;
- ускладнення адміністрування та моніторингу системи;
- підвищена складність DevOps-підтримки;
- необхідність забезпечення узгодженості даних між компонентами.

Таким чином, мікросервісна архітектура є більш складною з точки зору реалізації, однак вона забезпечує значно вищу гнучкість та масштабованість у порівнянні з класичним монолітним підходом, що особливо важливо для сучасних багатокористувацьких платформ із розвиненою серверною інфраструктурою.

1.2.3 Обраний архітектурний підхід

У межах кваліфікаційної роботи використовується комбінований архітектурний підхід, що поєднує монолітну основу ігрового сервера та розподілену інфраструктуру зовнішніх сервісів.

Основою системи виступає серверна платформа Paper, яка забезпечує виконання всієї ігрової логіки за допомогою плагінів (Paper API). Саме цей рівень відповідає за обробку ігрових подій, взаємодію між гравцями, реалізацію ігрових механік та підтримку стану ігрового світу.

Окремо від ігрового ядра реалізовано набір зовнішніх сервісів, які забезпечують інфраструктурну підтримку системи. Вони відповідають за зберігання та обробку користувацьких даних, авторизацію, взаємодію з веб-системою, а також передачу інформації між компонентами через мережевий рівень.

Такий підхід дозволяє розділити ігрову логіку та сервісну інфраструктуру, що підвищує гнучкість системи, спрощує її масштабування та забезпечує можливість незалежного розвитку окремих компонентів.

1.3 Аналіз серверних ядер Minecraft

У процесі розробки багатокористувацьких ігрових систем на базі Minecraft одним із ключових факторів є вибір серверного ядра, яке визначає продуктивність, стабільність роботи та можливості розширення функціоналу. Серверне ядро є базовою програмною реалізацією ігрового сервера, що відповідає за обробку ігрових подій, синхронізацію стану світу, взаємодію з клієнтами та виконання ігрової логіки.

Сучасна екосистема Minecraft містить декілька основних серверних реалізацій, які відрізняються між собою архітектурою, рівнем оптимізації та підходами до розширення функціональності. Найбільш поширеними серед них є Spigot, Paper, а також різні похідні рішення, що базуються на них або розширюють їх можливості.

Окремою особливістю серверних ядер Minecraft є їхня тісна інтеграція із системою плагінів, яка дозволяє значно розширювати стандартний функціонал гри. Саме завдяки підтримці серверних API розробники мають можливість реалізовувати власні механіки, системи економіки, багатокористувацькі режими, внутрішньоігрові меню, засоби модерації та інші компоненти серверної інфраструктури.

Фактично серверне ядро виступає не лише як середовище виконання гри, а і як програмна платформа для створення складних багатокористувацьких систем. Від його можливостей напряму залежить стабільність обробки ігрових подій, швидкість взаємодії між компонентами, а також рівень підтримки сучасних механізмів оптимізації.

У контексті високонавантажених серверів вибір серверного ядра набуває особливого значення, оскільки навіть незначні затримки обробки ігрового циклу можуть суттєво впливати на загальний користувацький досвід. Саме тому сучасні серверні платформи приділяють значну увагу оптимізації використання процесорних ресурсів, роботі з мережею та зменшенню навантаження на систему обробки світу.

Також важливим фактором є рівень сумісності серверного ядра з існуючою екосистемою плагінів. Для багатьох серверних проєктів критично важливо мати можливість використовувати вже наявні програмні рішення або інтегрувати власні

модулі без необхідності модифікації базового серверного коду.

1.3.1 Spigot як базова серверна платформа

Spigot є однією з найперших широко використовуваних модифікацій офіційного Minecraft сервера, яка була створена з метою покращення продуктивності та додавання можливості використання плагінів. Ця платформа стала фундаментом для створення багатьох серверних рішень і фактично визначила стандарт плагінної архітектури в Minecraft.

Основними характеристиками Spigot є:

- підтримка плагінної системи через API;
- базова оптимізація ігрового циклу;
- сумісність з великою кількістю існуючих розширень;
- використання класичної монолітної архітектури.

Перевагами Spigot є його простота, стабільність та популярність серед розробників. Через це існує значна кількість готових плагінів та документації.

Водночас ядро має важливі недоліки. До них відноситься низький рівень оптимізації для серверів з високим навантаженням. Такий недолік впливає на зниження продуктивності системи при великій кількості одночасних гравців. Також архітектура Spigot не передбачає сучасних механізмів асинхронної обробки багатьох процесів.

Попри це, Spigot затяжний час залишався основною платформою для створення Minecraft серверів різного масштабу. Численні сучасні серверні ядра та модифікацій прямо базуються саме на ньому або використовують його API як основу для забезпечення сумісності з наявними плагінами.

Важливою перевагою Spigot є його стабільність та передбачуваність у роботі. Для багатьох невеликих серверів цього рівня функціональності є цілком достатньо, особливо у випадках, коли сервер не передбачає великого навантаження або складної інфраструктури.

Також саме завдяки Spigot сформувався сучасний підхід до розробки серверних плагінів Minecraft. Велика частина існуючих бібліотек, документації та прикладів програмного коду орієнтована саме на дану платформу, що значно спрощує процес входження нових розробників у сферу створення серверних рішень.

Разом із цим при побудові масштабованих багатокористувацьких платформ можливостей Spigot часто стає недостатньо. Особливо це помітно у випадках використання великої кількості плагінів, складних систем синхронізації або високого навантаження на серверну частину. Саме ці обмеження стали причиною появи більш оптимізованих серверних ядер нового покоління.

1.3.1 Paper як оптимізоване серверне ядро

Paper є розвитком і вдосконаленням платформи Spigot і на сьогодні розглядається як де-факто стандарт для високонавантажених Minecraft серверів. Основною метою Paper є підвищення продуктивності, оптимізація внутрішніх процесів та зменшення затримок при обробці ігрових подій.

На відміну від базового Spigot, Paper містить значну кількість оптимізацій, які стосуються роботи з чанками, обробки подій, мережевої взаємодії та використання ресурсів процесора.

Основними перевагами Paper є:

- суттєво вища продуктивність у порівнянні зі Spigot;
- оптимізована обробка ігрового тіку;
- покращена робота з асинхронними процесами;
- зменшене навантаження на серверні ресурси;
- повна сумісність зі Spigot API та плагінами.

Окрім фундаментальних оптимізацій, Paper також надає більше підтримки для розробників плагінів. Існує покращений API, який підтримує спрощену обробку ігрових подій, асинхронних запитів та оптимізацію.

Найкраще в Paper це його концентрація на серверах з високим навантаженням.

Значна частина внутрішніх механізмів оновлена, щоб він міг вміщувати багато гравців і складні серверні системи. Це може значно зменшити навантаження на процесор і підвищити стабільність сервера.

Крім того, Paper надає гнучкість у налаштуванні внутрішніх параметрів сервера. Адміністратор може контролювати систему генерації світу, як обробляються сутності, як працює фізика та інші аспекти, щоб сервер можна було налаштувати відповідно до вимог проекту. І ще однією з особливостей гри є те, що ядро сервера активно оновлюється, з виправленими помилками та додатковими оптимізаціями, щоб бути актуальним для сьогоднішнього Minecraft.

Paper набагато більше підходить для використання у великих багатокористувацьких проектах, де стабільність, продуктивність та підтримка складної серверної інфраструктури є ключовими. Він забезпечує кращу продуктивність, сумісний з екосистемою плагінів і пропонує набагато більш гнучку конфігурацію, щоб зробити його ефективним рішенням для сучасної реалізації серверів Minecraft у порівнянні зі Spigot.

1.3.4 Порівняльний аналіз серверних ядер

Для визначення найбільш доцільного рішення було проведено порівняльний аналіз основних серверних ядер Minecraft.

У процесі аналізу враховувались такі критерії:

- продуктивність при високому навантаженні;
- стабільність роботи;
- рівень оптимізації внутрішніх процесів;
- підтримка плагінної архітектури;
- сумісність із наявною екосистемою розширень;
- можливості масштабування серверної інфраструктури;
- активність підтримки та оновлення платформи.

Критерій	Spigot	Paper	Purpur
Продуктивність	середня	висока	висока
Оптимізація	базова	розширена	розширена
Підтримка плагінів	висока	висока	висока
Стабільність	висока	дуже висока	висока
Масштабованість	обмежена	покращена	покращена

Рис. 1.3.1. Аналіз серверних ядер.

Джерело: Розроблено автором

Згідно з результатами, ядро сервера Spigot дозволяє мати адекватний рівень стабільності та сумісності для налаштування малих і середніх серверів. Також, потужність оптимізації все ще відносно низька для ситуацій з великим навантаженням або використанням складної серверної інфраструктури.

У порівнянні, ядро сервера Paper забезпечує набагато вищу продуктивність завдяки оптимізації внутрішніх механізмів обробки світу, мережевої взаємодії та використання серверних ресурсів. Воно абсолютно сумісне зі Spigot API, що дозволяє не змінювати існуючі плагіни або розробляти їх.

Paper також пропонує кращу підтримку асинхронної обробки окремих процесів, що є важливим аспектом для серверів, які можуть також взаємодіяти з зовнішніми сервісами поза ігровим ядром або використовувати іншу інфраструктуру.

Ядро сервера Purpur (похідне від Paper) вводить нові методи налаштування ігрових процесів. Усе це мало покращує базове серверне середовище, лише вносить деякі зміни в ігрову механіку. Ще одним ключовим елементом є активність розробки

платформи.

Зокрема, Paper часто оновлюється, виправляються помилки та оптимізується на дуже частій основі, що забезпечує стабільну роботу та безперервну серверну частину системи.

Аналіз та дослідження в результатах показують, що Paper надає найбільш збалансоване поєднання продуктивності, стабільності, сумісності та гнучкості, щоб забезпечити найкращий компроміс між швидкістю, стабільністю, сумісністю та гнучкістю для впровадження сучасного багатокористувацького сервера Minecraft.

1.3.5 Обґрунтування вибору серверного ядра

На основі проведеного аналізу було визначено, що оптимальним серверним ядром для створення ігрового сервера Minecraft є Paper. Це рішення гарантує високу продуктивність, стабільність під значними навантаженнями та повну сумісність із плагінною екосистемою Spigot API.

Використання Paper забезпечує ефективну організацію виконання ігрової логіки навіть за умов великої кількості одночасних гравців. Оптимізовані механізми обробки подій та взаємодії зі світом позитивно впливають на швидкість системи та підтримують стабільний ігровий процес.

Серед важливих переваг також слід відзначити підтримку сучасних методів розробки серверних розширень. Наявність розширеного API та асинхронних механізмів дозволяє інтегрувати додаткові сервіси й зовнішню інфраструктуру без істотного впливу на продуктивність ядра гри.

Сумісність з численними вже існуючими плагінами та бібліотеками грає суттєву роль, спрощуючи процес розробки, скорочуючи час інтеграції окремих компонентів і забезпечуючи підтримку багатьох готових рішень.

Крім технічних переваг, Paper має активну спільноту розробників та регулярну підтримку, що гарантує його актуальність для новітніх версій Minecraft і швидке усунення виявлених проблем.

Отже, Paper найбільш відповідає вимогам сучасної серверної платформи Minecraft, забезпечуючи стабільну роботу, гнучкість у розвитку та можливість подальшого масштабування всієї системи.

ВИСНОВКИ ДО РОЗДІЛУ 1

У першому розділі було проведено теоретичне дослідження, яке дало можливість побачити властивості серверів Minecraft як особливого типу сучасної багатокористувацької інформаційної системи. На основі аналізу предметної області, протягом років Minecraft еволюціонував від простої ідеї гри до повноцінного цифрового середовища та платформи мережевої взаємодії. Його архітектура є дуже гнучкою, що дозволяє реалізовувати складну ігрову логіку, економічні моделі та індивідуальні сценарії взаємодії користувачів, що підтверджує актуальність цієї теми для практики проектування власної серверної інфраструктури в технічно цікавому завданні, яке потребує інструментів мережевого програмування та системного адміністрування.

Особливу увагу в обговоренні було приділено аналізу ключових архітектурних міркувань для розробки масштабованих ігрових платформ під час дослідження. Класична монолітна архітектура, де все працює в одній операційній моделі, досягаючи високої швидкості внутрішньої системи, відома як монолітна архітектура, але горизонтальне масштабування та модель відмовостійкості мають значні проблеми. Мікросервіс, з іншого боку, надає спосіб перенести завдання інфраструктури в окремі модулі – що може покращити надійність та масштабованість системи. Через це в реалізації цього проекту було обґрунтовано гібридний підхід, що складається з високопродуктивного ігрового ядра для обробки логіки в реальному часі та розподіленої системи зовнішніх сервісів.

Завершальним етапом стало порівняння серверних ядер, де було обрано платформу Paper на основі таких характеристик, як продуктивність, стабільність та сумісність. Краща швидкість обробки ігрового циклу; асинхронна робота; та повна інтеграція з екосистемою плагінів Spigot API обґрунтовують це рішення. Таким чином, встановлена теоретична основа стала передумовою для переходу до фази концептуалізації розподіленої ігрової системи.

РОЗДІЛ 2
КОНЦЕПТУАЛЬНЕ ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Аналіз предметної області

Предметною областю цієї роботи є розробка розподіленої ігрової платформи на основі Minecraft, яка включає ігрові сервери, клієнтські застосунки та набір допоміжних інфраструктурних сервісів. Такі системи належать до класу багатокористувацьких онлайн-платформ реального часу, де ключовим фактором є забезпечення стабільної взаємодії між великою кількістю користувачів у межах спільного ігрового середовища.

У межах предметної області можна виділити кілька основних категорій об’єктів, які беруть участь у функціонуванні системи:

- **Користувачі (гравці)** – кінцеві учасники системи, які взаємодіють через ігровий клієнт;
- **Ігрові сервери Minecraft** – середовище виконання ігрової логіки та обробки подій;
- **Клієнтські застосунки** – програмні засоби доступу до системи (ігровий клієнт та супутні інтерфейси);
- **Інфраструктурні сервіси** – програмні компоненти, що забезпечують підтримку акаунтів, сесій, обміну даними та інших системних функцій;
- **Комунікаційний шар** – механізм передачі даних між усіма компонентами системи.

Особливістю предметної області є робота в умовах високого навантаження та необхідності обробки великої кількості подій у режимі реального часу. Це накладає ряд вимог до системи, серед яких:

- мінімальна затримка передачі даних між компонентами;
- висока пропускна здатність при одночасному підключенні великої кількості користувачів;

- стійкість до відмов окремих частин системи;
- збереження цілісності даних у розподіленому середовищі;
- можливість горизонтального масштабування.

Додатково слід враховувати, що ігрова логіка Minecraft є подієво-орієнтованою та тісно пов’язаною з мережевою взаємодією, що вимагає чіткої синхронізації стану світу між клієнтом і сервером.

У процесі аналізу предметної області також встановлено, що існуючі рішення переважно базуються на монолітній архітектурі ігрового сервера, тоді як інфраструктурні задачі часто реалізуються окремо. Це створює передумови для побудови більш гнучкої розподіленої системи, яка поєднує ігрову та сервісну частини.

2.2 Загальна концепція системи

Загальна концепція розроблюваної системи полягає у створенні розподіленої багатокомпонентної платформи, призначеної для організації багатокористувацької взаємодії в середовищі Minecraft. Система розглядається як набір взаємопов’язаних програмних модулів, що забезпечують узгоджену роботу ігрових серверів, клієнтських застосунків і інфраструктурних сервісів.

Ключовим архітектурним принципом є поділ рішення на логічно автономні компоненти. Кожен із них відповідає за конкретний перелік функцій і взаємодіє з іншими частинами через уніфікований протокол обміну даними. Така структура забезпечує масштабованість, адаптивність і можливість поступового нарощування функціональності без необхідності суттєвих змін у базовому ядрі платформи.

Центральне місце в архітектурі займає комунікаційний вузол, який виступає координаційним рівнем для взаємодії всіх компонентів. Через нього здійснюється обмін структурованими повідомленнями між ігровими серверами та допоміжними сервісами. До його задач належать маршрутизація запитів, синхронізація станів, передавання команд, а також підтримка єдиного формату даних у межах системи.

Ігрові сервери Minecraft є основною виконавчою частиною платформи та відповідають за обробку безпосереднього ігрового процесу. Уся прикладна ігрова логіка реалізується за допомогою плагінів, які розширюють можливості сервера: обробляють ігрові події, забезпечують взаємодію між гравцями, реалізують внутрішньоігрові механіки та правила обраних режимів.

Для включення ігрових серверів до спільної екосистеми використовується спеціальний мережевий модуль, що встановлюється на кожному сервері. Він забезпечує двосторонній зв’язок між ігровим середовищем та інфраструктурною частиною системи, зокрема передачу подій і станів, а також отримання керувальних команд.

Інфраструктурний рівень представлений набором незалежних сервісів, які виконують функції збереження, обробки та адміністрування даних. До основних напрямів їх роботи належать:

- керування обліковими записами користувачів і пов’язаними даними;
- збереження та синхронізація інформації про гравців і їхній прогрес;
- обробка ігрових сесій і контроль станів підключення;
- реалізація механізмів обміну повідомленнями між користувачами;
- керування додатковими даними профілю, зокрема візуальними елементами;
- виконання допоміжних задач, пов’язаних з аналітикою та адміністративними операціями.

Клієнтська частина включає ігровий клієнт та додаткові інтерфейси доступу, що забезпечують взаємодію користувача з платформою. Вони виконують роль точки входу та обмінюються даними із серверними компонентами через стандартизований протокол комунікації.

Загалом архітектуру можна визначити як гібридну розподілену модель, яка поєднує високопродуктивне ігрове середовище з масштабованою інфраструктурою. Це дозволяє підтримувати стабільний ігровий процес, спрощує розширення функцій

та забезпечує ізоляцію окремих компонентів системи.

2.3 Проєктування функціональних компонентів системи

У межах концептуального проєктування розроблювана система розглядається як комплекс взаємопов’язаних програмних компонентів, які забезпечують функціонування багатокористувацької ігрової платформи на основі Minecraft. Основною особливістю системи є поєднання ігрової серверної частини з окремими інфраструктурними компонентами, що забезпечують централізоване управління даними, автентифікацію користувачів, обробку службових запитів та підтримку додаткового функціоналу.

Сучасні багатокористувацькі платформи вимагають не лише реалізації базової ігрової логіки, а й створення повноцінного програмного середовища, у межах якого здійснюється обробка великої кількості одночасних підключень, підтримується узгодженість даних між різними серверами, забезпечується безпечна робота користувачів та реалізується взаємодія між усіма компонентами системи.

На відміну від класичних локальних серверів Minecraft, де більшість функціоналу зосереджується безпосередньо всередині ігрового ядра, у цій системі використовується підхід із логічним розділенням відповідальності між окремими компонентами. Це дозволяє ізолювати інфраструктурну логіку від безпосередньої ігрової взаємодії, що позитивно впливає на масштабованість, стабільність та подальший розвиток системи.

Кожен компонент системи виконує окрему функціональну роль та взаємодіє з іншими компонентами через централізований механізм обміну даними. Такий підхід дозволяє забезпечити узгоджену роботу всієї платформи навіть за умов високого навантаження або великої кількості одночасно підключених користувачів.

Загальна архітектура системи може бути охарактеризована як розподілена клієнт-серверна модель, у межах якої ігрові сервери, допоміжні сервіси та клієнтські застосунки працюють як єдина інформаційна система. Основою взаємодії між компонентами є централізований комунікаційний рівень, який забезпечує передачу

структурованих даних, координацію подій та підтримку узгодженого стану системи.

Такий підхід дозволяє:

- розділити ігрову та інфраструктурну логіку;
- забезпечити незалежність окремих компонентів;
- спростити масштабування системи;
- зменшити вплив відмов окремих частин системи на загальну працездатність платформи;
- забезпечити можливість подальшого розвитку функціоналу без необхідності повної перебудови архітектури.

2.3.1 Проєктування центрального комунікаційного сервісу

Центральний комунікаційний сервіс є одним із найважливіших компонентів системи, оскільки саме через нього здійснюється основна взаємодія між ігровими серверами та інфраструктурною частиною платформи. Даний компонент виконує роль координаційного вузла, який забезпечує централізовану обробку запитів, передачу інформації та підтримку актуального стану даних у межах усієї системи.

Необхідність використання такого компонента пояснюється тим, що багатокористувацька ігрова платформа складається з великої кількості взаємопов’язаних елементів, кожен із яких потребує доступу до спільних даних. До таких даних можуть належати профілі користувачів, ігрова статистика, статуси серверів, інформація про блокування користувачів, параметри доступу та інші службові відомості.

Центральний сервіс забезпечує єдину точку взаємодії між ігровими серверами та системами збереження даних. При цьому він не обмежується лише передачею повідомлень, а виконує безпосередню обробку запитів та отримання необхідної інформації із систем збереження.

Основними функціями компонента є:

- прийом запитів від ігрових серверів;

- отримання необхідних даних із систем збереження;
- передача актуальної інформації назад на ігрові сервери;
- підтримка зв’язку між вузлами системи;
- забезпечення узгодженого формату обміну даними;
- контроль активності підключених серверів;
- координація міжсерверної взаємодії;
- підтримка службових систем платформи.

Окрему роль центральний сервіс відіграє у забезпеченні цілісності інформації між різними серверами Minecraft. Оскільки користувач може взаємодіяти з кількома серверами в межах єдиної платформи, система повинна гарантувати актуальність даних незалежно від конкретного вузла, до якого підключений користувач.

Крім того, через центральний сервіс реалізується частина глобальних функцій платформи, які не повинні залежати від окремого ігрового сервера. До таких функцій належать обмін приватними повідомленнями між користувачами, централізоване блокування або обмеження доступу, отримання інформації про доступні сервери платформи та підтримка службових механізмів керування інфраструктурою.

В результаті, центральний сервіс виконує роль інтеграційного ядра системи, яке забезпечує взаємозв’язок між усіма компонентами платформи та підтримує узгоджену роботу розподіленого ігрового середовища.

2.3.2 Проєктування сервісу управління обліковими записами користувачів

Система управління обліковими записами користувачів є одним із базових компонентів платформи та відповідає за роботу з персональними даними користувачів, їх автентифікацію та контроль доступу до функціоналу системи.

У сучасних багатокористувацьких платформах обліковий запис користувача виступає центральним елементом взаємодії з усією системою. Саме через нього

здійснюється ідентифікація користувача, збереження персональних параметрів, налаштувань та ігрових даних.

Даний компонент забезпечує повний життєвий цикл роботи користувача в системі – від моменту створення облікового запису до подальшого управління параметрами профілю.

До основних функцій системи належать:

- реєстрація нових користувачів;
- автентифікація користувачів;
- завершення активних сесій;
- керування параметрами безпеки;
- зміна та відновлення паролів;
- підтримка механізмів додаткового захисту доступу;
- зміна ігрового зовнішнього вигляду користувача;
- облік внутрішнього балансу;
- підтримка бонусних механізмів та промокодів.

Особливу увагу в межах даного компонента приділено питанням безпеки користувацьких даних. Оскільки система працює в умовах відкритого мережевого середовища, необхідно забезпечити захист персональної інформації користувачів та запобігти несанкціонованому доступу до облікових записів.

Крім того, система профілів забезпечує централізоване зберігання користувацьких даних, які можуть використовуватись іншими компонентами платформи. Це дозволяє підтримувати єдиний профіль користувача незалежно від конкретного сервера або клієнтського застосунку, з яким він взаємодіє.

З огляду на це, система управління обліковими записами формує основу персоналізованої взаємодії користувача з платформою та забезпечує централізоване управління його даними.

2.3.3 Проєктування сервісу роботи з ігровими сесіями

Система роботи з ігровими сесіями забезпечує перевірку та підтвердження права користувача на підключення до ігрового сервера, а також контроль активного стану його входу в систему.

Цей компонент виступає проміжною ланкою між клієнтською частиною та серверною інфраструктурою, забезпечуючи коректну ідентифікацію користувача в межах ігрового середовища.

Основною задачею системи є підтвердження того, що користувач, який намагається підключитися до сервера, дійсно володіє дійсним обліковим записом та має активну сесію, створену в межах системи авторизації.

Основні функції компонента:

- перевірка права користувача на підключення до сервера;
- підтвердження автентичності клієнтського підключення;
- контроль активного стану ігрових сесій;
- отримання та надання профільної інформації користувача;
- забезпечення цілісності процесу входу до гри;
- синхронізація статусу користувача між компонентами системи.

Використання окремої системи керування ігровими сесіями дозволяє відокремити процес автентифікації від безпосередньої ігрової логіки сервера, що позитивно впливає на безпеку, масштабованість та гнучкість усієї платформи.

Крім того, такий підхід забезпечує можливість централізованого управління користувацькими підключеннями та підтримує єдину систему авторизації для всіх компонентів ігрової інфраструктури.

Таким чином, система роботи з ігровими сесіями є важливою частиною механізму безпечного доступу користувачів до платформи та забезпечує коректну інтеграцію клієнтської частини з серверним середовищем.

2.3.4 Проєктування сервісу генерації аватарів користувачів

У межах багатокористувацької ігрової платформи важливу роль відіграє візуальна ідентифікація користувачів. Для забезпечення єдиного стилю представлення профілів у системі використовується окремий сервіс генерації аватарів користувачів, який формує графічне представлення гравця на основі вже існуючих даних про його зовнішній вигляд.

Основною задачею даного сервісу є отримання інформації про зовнішній вигляд персонажа користувача зі сховища скінів та генерація аватарного зображення у момент отримання запиту.

На концептуальному рівні сервіс не виконує функцій постійного збереження графічних файлів або керування сховищем користувацьких ресурсів. Його задача обмежується динамічною обробкою вже наявних даних та поверненням готового результату іншим компонентам системи.

Такий підхід дозволяє:

- уникнути дублювання графічних даних;
- зменшити навантаження на системи збереження;
- забезпечити актуальність аватарів після зміни зовнішнього вигляду користувача;
- спростити архітектуру сервісу;
- мінімізувати обсяг службових ресурсів.

У процесі роботи сервіс отримує дані про зовнішній вигляд персонажа, після чого виконує обробку необхідних елементів скина та формує аватар зображення у визначеному форматі.

До основних функцій сервісу належать:

- отримання даних про зовнішній вигляд користувача зі сховища;
- обробка текстур персонажа;
- генерація аватарного зображення на основі отриманих даних;

- повернення сформованого результату іншим компонентам системи;
- обробка повторних запитів на генерацію.

Окрему роль згаданий сервіс відіграє у взаємодії із системою ігрових сесій. Саме через нього компонент авторизації та обробки сесій отримує інформацію про текстури користувача, необхідні для коректного відображення зовнішнього вигляду персонажа у клієнті Minecraft. Таким чином, сервіс генерації аватарів виступає одним із допоміжних елементів механізму персоналізації користувача в межах усієї платформи.

Важливою особливістю даного компонента є його повна незалежність від процесів збереження користувацьких даних. Сервіс не змінює інформацію про користувача та не виконує збереження результатів генерації, а лише виступає як окремий механізм динамічного формування графічного представлення.

Крім того, винесення такого функціоналу в окремий компонент дозволяє ізолювати операції обробки графічних даних від інших частин системи, що позитивно впливає на стабільність та продуктивність основної серверної інфраструктури.

Отже, сервіс генерації аватарів користувачів забезпечує уніфіковане та динамічне формування графічного представлення профілів без необхідності окремого збереження результатів, що спрощує підтримку системи та забезпечує актуальність візуальних даних користувачів.

2.3.5 Проєктування сервісу електронних повідомлень

У процесі функціонування багатокористувацької платформи виникає необхідність забезпечення стабільної взаємодії з користувачами поза межами ігрового середовища. Для реалізації таких задач у системі передбачено окремий компонент електронних повідомлень, який забезпечує передачу службової інформації користувачам через електронну пошту.

На концептуальному рівні названий компонент виконує роль зовнішнього

комунікаційного механізму, який використовується для підтвердження окремих дій користувача, передачі системних повідомлень та забезпечення додаткового рівня безпеки під час роботи з обліковим записом.

Основною задачею компонента є автоматизоване формування та надсилання повідомлень у відповідь на певні події, що виникають у системі.

До таких подій можуть належати:

- реєстрація нового користувача;
- відновлення доступу до облікового запису;
- підтвердження окремих дій користувача;
- надсилання службових кодів безпеки;
- інформування про зміну параметрів облікового запису;
- повідомлення про важливі системні події.

У структурі системи даний компонент виконує роль зовнішнього комунікаційного механізму електронних повідомлень взаємодіє з іншими частинами системи через механізм передачі службових запитів. Інші компоненти формують подію або запит на надсилання повідомлення, після чого система електронних повідомлень виконує обробку даних та забезпечує доставку листа користувачу.

Такий підхід дозволяє:

- відокремити логіку надсилання повідомлень від основної бізнес-логіки;
- зменшити навантаження на інші компоненти системи;
- централізувати механізми зовнішньої комунікації;
- спростити масштабування системи повідомлень;
- забезпечити незалежність процесів доставки листів;
- уніфікувати формат повідомлень для різних частин платформи.

Основні функції компонента:

- формування електронних повідомлень на основі готових шаблонів;
- підтримка багатомовності повідомлень;

- підстановка необхідних даних у шаблони листів;
- надсилання повідомлень користувачам;
- обробка помилок доставки;
- повторна спроба надсилання повідомлень;
- централізована взаємодія із зовнішніми поштовими сервісами.

Окрему роль система електронних повідомлень відіграє у процесах, пов’язаних із роботою облікових записів користувачів. Через електронну пошту користувач отримує інформацію, необхідну для підтвердження окремих дій, відновлення доступу та взаємодії із системою поза межами ігрового середовища.

Крім того, винесення даного функціоналу в окремий компонент дозволяє забезпечити незалежне адміністрування та модернізацію механізмів взаємодії із зовнішніми поштовими системами без необхідності внесення змін до інших частин платформи.

У підсумку, система електронних повідомлень забезпечує стабільний канал взаємодії між платформою та користувачами, підтримує процеси функціонування облікових записів та підвищує загальну надійність роботи всієї системи.

2.3.6 Проєктування клієнтської частини системи

Клієнтська частина системи представлена модифікованим клієнтом Minecraft, який забезпечує взаємодію користувача з усією ігровою платформою та надає доступ до серверної інфраструктури й додаткових можливостей системи. Саме через клієнтське середовище користувач проходить автентифікацію, підключається до ігрових серверів, взаємодіє з ігровим середовищем та використовує функціональні можливості платформи.

Клієнтська частина розглядається як окремий компонент системи, який поєднує стандартні механізми роботи Minecraft із додатковими розширеннями, необхідними для інтеграції з розробленою серверною інфраструктурою.

Основною задачею клієнтської частини є забезпечення коректної взаємодії

користувача з усіма компонентами системи при збереженні стабільності та сумісності з базовими механізмами гри.

До основних функцій клієнтського компонента належать:

- автентифікація користувача в системі;
- взаємодія із системою ігрових сесій;
- отримання даних користувача;
- відображення індивідуального зовнішнього вигляду персонажа;
- забезпечення підключення до ігрових серверів;
- підтримка додаткових параметрів взаємодії;
- розширення стандартного функціоналу гри.

Особливістю цього компонента є інтеграція додаткових механізмів роботи з користувацькими даними та системою зовнішнього вигляду персонажів. Завдяки цьому забезпечується використання централізованих даних платформи замість стандартних механізмів отримання інформації із зовнішніх джерел.

Клієнтська частина взаємодіє із системою ігрових сесій для підтвердження автентичності користувача, а також використовує централізоване сховище текстур персонажів для отримання індивідуального зовнішнього вигляду гравця. Це дозволяє забезпечити єдину систему персоналізації користувачів у межах усієї платформи.

Крім того, клієнтська частина підтримує додаткові параметри взаємодії користувача з грою, які дозволяють розширити стандартний набір можливостей ігрового середовища без зміни базових механізмів роботи Minecraft. До таких можливостей можуть належати додаткові налаштування поведінки персонажа, допоміжні механізми керування та інші елементи персоналізації ігрового процесу.

У загальній архітектурі системи клієнт виступає не лише як програма для запуску гри, а як окрема частина єдиної екосистеми, що забезпечує:

- єдину систему авторизації;
- централізовану взаємодію із сервісами платформи;

- узгоджене використання користувацьких даних;
- інтеграцію додаткових функціональних можливостей;
- підтримку розширеного користувацького досвіду.

Важливою перевагою такого підходу є можливість поступового розвитку клієнтської частини без необхідності модифікації серверної інфраструктури. Це дозволяє незалежно вдосконалювати користувацький інтерфейс, механізми взаємодії та допоміжний функціонал системи.

Отже, клієнтська частина системи забезпечує цілісну взаємодію користувача з платформою, поєднує ігрові та інфраструктурні можливості системи та виступає ключовим елементом доступу до багатокористувацького середовища.

2.3.7 Проєктування лаунчера клієнтської частини

Лаунчер клієнтської частини є окремим програмним компонентом платформи, який забезпечує користувачу централізований доступ до ігрового середовища, а також виконує функції підготовки та запуску модифікованого клієнта Minecraft. У загальній структурі системи він виступає як основна точка входу користувача в екосистему проєкту та об’єднує в собі механізми авторизації, конфігурації та персоналізації клієнтського середовища.

З точки зору архітектурного проєктування лаунчер виконує роль проміжного шару між користувачем і серверною інфраструктурою, забезпечуючи узгоджену взаємодію з іншими компонентами системи та централізоване керування параметрами запуску клієнта.

Основною задачею лаунчера є організація контрольованого запуску ігрового клієнта з урахуванням користувацьких налаштувань, параметрів системи та даних, отриманих із серверної частини платформи.

До основних функцій лаунчера належать:

- автентифікація користувача через систему облікових записів;
- підтримка роботи з кількома обліковими записами на одному пристрої;

- отримання та відображення аватарів користувачів у інтерфейсі лаунчера;
- запуск клієнтської частини з попередньо заданими параметрами;
- керування розміром вікна гри та режимом відображення (віконний/повноекранний);
- налаштування автоматичного підключення до ігрового сервера після запуску;
- керування обсягом оперативної пам’яті, що виділяється для клієнта;
- передача додаткових параметрів запуску JVM для оптимізації роботи;
- вибір мови інтерфейсу користувача;
- збереження локальних конфігурацій та користувацьких профілів запуску.

Важливою складовою функціонування лаунчера є його інтеграція з іншими частинами платформи. Під час процесу авторизації він взаємодіє з системою облікових записів, що забезпечує перевірку користувача та отримання необхідних даних профілю. У свою чергу, для відображення індивідуального вигляду користувача використовується сервіс аватарів, який надає згенеровані зображення на основі збережених текстур.

Окрему увагу приділено збереженню користувацьких налаштувань, що дозволяє кожному профілю мати власну конфігурацію запуску клієнта. Такий підхід забезпечує гнучкість використання системи та дозволяє адаптувати ігрове середовище під індивідуальні потреби користувача без необхідності повторного налаштування параметрів.

У загальній архітектурі платформи лаунчер виконує не лише роль інструмента запуску гри, а й виступає інтеграційним елементом, що поєднує користувацький інтерфейс із серверною інфраструктурою. Завдяки цьому досягається централізація процесу входу в систему та підвищується зручність взаємодії користувача з усією екосистемою.

Таким чином, лаунчер забезпечує комплексний механізм доступу до ігрової платформи, об’єднує процеси авторизації, конфігурації та запуску клієнта, а також

створює єдину точку входу до розподіленої ігрової системи.

2.3.8 Проєктування мережевого плагіна інтеграції ігрових серверів

Мережевий плагін є необхідним елементом ігрових серверів і виконує функцію проміжного програмного рівня, який забезпечує зв’язок між ігровим середовищем та центральною інфраструктурою платформи. Його головне призначення полягає в організації стабільного обміну даними між сервером Minecraft і центральним комунікаційним сервісом, а також у обробці службових повідомлень, що надходять від інфраструктурної частини системи.

У межах архітектури платформи плагін встановлюється на кожному ігровому сервері та виступає єдиною точкою інтеграції з центральною системою. Через нього відбувається передача запитів, отримання актуальної інформації та виконання адміністративних і системних операцій.

Основне завдання мережевого плагіна полягає в забезпеченні двосторонньої взаємодії між ігровим сервером і центральною інфраструктурою, включаючи отримання даних і виконання команд, що формуються на стороні серверних сервісів.

До ключових функцій мережевого плагіна належать:

- отримання та обробка службових пакетів від центрального комунікаційного сервісу;
- передача даних про стан гравців до інфраструктурної частини системи;
- отримання та застосування інформації про користувача (профіль, баланс, ранги);
- обробка системних повідомлень, зокрема тих, що стосуються приватної взаємодії між гравцями;
- синхронізація даних про адміністративні обмеження (блокування, мут, попередження);
- отримання оновленого списку ігрових серверів та їхнього статусу;
- забезпечення взаємодії між ігровими режимами та центральною

платформою.

Окремим напрямом роботи плагіна є обробка адміністративних команд, що виконуються безпосередньо на ігрових серверах із подальшою синхронізацією з центральною системою. Для цього реалізовано набір внутрішніх серверних команд, які дозволяють адміністрації здійснювати вплив на користувачів.

До таких команд належать:

- блокування акаунта користувача (бан);
- обмеження можливості спілкування (мут);
- інші адміністративні дії, пов’язані з контролем поведінки гравців.

Після виконання цих команд відповідні дані передаються до центрального сервісу для узгодження стану користувача в межах усієї платформи.

Також плагін забезпечує актуальність інформації під час гри. Усі зміни профілю користувача, включаючи оновлення рангу, статусу або обмежень доступу, оперативно надходять із центральної системи та застосовуються на ігровому сервері без необхідності його перезапуску.

Таким чином, мережевий плагін є критично важливим інтеграційним компонентом, який забезпечує зв’язок між ігровою логікою та інфраструктурними сервісами платформи, підтримує актуальність даних у реальному часі та гарантує узгодженість стану користувачів у межах усієї системи.

2.3.9 Проєктування ігрового режиму GunGame

Ігровий режим GunGame є окремим функціональним модулем ігрової платформи, який реалізує динамічний змагальний сценарій у форматі «гонки озброєнь». Загальна логіка режиму базується на поступовому розвитку гравця через систему рівнів, де кожен наступний рівень відкриває доступ до нової зброї. Основною метою є досягнення максимального рівня швидше за інших учасників матчу.

У межах системи режим GunGame функціонує як самостійний ігровий компонент, який не потребує постійної взаємодії з центральним комунікаційним сервісом. Водночас у разі необхідності він може використовувати інтерфейси мережевого плагіна для отримання або оновлення супутньої інформації про гравця, що забезпечує узгодженість даних у межах платформи.

Основою ігрового процесу є система прогресії, яка визначає розвиток гравця під час матчу. Кожен учасник починає гру з початкового рівня та отримує нову зброю після виконання умов підвищення рівня. Перехід між рівнями здійснюється автоматично на основі нарахування ігрових очок.

Основні функції системи GunGame включають:

- організацію ігрового матчу у форматі змагання між гравцями;
- реалізацію системи рівнів прогресії;
- нарахування ігрових очок за результативні дії;
- зміну доступної зброї залежно від рівня гравця;
- визначення переможця матчу;
- керування ігровими аренами;
- контроль кількості учасників на кожній арені;
- забезпечення можливості приєднання гравців до доступних арен.

Ігрові арени є окремими ізольованими просторами, кожен з яких підтримує обмежену кількість учасників, що становить до двадцяти гравців. Це дозволяє забезпечити стабільність ігрового процесу та рівномірний розподіл навантаження між матчами.

Механізм приєднання до арени реалізовано таким чином, що гравець може вільно увійти до доступної гри за умови наявності вільних слотів. Це забезпечує динамічність системи та дозволяє уникнути тривалого очікування перед початком матчу.

Переможцем у межах одного матчу визначається гравець, який першим

досягає максимального рівня прогресії. Після цього матч завершується, а результати фіксуються системою для подальшого використання у статистиці або рейтингових механізмах.

В результаті, ігровий режим GunGame є самодостатнім модулем, який забезпечує реалізацію швидких змагальних матчів із чітко визначеною системою прогресії, підтримує динамічну взаємодію між гравцями та організовує ігровий процес у межах ізольованих арен з обмеженою кількістю учасників.

2.3.10 Проєктування системи авторизації на основі OAuth 2.0

У процесі функціонування сучасних багатокомпонентних платформ важливу роль відіграє централізована система авторизації, яка забезпечує контроль доступу користувачів до окремих компонентів інфраструктури. У межах розроблюваної системи для реалізації таких механізмів використовується протокол OAuth 2.0, який виступає основою взаємодії між користувачем, клієнтськими компонентами та серверною частиною платформи.

OAuth 2.0 є стандартизованим протоколом авторизації, основною задачею якого є надання обмеженого доступу до ресурсів системи без необхідності передачі користувацького пароля між різними компонентами. На відміну від класичних механізмів автентифікації, де кожен сервіс окремо працює з обліковими даними користувача, використання OAuth 2.0 дозволяє централізувати процес підтвердження особи та забезпечити єдину систему доступу для всієї платформи.

Основною ідеєю протоколу є використання спеціальних токенів доступу, які видаються сервером авторизації після успішного підтвердження особи користувача. Надалі саме ці токени використовуються клієнтськими компонентами для доступу до ресурсів системи без повторної передачі пароля.

У межах реалізованої платформи використовується власний сервер авторизації, який забезпечує:

- автентифікацію користувачів;

- видачу токенів доступу;
- контроль активних сесій;
- перевірку прав доступу;
- централізоване управління авторизацією;
- підтримку механізмів оновлення токенів.

Використання OAuth 2.0 дозволяє:

- уникнути дублювання механізмів входу в різних компонентах системи;
- підвищити рівень безпеки користувацьких даних;
- централізувати управління активними сесіями;
- спростити контроль доступу до сервісів;
- забезпечити єдину систему автентифікації для платформи;
- зменшити ризик несанкціонованого доступу до ресурсів.

Окремою перевагою є розділення процесів автентифікації та бізнес-логіки окремих компонентів системи. Завдяки цьому клієнтські та серверні модулі не працюють напряму з обліковими даними користувача, а використовують централізований механізм підтвердження особи через сервер авторизації.

Крім того, такий підхід дозволяє у майбутньому розширювати систему та підключати нові компоненти платформи до єдиної інфраструктури авторизації без необхідності реалізації окремих механізмів входу для кожного сервісу.

Таким чином, система авторизації на основі OAuth 2.0 забезпечує централізований контроль доступу до ресурсів платформи, підтримує безпечну взаємодію між компонентами системи та формує єдиний механізм автентифікації користувачів у межах розподіленої ігрової інфраструктури.

ВИСНОВКИ ДО РОЗДІЛУ 2

Другий розділ роботи присвячено концептуальному проєктуванню розподіленої системи, де основним об’єктом дослідження виступив ігровий сервер Minecraft як складна інформаційна платформа для організації багатокористувацької взаємодії. Процес проєктування базувався на принципі логічного розділення відповідальності між окремими компонентами, що дозволило ізолювати інфраструктурну логіку від безпосередньої ігрової взаємодії. Ключовим елементом системи визначено центральний комунікаційний сервіс, який виконує роль координаційного вузла та забезпечує маршрутизацію запитів і підтримку цілісності даних про гравців у межах усієї розподіленої мережі.

У ході деталізації інфраструктурної частини системи було спроєктовано набір незалежних сервісів, які забезпечують виконання критичних функцій платформи. Зокрема, розроблено концепцію сервісів управління обліковими записами та ігровими сесіями, що відповідають за безпечну автентифікацію, контроль доступу та синхронізацію статусів користувачів. Додатково було спроєктовано сервіс динамічної генерації аватарів для візуальної ідентифікації профілів та систему електронних повідомлень, яка реалізує автоматизований канал зв’язку з користувачами через електронну пошту для підтвердження реєстрації та відновлення доступу.

Окрему увагу було приділено проєктуванню клієнтської частини та механізмів інтеграції. Створена концепція охоплює модифікований клієнт Minecraft та спеціалізований лаунчер, який слугує єдиною точкою входу, забезпечуючи автентифікацію та конфігурацію параметрів запуску ігрового середовища. Для забезпечення зв’язку ігрового ядра з інфраструктурою було розроблено архітектуру мережевого плагіна, який встановлюється на кожному сервері та забезпечує синхронізацію балансу, рангів і адміністративних обмежень у реальному часі.

На завершення розділу було представлено опис ігрового режиму GunGame як самостійного функціонального модуля, що реалізує змагальний сценарій із власною

системою прогресії та ізольованими аренами. Запропонована архітектурна модель дозволяє забезпечити високу масштабованість, незалежність модулів та стабільну роботу платформи за умов високого навантаження. Спроєктована структура є готовим і обґрунтованим підґрунтям для подальшої технічної реалізації системи та її програмного забезпечення.

РОЗДІЛ 3
ТЕХНІЧНА РЕАЛІЗАЦІЯ І ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

3.1 Обґрунтування вибору інструментальних засобів та технологій розробки

Підбір технологічного стеку для реалізації системи здійснювався з урахуванням вимог до високонавантажених розподілених платформ, характерних для ігрових проєктів. Основними критеріями вибору виступали продуктивність, стабільність роботи, можливість горизонтального масштабування, простота інтеграції між сервісами, а також підтримка сучасних підходів до DevOps та автоматизації процесів розробки.

Окрему увагу було приділено тому, що система повинна ефективно працювати як у режимі реального часу (ігрові події), так і в асинхронному режимі (обробка даних, повідомлення, аналітика). Саме тому технології обирались з урахуванням їх ролі в загальній архітектурі.

3.1.1 Мови програмування та базові технології реалізації

Фундаментальним технологічним вибором для реалізації ігрової логіки на базі ядра Paper стала мова програмування Java 21, яка є актуальним стандартом у сучасній екосистемі розробки Minecraft-серверів. Її використання зумовлене як стабільністю платформи, так і наявністю сучасних механізмів оптимізації виконання програмного коду.

Особливу роль у виборі Java 21 відіграло впровадження віртуальних потоків (Project Loom), що дозволяють ефективно обробляти тисячі паралельних задач із мінімальним навантаженням на системні ресурси. Це має критичне значення для підтримки стабільного ігрового циклу, особливо в умовах високої кількості одночасно підключених користувачів.

Для автоматизації процесів збирання, керування залежностями та структурування проєкту використано Gradle із Kotlin DSL. Такий підхід забезпечує

не лише високу швидкість компіляції, але й більш декларативний та гнучкий опис інфраструктури проєкту, що спрощує його супровід і масштабування.

Паралельно з Java для реалізації інфраструктурних сервісів застосовується мова програмування Go, яка добре зарекомендувала себе у сфері високонавантажених мережевих застосунків. Її ефективна модель конкурентності та низькі затримки роблять її доцільною для побудови мікросервісної архітектури.

Для реалізації API та веб-сервісів використовується фреймворк Gofiber, який базується на високопродуктивному рушії Fasthttp. Це дозволяє досягати мінімальних затримок обробки запитів, що є важливим для сервісів авторизації, профілів користувачів та генерації аватарів у реальному часі.

Уніфікація взаємодії між сервісами, реалізованими на Go та Java, забезпечується за допомогою Protocol Buffers. Використання цього підходу гарантує компактну бінарну серіалізацію даних, чітко визначені контракти між сервісами та значне зменшення мережевого трафіку порівняно з текстовими форматами обміну даними, такими як JSON.

3.1.2 Системи керування даними та хмарна інфраструктура

Ключовим компонентом зберігання структурованих даних виступає реляційна база даних PostgreSQL. Вона забезпечує високий рівень надійності, підтримку транзакційності відповідно до принципів ACID та ефективну роботу зі складними зв’язками між сутностями, зокрема обліковими записами, статистикою та ігровими даними користувачів.

Для зберігання неструктурованого контенту, такого як скіни та аватари гравців, інтегровано об’єктне сховище Cloudflare R2. Вибір цього рішення обґрунтований його сумісністю з S3 API та відсутністю плати за вихідний трафік, що є суттєвою перевагою в умовах великої кількості запитів від клієнтської частини системи.

Окрему роль у забезпеченні користувацької взаємодії поза ігровим середовищем відіграє сервіс Resend, який використовується для надсилання

транзакційних електронних листів. Його впровадження дозволяє ефективно організувати процеси реєстрації, підтвердження облікових записів та відновлення доступу, забезпечуючи при цьому високу доставлюваність повідомлень і стабільну репутацію відправника.

3.1.3. Клієнтська частина та користувацькі інтерфейси

Розробка клієнтської частини системи, включаючи ігровий лаунчер, реалізована на основі фреймворку Wails, який поєднує можливості мови Go з сучасними веб-технологіями. Такий підхід дозволяє створювати настільні застосунки з високою продуктивністю та гнучким інтерфейсом.

Інтерфейсна складова побудована із використанням React 19 та TypeScript, що забезпечує типобезпеку, структурованість коду та зменшення кількості помилок на етапі виконання. Це особливо важливо для складних сценаріїв авторизації, конфігурації клієнта та управління параметрами JVM.

Для стилізації інтерфейсів застосовано Tailwind CSS у поєднанні з компонентною бібліотекою shadcn/ui. Це дозволяє створювати сучасні, адаптивні та доступні інтерфейси, що відповідають вимогам UX/UI-дизайну та стандартам веб-доступності WCAG 2.2.

Додатково для оптимізації процесу розробки використовується інструмент BiomeJS, який поєднує функції лінтера та форматера коду, забезпечуючи єдність стилю кодування в команді. Локалізація платформи реалізована через i18next, що дозволяє динамічно змінювати мову інтерфейсу без перезавантаження застосунку.

3.1.3 Контейнеризація, DevOps та стратегії безпеки

Забезпечення ізоляції та стабільності сервісів реалізується за допомогою технологій контейнеризації Docker та Docker Compose. Це дозволяє формалізувати інфраструктуру як код та забезпечити відтворюваність середовищ на всіх етапах життєвого циклу розробки.

Процеси CI/CD реалізовано на базі GitHub Actions. Додатково використовується статичний аналіз коду golangci-lint, що дозволяє виявляти потенційні помилки та підвищувати рівень безпеки бекенд-сервісів ще на етапі розробки.

Для зберігання приватних Docker-образів та артефактів використовується self-hosted реєстр Zot, що підвищує контроль над інфраструктурою та зменшує залежність від зовнішніх сервісів.

Операційне управління контейнерною інфраструктурою здійснюється через Portainer, який надає зручні інструменти моніторингу стану сервісів, журналів подій та загального стану системи в реальному часі.

Мережева безпека базується на використанні Cloudflare Tunnel у поєднанні з реверс-проксі сервером Caddy. Така архітектура дозволяє реалізувати концепцію «закритої мережі», за якої всі зовнішні порти залишаються недоступними напряму, а доступ до сервісів здійснюється виключно через захищені тунелі з автоматичною видачею TLS-сертифікатів. Це суттєво підвищує загальний рівень захисту системи та знижує ризики мережевих атак, зокрема DDoS.

3.2 Програмна реалізація комунікаційного сервісу

3.2.1 Загальна характеристика та призначення сервісу

Комунікаційний сервіс є одним із ключових компонентів серверної інфраструктури та забезпечує взаємодію між окремими ігровими серверами. Основним призначенням сервісу є централізована передача службових повідомлень, синхронізація стану користувачів між серверами та підтримка постійного мережевого з’єднання між компонентами системи.

У межах реалізованої архітектури сервіс функціонує як окремий TCP-вузол, до якого підключаються Minecraft-сервери для обміну внутрішніми пакетами. Передача даних виконується через Protocol Buffers, що дозволяє забезпечити компактну серіалізацію повідомлень та уніфікований формат мережевої взаємодії.

3.2.2 Структура проєкту

Структура проєкту організована відповідно до принципів модульної побудови застосунків мовою Go. Основні компоненти системи згруповані за функціональним призначенням, що спрощує підтримку коду, масштабування та подальший розвиток сервісу.

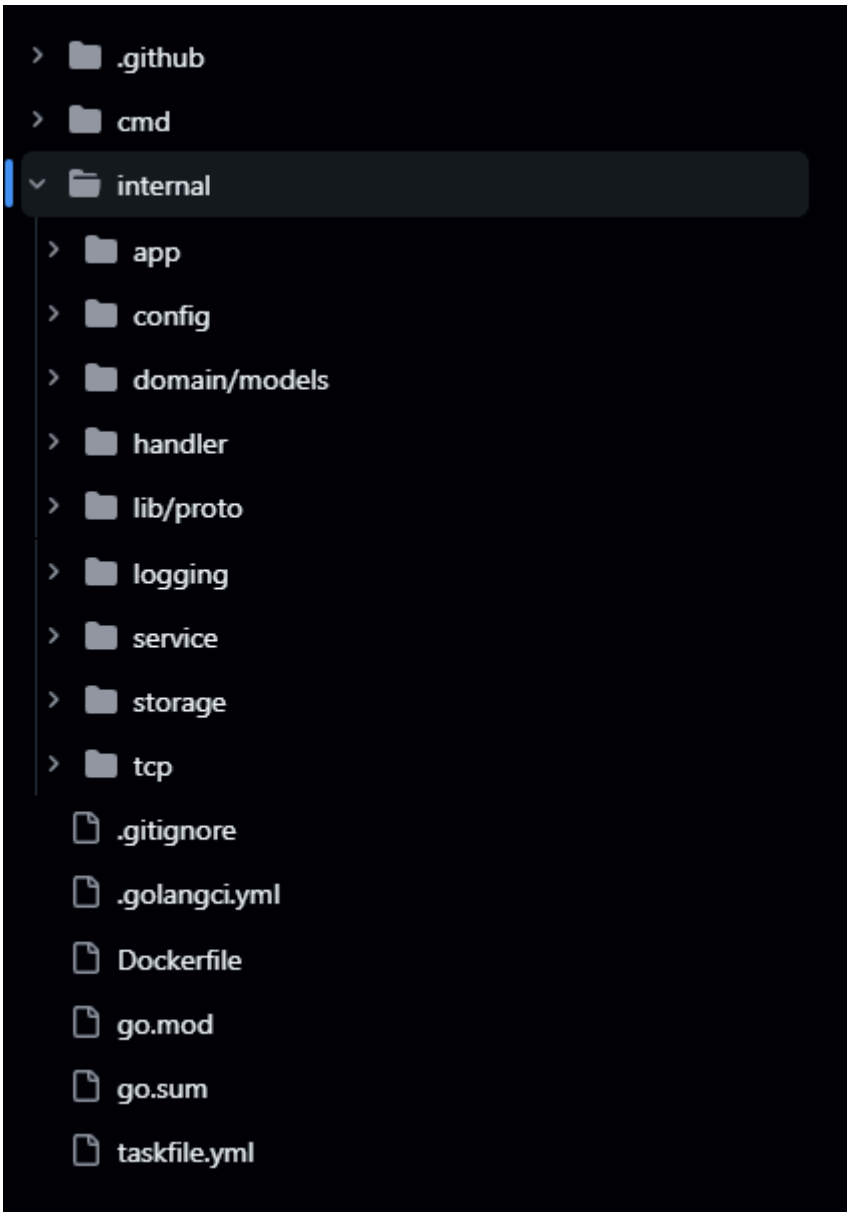

Рис. 3.2.1. Структура проєкту комунікаційного сервісу.

Джерело: Розроблено автором

Основні директорії проєкту мають таке призначення:

- **cmd/** – точка входу до застосунку; містить ініціалізацію сервісу, запуск

основного застосунку та реєстрацію внутрішніх модулів системи. У цьому модулі виконується створення контейнера залежностей, підключення конфігурації та запуск життєвого циклу сервісу.

- **internal/app/** – модуль ініціалізації застосунку та керування життєвим циклом сервісу. У цьому компоненті виконується реєстрація внутрішніх модулів системи, підключення TCP-сервера, конфігурація логування та обробка процедур запуску і завершення роботи сервісу.
- **internal/config/** – модуль завантаження та обробки конфігураційних параметрів. Містить налаштування мережевих з’єднань, параметри бази даних, конфігурацію середовища виконання та службові параметри системи.
- **internal/domain/models/** – внутрішні моделі даних, що використовуються у процесі роботи сервісу. Моделі описують основні сутності системи та застосовуються під час обробки мережевих пакетів і взаємодії між сервісними компонентами.
- **internal/handler/** – обробники мережевих пакетів. Кожен handler відповідає за окремий тип повідомлень та виконує десеріалізацію, первинну валідацію і передачу даних до сервісного шару.
- **internal/service/** – реалізація бізнес-логіки сервісу. У цьому модулі виконуються операції синхронізації серверів, управління підключеннями користувачів, маршрутизація повідомлень та обробка внутрішніх подій системи.
- **internal/storage/** – модуль взаємодії зі сховищем даних. Відповідає за виконання операцій читання, запису та оновлення інформації у базі даних.
- **internal/tcp/** – реалізація TCP-сервера та управління мережевими підключеннями. Модуль забезпечує приймання пакетів, підтримку активних TCP-з’єднань, обробку мережевих подій та передачу повідомлень між вузлами системи.
- **internal/logging/** – модуль централізованого логування. Забезпечує

фіксацію службових подій, помилок та мережевої активності системи. Підтримує різні рівні логування залежно від середовища виконання та використовується у всіх ключових компонентах сервісу для контролю стану роботи та діагностики.

- **internal/lib/proto/** – реалізує серіалізацію та десеріалізацію мережевих пакетів для TCP-взаємодії між сервісами, включаючи читання довжини повідомлення, перевірку коректності розміру та декодування байтів у структури даних, а також формування вихідних пакетів для передачі.

3.2.3 Робота TCP-з’єднання та механізм автентифікації

Після запуску сервіс відкриває TCP-порт та переходить у режим очікування підключень від серверних вузлів. Під час встановлення нового з’єднання сервер виконує процедуру первинної авторизації через пакет handshake.

Окрім базових параметрів підключення, у handshake-пакеті також передається секретний ключ, який використовується для перевірки автентичності вузла та запобігання несанкціонованим підключенням.

Обробка handshake-пакета виконується окремим handler-компонентом, який перевіряє коректність переданих даних, виконує валідацію секретного ключа та у разі успішної перевірки реєструє сервер у внутрішній системі активних вузлів. У випадку невідповідності секрету з’єднання відхиляється та закривається.

Після успішної авторизації між сервером та комунікаційним сервісом підтримується постійне TCP-з’єднання. Для контролю його актуальності використовується механізм keep alive.

Принцип роботи keep alive полягає у періодичному обміні службовими пакетами між сервером та комунікаційним вузлом. Кожен пакет містить nonce-значення, яке дозволяє підтвердити актуальність поточного підключення та виявити втрату мережевого з’єднання.

3.2.4 Синхронізація стану користувачів

Для синхронізації інформації між серверами про підключення користувача використовується пакет player_connect. Після входу користувача на сервер відповідний Minecraft-плагін формує службовий пакет та передає його до комунікаційного сервісу.

Отриманий пакет обробляється відповідним handler-компонентом, після чого інформація про користувача реєструється у внутрішньому стані системи та може бути використана іншими серверами мережі.

Аналогічним чином реалізовано механізм обробки відключення користувача. Після виходу користувача сервер надсилає пакет player_disconnect до комунікаційного сервісу, який оновлює інформацію про активний стан користувача.

3.2.5 Маршрутизація повідомлень між серверами

Для організації приватних повідомлень між серверами використовується окремий пакет private_message. Після отримання повідомлення сервіс визначає сервер, на якому перебуває цільовий користувач, та виконує маршрутизацію пакета до відповідного вузла.

Загальна схема обробки пакетів побудована за принципом маршрутизації повідомлень через handler-компоненти. Після отримання TCP-пакета виконується його десеріалізація, визначення типу повідомлення та передача у відповідний модуль обробки.

3.3 Специфікація структур даних Protocol Buffers

Для забезпечення уніфікованого та високопродуктивного обміну даними між розподіленими компонентами системи було розроблено набір схем серіалізації на базі Protocol Buffers. Вибір даної технології обумовлений потребою в жорсткій типізації контрактів між сервісами на Go та ігровими серверами на Java, а також необхідністю мінімізації розміру мережевих пакетів для зменшення затримок у

реальному часі.

Ключові структури даних протоколу мають таке призначення:

- **Packet** – основний контейнерний об'єкт (wrapper), що інкапсулює всі типи повідомлень у системі. Він містить унікальний ідентифікатор (id), версію схеми (schema_version) для підтримки зворотної сумісності, тип повідомлення (type) та мітку часу відправки (sent_at). Використання конструкції oneof payload дозволяє передавати специфічні дані (наприклад, пакети авторизації чи ігрові події) у межах єдиної структури, що спрощує логіку маршрутизації на рівні TCP-сервера.
- **HandshakePacket** – пакет, призначений для проходження процедури первинної авторизації ігрового сервера в комунікаційній мережі. Структура містить ідентифікатор вузла (server_id) та секретний токен автентифікації (secret). У відповідь система надсилає **HandshakeResponsePacket**, де поле status (тип **OperationStatus**) визначає результат перевірки – успішний доступ або відмову через невірні дані чи конфлікт ідентифікаторів.
- **KeepAlivePacket** – службовий пакет для підтримки стабільності TCP-з’єднання та моніторингу активності підключених вузлів. Містить поле nonce, яке використовується для підтвердження актуальності сесії: ігровий сервер повинен отримати та повернути це значення комунікаційному вузлу, що дозволяє вчасно виявляти розриви каналу або деградацію мережевої взаємодії.
- **PlayerConnectPacket** – структура для синхронізації стану підключення гравця до конкретного ігрового сервера. Пакет передає нікнейм гравця (player_name), його мережеву адресу (player_ip) для систем безпеки та ідентифікатор сервера (server_id), до якого відбулося підключення. Поле connected_at фіксує точний час події, що необхідно для коректного ведення ігрової статистики та глобального моніторингу сесій.
- **PlayerDisconnectPacket** – забезпечує оновлення глобального стану

системи у разі завершення ігрової сесії користувачем. Окрім ідентифікатора гравця та сервера, пакет містить перерахунок Reason, який деталізує причину виходу: звичайне завершення гри (QUIT), примусове відключення адміністрацією (KICK), блокування (BANNED) або технічна помилка (ERROR). Це дозволяє інфраструктурним сервісам коректно очищати тимчасові дані сесії та оновлювати статус гравця в реальному часі.

- **PrivateMessagePacket** – реалізує логіку міжсерверної маршрутизації приватних повідомлень між користувачами. Пакет містить імена відправника (sender_name) та отримувача (recipient_name), текстовий зміст (content) та мітку часу sent_at. На основі цих даних комунікаційний сервіс динамічно визначає фізичне розташування отримувача в мережі та ретранслює пакет на відповідний ігровий вузол, забезпечуючи прозору комунікацію незалежно від топології серверів.

3.4 Реалізація системи управління обліковими записами

3.4.1 Загальна реалізація системи облікових записів

Система управління обліковими записами є центральним компонентом платформи та забезпечує повний життєвий цикл користувацьких даних, включаючи реєстрацію, автентифікацію, управління профілем та контроль доступу до ресурсів системи.

У межах архітектури платформи даний компонент виступає єдиною точкою істини для всіх користувацьких даних. Через нього здійснюється перевірка автентичності користувача, управління активними сесіями, а також доступ до персоналізованих даних, таких як ігровий профіль, параметри безпеки та зовнішній вигляд персонажа.

Окрему увагу приділено централізованому зберіганню даних користувача, що дозволяє забезпечити єдиний профіль незалежно від клієнтського середовища або ігрового сервера.

3.4.2 Реалізація авторизації на основі OAuth 2.0

Реалізація OAuth 2.0 виконана у вигляді набору HTTP-ендпоінтів, які забезпечують повний цикл авторизації користувача: ініціацію запиту, підтвердження доступу, видачу токенів, їх оновлення, відкликання та отримання інформації про користувача.

Основні API-методи:

- **Authorize** – приймає OAuth-параметри запиту (client_id, redirect_uri, response_type, scope, state). Виконує валідацію вхідних даних та формує відповідь для подальшого проходження авторизаційного процесу.
- **Approve** – обробляє підтвердження згоди користувача на доступ. Дані користувача беруться з активної сесії, після чого створюється запис авторизаційного дозволу для конкретного client_id та scope.
- **Token** – виконує обмін authorization code на access token і refresh token. Для перевірки клієнта використовується Basic Auth (client_id + client_secret). Після валідації повертається пара токенів доступу.
- **Refresh** – оновлює access token на основі refresh token. Перевіряється валідність refresh токена та прив’язка до клієнта, після чого видається новий access token.
- **Revoke** – інвалідує access або refresh token. Тип токена визначається через token_type_hint, після чого відповідний запис видаляється або помічається як недійсний.
- **UserInfo** – повертає дані користувача за Bearer token. Виконується парсинг заголовка Authorization, перевірка токена та отримання пов’язаного профілю користувача.

Процес передачі облікових даних клієнта на етапі Token та Refresh реалізовано через HTTP Basic Authentication, де client_id та client_secret декодуються із заголовка Authorization.

Вхідні параметри для кожного етапу проходять базову валідацію на рівні handler-шару, після чого передаються у сервісний шар для бізнес-обробки.

Авторизаційні токени мають розділення на:

- **access token** – для доступу до ресурсів;
- **refresh token** – для оновлення access token без повторної авторизації.

Отримання даних користувача через UserInfo виконується виключно на основі валідного Bearer token, без додаткових параметрів у запиті.

3.4.3 Двофакторна автентифікація

Для підвищення рівня захисту облікових записів у системі реалізовано двофакторну автентифікацію на основі алгоритму TOTP, який забезпечує генерацію одноразових кодів доступу з обмеженим часом дії.

Реалізація механізму виконана з використанням бібліотеки github.com/pquerna/otp, яка надає стандартні засоби для роботи з одноразовими паролями відповідно до RFC 6238.

У межах системи TOTP працює на основі спільного секретного ключа, який генерується під час активації двофакторної автентифікації та асоціюється з конкретним обліковим записом користувача. На його основі клієнтська сторона генерує одноразові коди, що змінюються з фіксованим часовим інтервалом.

Перевірка введеного коду виконується на серверній стороні під час виконання критичних операцій, зокрема входу в систему або зміни чутливих параметрів облікового запису. Система порівнює отримане значення з очікуваним на основі поточного часу та з урахуванням допустимого часового вікна, що дозволяє компенсувати незначні розбіжності синхронізації годинників.

Успішна валідація другого фактора є обов’язковою умовою підтвердження дії користувача та подальшого доступу до захищених ресурсів системи. У випадку невідповідності коду операція автентифікації або запитана дія відхиляється.

3.4.4 Система управління скінами

Зовнішній вигляд персонажа користувача є складовою частиною його профілю та реалізований як окремий модуль персоналізації, що інтегрований у загальну систему управління обліковими записами. Для зберігання і доставки файлів зовнішнього вигляду використовується централізоване об’єктне сховище, сумісне з S3-API (R2), яке забезпечує надійне зберігання медіа-ресурсів та високу доступність даних.

Під час завантаження або оновлення скіна користувачем зображення передається безпосередньо до об’єктного сховища. Після успішного збереження файл отримує унікальну адресу доступу, яка надалі використовується системою для отримання ресурсу. У базі даних при цьому не зберігаються самі файли зображень – фіксується лише метаінформація профілю користувача та посилання на відповідний ресурс у сховищі.

Окрім самого зображення скіна, система зберігає додаткові параметри, які визначають спосіб його відображення в ігровому середовищі. Зокрема, використовується метадані моделі персонажа, що можуть приймати значення:

- **default** – стандартна модель персонажа з класичною геометрією;
- **slim** – альтернативна тонка модель персонажа з модифікованими пропорціями рук.

Ці параметри враховуються під час формування профілю користувача та впливають на спосіб рендерингу персонажа на клієнтській стороні гри. Таким чином, система забезпечує узгоджене відображення зовнішнього вигляду незалежно від того, з якого сервера або клієнта виконується підключення.

Запропонований підхід дозволяє винести роботу з великими файлами у спеціалізоване сховище, зменшити навантаження на основну систему та повністю відмовитися від зберігання бінарних даних у базі даних. У результаті профіль користувача залишається легким з точки зору структури, а управління ресурсами

зовнішнього вигляду стає масштабованим і незалежним від внутрішньої логіки системи.

3.5 Програмна реалізація сервісу ігрових сесій

3.5.1 Загальна реалізація та відповідність Yggdrasil API

Сервіс ігрових сесій реалізовано як HTTP-сервіс на основі Fiber v3, який сумісний із протоколом Yggdrasil Session Server, що використовується Minecraft для перевірки автентичності гравців та отримання профілю користувача.

Система реалізує набір HTTP-ендпоінтів, які відповідають специфікації authlib-injector та забезпечують:

- підтвердження входу користувача на сервер;
- перевірку активної сесії;
- отримання профілю гравця з текстурами;
- підпис даних профілю для клієнта Minecraft.

Додатково реалізовано endpoint метаданих сервера, який повертає інформацію про реалізацію сервісу, доступні домени та публічний ключ підпису.

3.5.2 Реалізація API ендпоінтів

Сервіс надає наступні HTTP-ендпоінти

- **GET /** – повертає метадані сервісу (назва реалізації, версія, підтримувані функції, домени для скіна та публічний ключ підпису);
- **POST /sessionserver/session/minecraft/join** – реєструє факт входу гравця на ігровий сервер;
- **GET /sessionserver/session/minecraft/hasJoined** – перевіряє, чи підключений користувач до сервера;
- **GET /sessionserver/session/minecraft/profile/:uuid** – повертає профіль користувача з інформацією про текстури.

Кожен запит проходить базову валідацію вхідних параметрів, включаючи перевірку access token, serverId та UUID профілю.

3.5.3 Обробка сесій (Join / HasJoined)

Функція Join реалізує механізм реєстрації факту входу користувача на ігровий сервер та є частиною процесу формування активної ігрової сесії. Після отримання запиту виконується валідація вхідних параметрів, включаючи перевірку коректності ідентифікатора профілю та наявності обов'язкових полів запиту. Далі дані передаються до сервісного шару, де здійснюється фіксація сесії та прив'язка користувача до конкретного serverId.

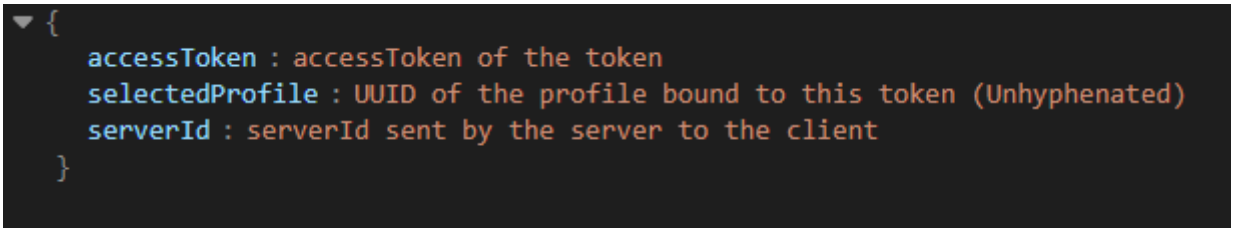

Рис. 3.5.1. Приклад JSON запиту для Join ендпоінта.

Джерело: Розроблено автором

Функція HasJoined використовується для перевірки існування активної сесії користувача на вказаному ігровому сервері. Перевірка виконується на основі комбінації username та serverId, що дозволяє однозначно ідентифікувати сесію в межах розподіленої системи. У випадку відсутності активного входу сервіс повертає відповідь без тіла (204 No Content), що відповідає специфікації Yggdrasil-протоколу та використовується як індикатор відсутності сесії без генерації помилки.

Parameter	Value
username	Name of the profile
serverId	serverId sent by the server to the client
ip *(Optional)*	Client IP obtained by the Minecraft Server, included only when the `prevent-proxy-connections` option is enabled

Рис. 3.5.2. Параметри запиту для HasJoined ендпоінту.

Джерело:

https://yushijinhun.github.io/authlib-injector/en/yggdrasil-server-technical-specification.html

3.5.4 Формування профілю та система текстур

Функція Profile реалізує формування відповіді профілю користувача у форматі, сумісному з клієнтом Minecraft. Основним структурним елементом відповіді є набір властивостей профілю, де ключову роль відіграє property, що містить дані про текстури персонажа.

Формування текстур відбувається шляхом побудови проміжної структури профільних даних, яка включає часову мітку створення, унікальний ідентифікатор користувача, ім’я профілю та посилання на ресурс скіна. Посилання на текстури формується динамічно на основі базового URL сховища аватарів та імені користувача, що дозволяє централізовано керувати ресурсами зовнішнього вигляду.

Сформована структура серіалізується у JSON-формат і додатково кодується у Base64, після чого передається як значення поля textures. Такий підхід забезпечує сумісність із форматом даних, який очікується клієнтом Minecraft, та дозволяє інкапсулювати дані про текстури в єдине поле без необхідності додаткових запитів.

Рис. 3.5.3. Приклад JSON відповіді.

Джерело: Розроблено автором

3.5.5 Підпис профілю та безпека даних

Для забезпечення цілісності та автентичності даних профілю використовується механізм цифрового підпису. Підписування здійснюється на рівні сформованого значення textures та виконується за допомогою асиметричного криптографічного ключа, який зберігається на стороні сервісу.

У випадку, коли клієнтський запит не містить ознаки вимкнення підпису (unsigned=true), система формує криптографічний підпис для значення textures та додає його до відповіді як окрему властивість профілю. Це дозволяє клієнту перевірити, що дані не були змінені після формування сервером.

Якщо у запиті явно вказано режим без підпису, сервіс повертає дані профілю без криптографічного супроводу, залишаючи лише базове представлення інформації.

3.6 Програмна реалізація сервісу генерації аватарів користувачів

Сервіс генерації аватарів користувачів є одним із допоміжних компонентів серверної інфраструктури платформи та забезпечує динамічне формування графічного представлення користувача на основі його ігрового зовнішнього вигляду. Основною функцією даного сервісу є обробка даних скіна користувача та генерація відповідного аватарного зображення у різних форматах для подальшого використання іншими компонентами системи, включаючи веб-інтерфейс, лаунчер та ігрові модулі.

У межах реалізованої архітектури сервіс побудований із використанням мови програмування Go, що забезпечує високу продуктивність, низькі затримки обробки HTTP-запитів та ефективну роботу з великою кількістю одночасних звернень. Взаємодія з сервісом здійснюється через HTTP API, яке надає набір маршрутів для отримання різних типів аватарів користувача.

Основний функціонал сервісу реалізовано через наступні HTTP-ендпоінти:

- **/face/:username** – генерація зображення обличчя;

- **/head/:username** – генерація зображення голови;
- **/skin/:username** – отримання повного зображення скіна у вихідному вигляді.

Кожен із зазначених маршрутів використовує спільний механізм отримання даних скіна користувача та подальшу обробку відповідно до типу запиту. Для генерації зображень застосовуються спеціалізовані рендер-компоненти, які формують фінальний PNG-результат у реальному часі.

Рис. 3.6.1. Приклад генерації обличчя гравця.

Джерело: Розроблено автором

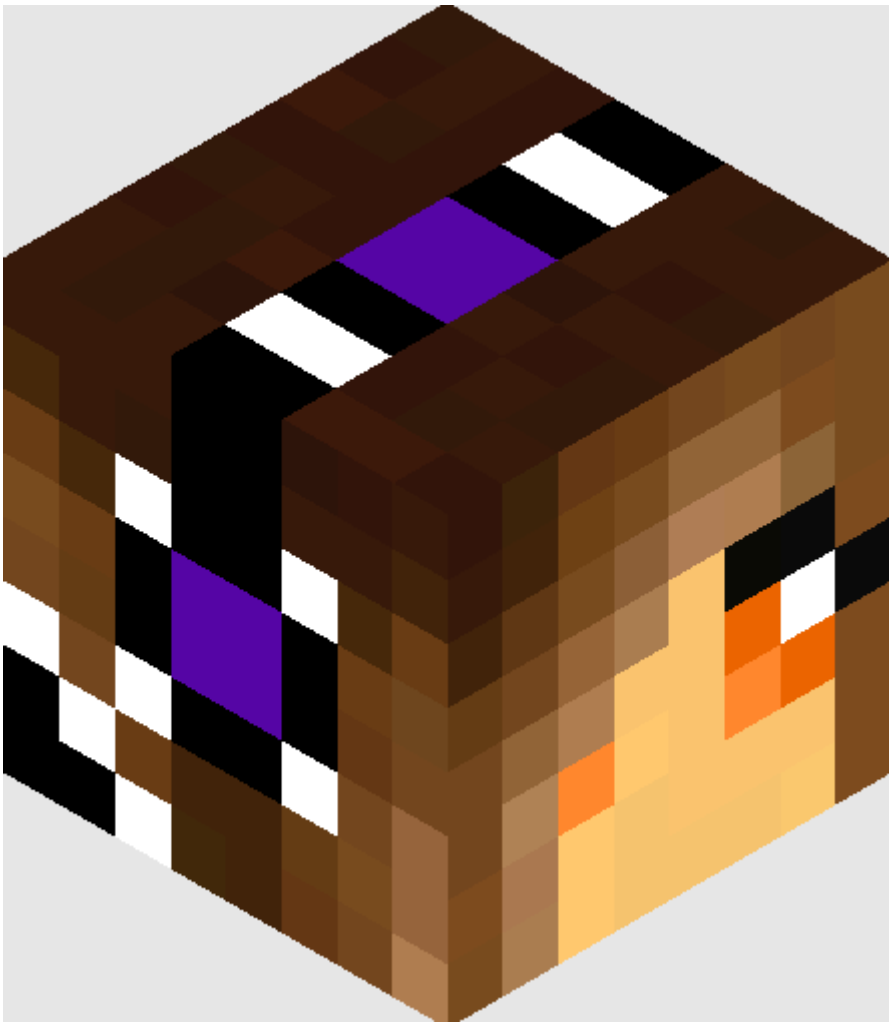

Рис. 3.6.2. Приклад генерації голови гравця з overlay.

Джерело: Розроблено автором

Окремо реалізовано службовий маршрут:

- **/private/:token/cache/skin/:username** – примусове очищення кешу згенерованих зображень для конкретного користувача.

Такий механізм використовується для оновлення аватарів у випадку зміни зовнішнього вигляду персонажа, що дозволяє підтримувати актуальність зображень без необхідності ручного втручання у систему кешування.

Сервіс підтримує параметризовану генерацію аватарів через query-параметри запиту. Зокрема, доступними є параметри size та overlay. Параметр size визначає розмір вихідного зображення та підтримує фіксований набір значень: 8, 16, 32, 64, 128, 256, 512. Значення за замовчуванням конфігуроване на рівні 256. Параметр

overlay дозволяє застосовувати додаткові візуальні накладки до аватара, що використовується для розширення варіантів відображення користувача в інтерфейсі системи.

Важливу роль у роботі сервісу відіграє механізм кешування, який реалізовано на рівні CDN Cloudflare. Це дозволяє значно зменшити навантаження на сервіс генерації зображень та прискорити отримання аватарів для кінцевих користувачів. При зміні зовнішнього вигляду користувача або оновленні скіна виконується примусова інвалідація кешу через окремий службовий запит, що забезпечує актуальність даних.

У процесі виконання запиту сервіс отримує ім’я користувача, після чого виконує отримання відповідного скіна, формує необхідний тип аватара та повертає згенероване зображення у форматі PNG. Логіка обробки побудована таким чином, щоб мінімізувати затримки генерації та забезпечити стабільну роботу навіть при високій кількості паралельних запитів.

Загальна схема роботи сервісу базується на принципі динамічного формування графічних даних без їх попереднього збереження, що дозволяє зменшити обсяг службових операцій та спростити підтримку системи. Такий підхід забезпечує актуальність аватарів користувачів та дозволяє інтегрувати сервіс у різні частини платформи без додаткових залежностей.

Отже, сервіс генерації аватарів користувачів забезпечує ефективне формування графічного представлення персонажів у реальному часі, підтримує механізми кешування та інвалідації даних, а також виступає важливим елементом системи персоналізації користувачів у межах ігрової платформи.

3.7 Програмна реалізація сервісу електронних повідомлень

3.7.1 Загальна реалізація сервісу

Сервіс електронних повідомлень реалізовано як асинхронний компонент системи на основі RabbitMQ, який використовується як брокер повідомлень для

передачі подій між сервісами платформи. Така архітектура дозволяє відокремити процес формування та відправки електронної пошти від основної бізнес-логіки та забезпечує неблокуючу обробку системних подій.

Після виникнення події (реєстрація користувача, відновлення пароля, тощо) відповідний сервіс формує повідомлення та публікує його в RabbitMQ exchange, після чого воно маршрутизується до черги email-сервісу. Для підвищення надійності доставки використовується механізм DLX, який дозволяє перенаправляти повідомлення у випадку помилок обробки або неможливості їх доставки. Це забезпечує ізоляцію проблемних задач та можливість їх повторної обробки.

3.7.2 Реалізація Worker Pool

Обробка повідомлень реалізована за допомогою механізму worker pool, побудованого на конкурентній моделі Go з використанням goroutine та channel.

Worker pool складається з набору паралельних workers, кількість яких задається конфігурацією системи. Якщо значення не вказано, використовується кількість доступних процесорних ядер (runtime.NumCPU()). Кожен worker запускається як окрема goroutine та безперервно очікує задачі з черги jobs chan Job.

Задача в системі представлена як функція типу Job func(), яка інкапсулює повний цикл обробки одного електронного повідомлення. Обробка виконується у циклі з використанням select, який одночасно слухає чергу задач та канал завершення роботи quit.

У разі завершення роботи сервісу worker переходить у режим drain, у якому дочитує всі залишкові задачі з черги перед зупинкою. Це дозволяє уникнути втрати повідомлень при вимкненні системи.

Виконання кожної задачі відбувається через метод run(), який містить захист від критичних помилок. У випадку panic використовується recover(), а інформація про помилку та стек викликів записується через zap logger. Це забезпечує стабільність роботи worker pool навіть при помилках окремих задач.

Така реалізація забезпечує паралельну обробку повідомлень, контроль навантаження через обмежену чергу, ізоляцію помилок та можливість масштабування шляхом збільшення кількості workers.

3.7.3 Формування шаблонів та інтеграція з Resend

Перед відправкою кожне електронне повідомлення проходить етап формування на основі шаблонів. У системі використовується механізм шаблонізації, який дозволяє створювати уніфіковані листи з динамічно підставленими даними користувача.

Шаблони містять змінні поля, які заповнюються під час обробки задачі. До таких даних належать ім’я користувача, коди підтвердження, посилання для відновлення доступу та інші параметри події.

Окремо реалізовано підтримку багатомовності шаблонів. Система підтримує щонайменше дві мови: англійську та українську. Вибір мови здійснюється на основі параметрів користувача або події, після чого підвантажується відповідний шаблон для формування фінального повідомлення.

Для відправки електронних листів використовується зовнішній сервіс Resend, який виконує роль SMTP-провайдера. Інтеграція здійснюється через API-запити. Після формування повідомлення worker передає його до Resend API, який виконує доставку електронної пошти кінцевому користувачу та повертає статус виконання операції. У разі помилки відправки повідомлення може бути повторно поставлене в чергу або перенаправлене до DLX.

3.8 Програмна реалізація клієнтської частини

Клієнтська частина системи реалізована на основі модифікованого клієнта Minecraft версії 1.21.4, побудованого на базі MCP-Reborn. MCP-Reborn використовується як основа для доступу до декомпільованого та ремапнутого коду гри, що дозволяє змінювати внутрішню логіку клієнта, зокрема механізми авторизації, завантаження профілю, обробки подій та відображення користувацьких

даних.

У межах реалізації змінено стандартний механізм авторизації Minecraft через використання authlib-injector. Даний компонент підміняє стандартний Mojang Session Server на власну реалізацію, завдяки чому запити клієнта до **sessionserver/session/minecraft/profile** перенаправляються на власний сервер. У відповідь клієнт отримує модифікований профіль користувача, який містить базові ідентифікаційні дані (UUID, нікнейм), а також додаткову інформацію про текстури персонажа. Отримані дані використовуються клієнтом під час ініціалізації профілю та завантаження зовнішнього вигляду гравця.

Для роботи з текстурами персонажа реалізовано компонент SkinProxy. Його основне призначення полягає у перехопленні процесу отримання скіна та підміні стандартного джерела даних на централізоване сховище системи. SkinProxy виконує запит до серверної частини, отримує URL або дані текстур, після чого передає їх у стандартний механізм завантаження скіна в клієнті Minecraft. Таким чином забезпечується використання єдиного джерела текстур для всіх користувачів системи.

Окрім механізмів авторизації та роботи з текстурами, у клієнт додано окремі ігрові модулі. Зокрема реалізовано AutoSprint – функціональність, яка автоматично вмикає режим спринту під час руху гравця. Модуль інтегрований у меню налаштувань клієнта (Options) і реалізований як окремий перемикач. Логіка роботи полягає у відстеженні стану руху персонажа та автоматичному встановленні стану sprint без необхідності натискання відповідної клавіші.

Загальна реалізація клієнтської частини побудована як набір модифікацій стандартного клієнта Minecraft, інтегрованих через MCP-Reborn. Зміни стосуються трьох основних напрямків: підміна системи авторизації через authlib-injector, реалізація централізованого отримання та застосування текстур через SkinProxy, а також додавання окремих клієнтських модулів, таких як AutoSprint. Усі зміни виконані на рівні клієнта без необхідності модифікації серверної частини гри.

3.9 Програмна реалізація лаунчера

Програмна реалізація лаунчера виконана у вигляді десктопного застосунку, який поєднує веб-інтерфейс користувача з нативною системною логікою запуску клієнта Minecraft. Основою архітектури застосунку використано фреймворк Wails, який забезпечує інтеграцію frontend-частини на базі вебтехнологій із backend-логікою мовою Go.

Backend-частина лаунчера відповідає за взаємодію із файловою системою, роботу з локальними конфігураціями, запуск Java-процесу Minecraft, авторизацію користувача та інтеграцію з серверною інфраструктурою платформи. Для реалізації frontend-частини використовується React, а збірка інтерфейсу виконується через Vite.

Користувацький інтерфейс побудовано з використанням shadcn/ui, а стилізація реалізована за допомогою Tailwind CSS.

Frontend-компонент відповідає за:

- відображення інтерфейсу користувача;
- взаємодію з формами авторизації;
- роботу з налаштуваннями запуску;
- управління профілями користувачів;
- відображення стану запуску клієнта;
- локалізацію інтерфейсу.

Backend-рівень реалізує:

- запуск Minecraft-клієнта;
- формування JVM-параметрів;
- керування Java Runtime;
- взаємодію із файловою системою;
- збереження локальних конфігурацій;
- роботу з токенами авторизації;

- інтеграцію із сервісами платформи.

Перед запуском клієнта лаунчер формує повний набір параметрів середовища виконання, включаючи обсяг доступної оперативної пам’яті, JVM arguments, параметри вікна гри та додаткові launch arguments. Після цього backend-частина ініціює запуск Java-процесу Minecraft із передачею необхідних параметрів автентифікації та конфігурації.

Для підтримки декількох облікових записів реалізовано локальне збереження профілів користувачів, що дозволяє швидко перемикатися між акаунтами без повторного введення облікових даних. Дані активних сесій та токени доступу використовуються для автоматичної авторизації під час повторного запуску застосунку.

Окремо реалізовано інтеграцію із сервісом аватарів платформи. Лаунчер отримує зображення користувачів через HTTP API та відображає їх у користувацькому інтерфейсі під час вибору профілю або роботи з обліковими записами.

Таким чином, реалізований лаунчер виступає повноцінним клієнтським застосунком, який об’єднує механізми авторизації, конфігурації та запуску Minecraft-клієнта, а також забезпечує інтеграцію користувача з усією серверною інфраструктурою платформи.

3.10 Програмна реалізація режиму GunGame

Ігровий режим GunGame реалізовано у вигляді окремого серверного плагіна, який інтегрується в загальну серверну інфраструктуру через внутрішній API-модуль мережевого рівня. Архітектура плагіна побудована за модульним принципом, де окремі компоненти відповідають за керування аренами, станом матчів, прогресією гравців та обробкою ігрових подій.

Основою реалізації є in-memory модель стану, у якій всі активні матчі, арени та дані гравців зберігаються у структурах оперативної пам’яті сервера. Такий підхід

дозволяє мінімізувати затримки під час обробки ігрових подій та уникнути звернення до зовнішніх сховищ у процесі матчу. Дані створюються під час ініціалізації арени та видаляються після завершення гри.

Керування аренами реалізовано через систему внутрішніх станів, яка визначає поточний етап життєвого циклу матчу. Для цього використовуються такі стани:

- **WAITING** – очікування гравців перед початком матчу;
- **STARTING** – підготовка до запуску гри та запуск таймера старту;
- **INGAME** – активна стадія матчу;
- **ENDING** – завершення гри, формування результатів та очищення стану арени.

Підключення нових гравців дозволяється лише для арен у станах WAITING та INGAME. Це забезпечує можливість динамічного приєднання до активних матчів без необхідності очікування створення нової гри.

Параметри арен, координати точок спавну, обмеження кількості гравців та конфігурація системи прогресії зберігаються у зовнішніх конфігураційних файлах. Для серіалізації та десеріалізації конфігурації використовується бібліотека Jackson, що дозволяє автоматично перетворювати конфігураційні дані у внутрішні структури плагіна.

Система прогресії реалізована як послідовний набір рівнів, визначений у конфігурації. Кожен рівень містить прив’язку до конкретного типу зброї. Поточний рівень гравця зберігається у runtime-стані матчу та оновлюється після обробки події вбивства.

Обробка ігрових подій побудована на подієвій моделі серверного API. Подія вбивства виступає основним тригером системи прогресії. Після її виникнення виконується оновлення внутрішнього стану гравця, перевірка умов переходу на наступний рівень та оновлення інвентаря відповідно до конфігурації.

Для форматування повідомлень у чаті використовується бібліотека Adventure

MiniMessage, яка забезпечує підтримку компонентного тексту, кольорів, стилізації та шаблонізації повідомлень. Це дозволяє централізовано формувати повідомлення інтерфейсу та забезпечує гнучке налаштування їхнього вигляду через конфігурацію.

Завершення матчу реалізовано через контроль досягнення максимального рівня прогресії одним із учасників. Після цього арена переходить у стан ENDING, виконується блокування подальшої обробки ігрових подій, формується фінальна статистика матчу та очищаються всі runtime-структури, пов’язані з ареною.

Таким чином, реалізація GunGame побудована як автономний серверний модуль із системою внутрішніх станів, конфігураційною моделлю арен та подієвою обробкою ігрових дій, що забезпечує стабільність роботи та інтеграцію режиму в загальну архітектуру платформи.

ВИСНОВКИ ДО РОЗДІЛУ 3

У межах третього розділу було успішно реалізовано програмне забезпечення та технічну частину розподіленої ігрової платформи Minecraft «Toweray». На основі виконаного аналізу розробленого коду обґрунтовано та впроваджено сучасний технологічний стек, орієнтований на високонавантажені розподілені системи реального часу. Основою для реалізації ігрової логіки та плагінів стала мова Java 21 із використанням віртуальних потоків, що забезпечило ефективну паралельну обробку задач. Інфраструктурні мікросервіси та API були розроблені мовою Go із застосуванням продуктивного фреймворку Gofiber.

Важливим етапом стало проєктування автономного комунікаційного сервісу на базі TCP-протоколу, який виконує роль інтеграційного ядра системи. Взаємодія між Go-сервісами та ігровим ядром на Java була стандартизована за допомогою Protocol Buffers, що забезпечило компактну бінарну серіалізацію, чітко визначені контракти даних і зменшення мережевого навантаження. Стабільність з’єднань між вузлами підтримується механізмом keep-alive через регулярний обмін службовими пакетами.

Для забезпечення безпеки та цілісності даних було розроблено централізовану систему керування обліковими записами та ігровими сесіями, яка виступає єдиним джерелом достовірної інформації для всієї платформи. Автентифікація користувачів і сервісів реалізована на основі OAuth 2.0 із використанням HTTP-ендпоінтів для видачі, оновлення та відкликання токенів доступу. Додатково інтегровано двофакторну автентифікацію відповідно до RFC 6238 для підвищення захисту облікових записів.

Сервіс ігрових сесій реалізовано з урахуванням специфікації Yggdrasil Session Server, що дозволило замінити стандартні механізми автентифікації через authlib-injector. Для захисту користувацьких даних впроваджено механізм криптографічного підпису текстур, а самі файли скінів зберігаються у централізованому об’єктному сховищі Cloudflare R2 із S3-сумісним API. У PostgreSQL при цьому зберігається лише легка метаінформація.

Додатково реалізовано допоміжні клієнт-серверні сервіси, зокрема сервіс генерації аватарів у реальному часі на Go з використанням CDN-кешування Cloudflare, а також асинхронний сервіс електронної пошти на базі RabbitMQ. Завдяки використанню моделі worker pool із goroutine та channel було розділено формування листів і основну бізнес-логіку, що забезпечило неблокуючу доставку повідомлень через Resend API.

Паралельно створено клієнтське середовище, яке включає модифікований Minecraft-клієнт версії 1.21.4 на основі MCP-Reborn та кросплатформений десктопний лаунчер. У клієнті реалізовано перехоплення процесів авторизації та завантаження текстур через authlib-injector і SkinProxy, а також додано ігровий модуль AutoSprint. Лаунчер, побудований із використанням Wails, React 19 і TypeScript, забезпечує зручний інтерфейс, підтримку локалізації, збереження профілів і гнучке налаштування параметрів запуску JVM.

Завершальним етапом стала практична перевірка архітектурних рішень на прикладі ігрового режиму GunGame. Він реалізований як автономний серверний плагін із in-memory моделлю стану, використанням Jackson для обробки конфігурацій і MiniMessage для форматування повідомлень. Реалізація керування життєвими циклами арен без звернення до зовнішніх сховищ підтвердила високу продуктивність застосованої Clean Architecture. У підсумку розроблене програмне забезпечення відповідає вимогам щодо низьких затримок, відмовостійкості та безпеки, створюючи надійну основу для подальшого масштабування системи.

РОЗДІЛ 4
РОЗГОРТАННЯ ТА ПІДТРИМКА СИСТЕМИ

4.1 Інфраструктурне забезпечення системи

У межах сучасного циклу розробки складного програмного забезпечення етап інфраструктурного забезпечення та розгортання розглядається не лише як завершальна стадія технічної реалізації, а як фундаментальний методологічний підхід, що гарантує стабільність, безпеку та можливість розширення системи в реальних умовах експлуатації. Процес переходу від програмного коду до повнофункціональної багатокористувацької платформи вимагає імплементації передових DevOps-практик, які дозволяють автоматизувати життєвий цикл програмного продукту та мінімізувати вплив людського фактора на стан операційного середовища. Враховуючи розподілену природу розробленого ігрового сервера Minecraft, особлива увага приділяється створенню архітектурної парадигми, де кожен функціональний компонент працює в ізольованому, строго контрольованому та легко відтворювальному середовищі.

4.2 Вибір хостинг-провайдера

Важливим чинником забезпечення стабільної роботи високонавантаженої платформи є вибір надійної апаратної бази та провайдера інфраструктурних послуг. Для розгортання розробленої системи було обрано платформу OVHcloud, яка спеціалізується на наданні високопродуктивних рішень для ігрових проєктів. Вибір даного провайдера зумовлений наявністю спеціалізованого захисту від DDoS-атак на рівні магістральної мережі, а також можливістю оренди виділених серверів із високою частотою процесорних ядер. Це є критично важливим для ігрового ядра Minecraft, продуктивність якого безпосередньо залежить від швидкодії одного потоку.

У межах інфраструктурного забезпечення перевага надається використанню сучасних процесорів із високим показником IPC, таких як AMD Ryzen 7 9800X3D, що дозволяє підтримувати стабільний показник TPS навіть за умов пікового

навантаження при великій кількості одночасних підключень користувачів. Використання потужностей OVHcloud дозволяє забезпечити низьку затримку для широкого кола гравців завдяки розвиненій мережевій інфраструктурі провайдера, що є фундаментальною вимогою для систем реального часу.

4.3 Контейнеризація сервісів на базі Docker

Основою для забезпечення ізоляції та переносності компонентів платформи стала технологія Docker. Використання Docker Compose дало змогу представити всю сервісну архітектуру як цілісну систему, що включає центральний комунікаційний вузол на базі Go, ігрові сервери Paper та пов’язані бази даних. Для ефективного управління розгорнутою інфраструктурою було розгорнуто self-hosted екземпляр Portainer. Це рішення слугує централізованим інтерфейсом адміністрування контейнерів, який функціонує безпосередньо на серверах проєкту.

Впровадження Portainer дозволяє реалізувати комплексний підхід до моніторингу та діагностики системи у режимі реального часу. Завдяки візуалізації станів контейнерів, доступу до інтерактивних логів та можливості динамічного коригування параметрів оточення, адміністратор отримує потужний інструмент для підтримки життєздатності платформи. Це забезпечує прозорість усіх внутрішніх процесів та дозволяє оперативно реагувати на критичні події, гарантуючи стабільність багатокористувацького середовища без необхідності прямого втручання в низькорівневі конфігурації операційної системи.

4.4 Автоматизація CI/CD процесів та стратегія версіювання

Центральним механізмом забезпечення життєвого циклу розробки є автоматизована система безперервної інтеграції та доставки, реалізована на базі GitHub Actions. Даний пайплайн виконує роль технологічного конвеєра, що починає роботу з моменту фіксації змін у коді та завершується оновленням робочих образів у реєстрі. Важливою складовою цього процесу є суворе дотримання стандарту Conventional Commits, що дозволяє автоматично структурувати історію розробки та забезпечувати змістовне версіювання. На початкових етапах пайплайну проводиться

ретельна перевірка якості програмного забезпечення, включаючи статичний аналіз коду та валідацію Protobuf-контрактів через buf.build, що критично важливо для запобігання деградації взаємодії між компонентами на Go та Java.

Для надійного збереження та дистрибуції зібраних артефактів було розгорнуто self-hosted рішення ZotRegistry, яке забезпечує повний контроль над приватними Docker-образами та підвищує незалежність інфраструктури. Стратегія версіювання передбачає використання подвійного тегування: динамічного тегу latest для актуальної версії системи та унікального ідентифікатора на основі HEAD SHA commit hash для кожної збірки. Це створює умови для забезпечення принципу незмінності інфраструктури та дозволяє проводити процедури відкату до попередніх стабільних станів у разі виявлення критичних помилок.

Рис. 4.4.1. Пайплайн у GitHub Actions.

Джерело: Розроблено автором

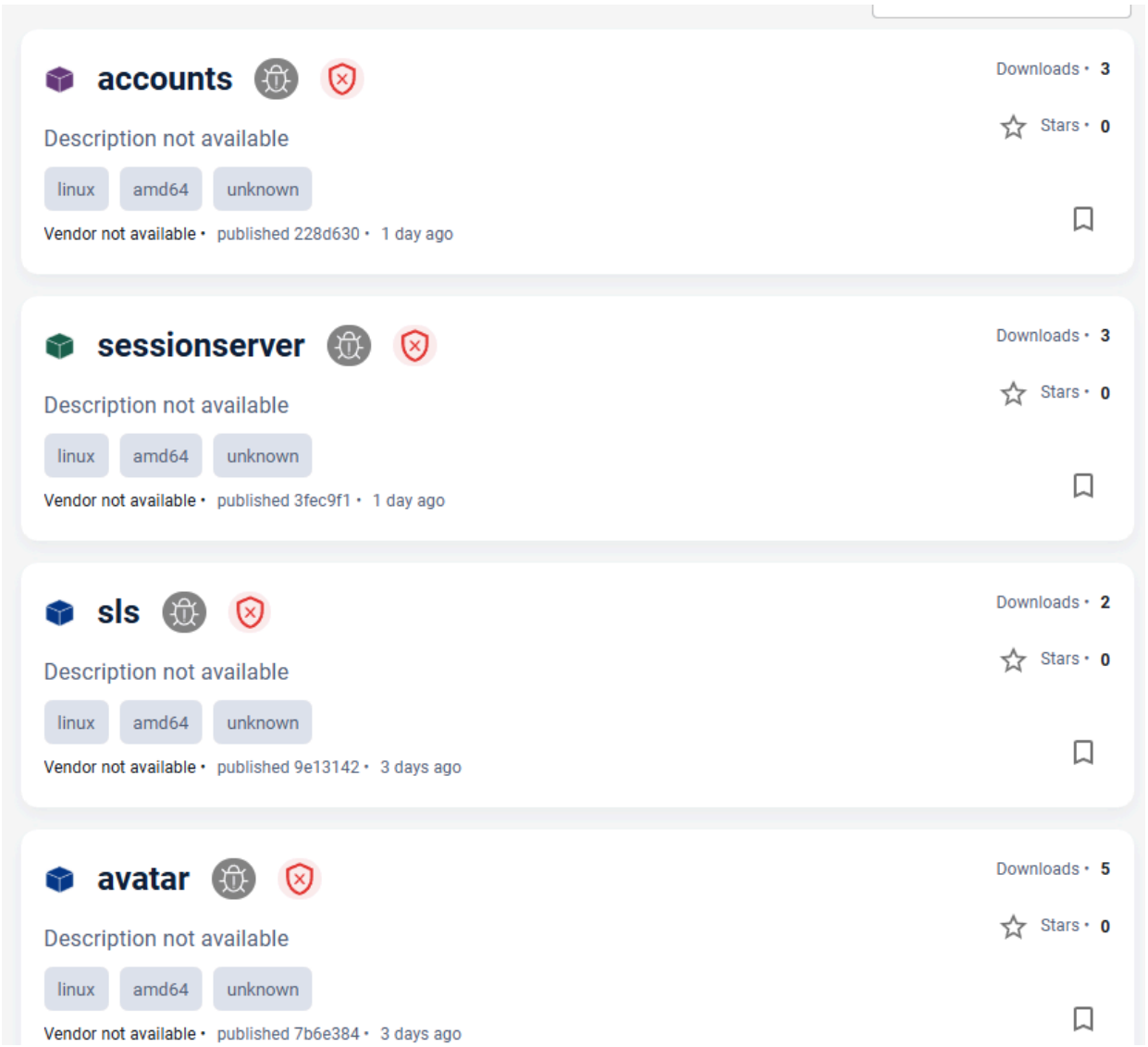

Рис. 4.4.2. Список образів у ZotRegistry.

Джерело: Розроблено автором

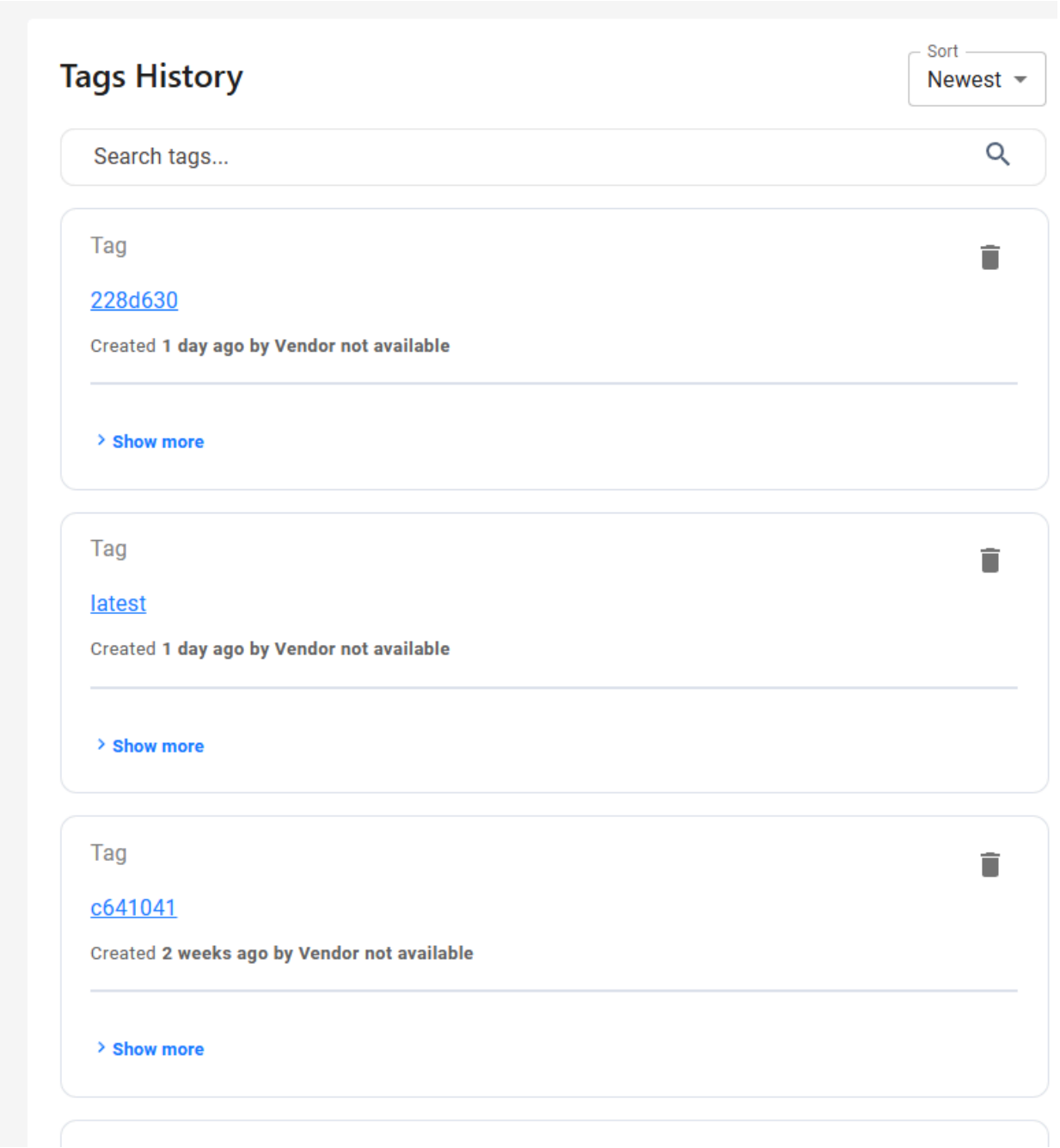

Рис. 4.4.3. Історія тегів образів.

Джерело: Розроблено автором

4.5 Забезпечення мережевої безпеки та маршрутизації трафіку

Окремим пріоритетним напрямком розробки інфраструктури стало забезпечення безкомпромісного рівня мережевої безпеки та захисту конфіденційних даних. Замість традиційних методів публікації сервісів через відкриття портів на мережевому обладнанні, було впроваджено технологію Cloudflare Tunnel. Це

дозволило реалізувати концепцію «невидимого сервера», де порти залишаються повністю закритими для зовнішнього світу, а доступ до платформи здійснюється через захищений шифрований тунель до глобальної мережі Cloudflare.

Маршрутизація трафіку до сервісу ZotRegistry всередині внутрішньої мережі Docker координується реверс-проксі сервером Caddy, який автоматизує процеси отримання TLS-сертифікатів та забезпечує безпечну передачу даних через протокол HTTPS. Фінальна верифікація працездатності розгорнутої платформи підтверджує, що поєднання сучасних методів автоматизації та мережевого екранування дозволяє створити стійке до зовнішніх викликів середовище, гарантуючи безперебійну багатокористувацьку взаємодію в довгостроковій перспективі.

ВИСНОВКИ ДО РОЗДІЛУ 4

У межах четвертого розділу було успішно реалізовано комплексну задачу створення сучасної, високотехнологічної інфраструктури, призначеної для стабільного розгортання та ефективного управління ігровою платформою. На основі проведеної роботи встановлено, що використання методології контейнеризації на базі Docker та засобів оркестрації дозволяє повністю ізолювати функціональні компоненти системи, мінімізуючи ризики виникнення конфліктів у програмному середовищі та забезпечуючи ідентичність умов роботи на всіх етапах експлуатації. Впровадження інструменту Portainer суттєво підвищило рівень операційного контролю, надавши зручні засоби для візуального моніторингу та діагностики стану сервісів.

Важливою основою для розгортання спроєктованої архітектури став вибір хостинг-провайдера OVHcloud, що дало змогу забезпечити систему необхідними апаратними ресурсами високої потужності. Вибір даної платформи обґрунтований наявністю спеціалізованого апаратного захисту від DDoS-атак, що є критично важливим для ігрових проєктів, а також можливістю використання високопродуктивних процесорів із високою частотою на ядро. Це забезпечує необхідну швидкість обробки ігрових подій в однопотоковому режимі, що є характерною особливістю ядра Minecraft, та гарантує стабільний показник TPS навіть за умов значного скупчення гравців на одному серверному вузлі.

Особливе значення для проєкту має реалізована система CI/CD через GitHub Actions, яка забезпечує повну автоматизацію процесів тестування, збирання та доставки програмного забезпечення. Поєднання стандартів Conventional Commits та використання спеціалізованого реєстру ZotRegistry з чіткою стратегією версіювання за SHA-хешами дозволило створити прозорий та надійний механізм управління змінами. Такий підхід гарантує високу швидкість ітерацій розробки при збереженні стабільності платформи та можливості швидкого відновлення системи після збоїв.

Завершальним ключовим аспектом інфраструктурного забезпечення стало

впровадження інноваційних рішень у сфері мережевої безпеки. Застосування технології Cloudflare Tunnel у поєднанні з реверс-проксі сервером Caddy дозволило радикально знизити вразливість системи перед зовнішніми загрозами, забезпечивши повну закритість мережевих портів та автоматичне шифрування трафіку. Спроєктована та впроваджена інфраструктура є логічним завершенням циклу розробки, перетворюючи теоретичну архітектурну модель на стійку, безпечну та готову до навантажень інформаційну систему.

ВИСНОВКИ

За результатами проведеного комплексного дослідження та розробки ігрової платформи Minecraft було досягнуто поставленої мети – створено стабільну, масштабовану та захищену серверну інфраструктуру, яка забезпечує ефективну багатокористувацьку взаємодію. Усі сформульовані у роботі задачі були послідовно виконані, що дозволило реалізувати цілісний програмний продукт, придатний до експлуатації в умовах підвищеного навантаження.

У процесі теоретичного аналізу встановлено, що Minecraft є зручною основою для побудови розподілених систем, а вибір серверного ядра Paper суттєво вплинув на підвищення продуктивності та стабільності ігрового процесу. Обґрунтування гібридної архітектури дало змогу поєднати монолітну обробку ігрової логіки з мікросервісним підходом до інфраструктурних компонентів, що створило основу для подальшого масштабування системи.

На етапі проєктування було визначено структуру платформи із центральним комунікаційним вузлом, який забезпечує узгоджений обмін даними між компонентами. Реалізовано окремі сервіси для управління користувачами, ігровими сесіями, генерацією аватарів та повідомленнями, що дозволило відокремити інфраструктурну логіку від ігрового процесу та підвищити відмовостійкість системи.

Програмна реалізація виконана із застосуванням сучасного стеку технологій, зокрема Go та Java, що відповідає вимогам високонавантажених систем. Використання Protocol Buffers забезпечило ефективний і типобезпечний обмін даними з мінімальними витратами мережевого трафіку. Дотримання принципів чистої архітектури сприяло гнучкості та подальшій масштабованості розробки.

Окрему увагу приділено автоматизації розгортання та мережевій безпеці. Впровадження CI/CD за допомогою GitHub Actions, використання реєстру ZotRegistry та системи керування Portainer забезпечили повністю автоматизований життєвий цикл застосунку. Комбінація Cloudflare Tunnel і реверс-проксі Caddy

реалізувала концепцію «закритих портів», що підвищило рівень захисту від зовнішніх загроз і DDoS-атак.

Практичне тестування на прикладі режиму GunGame підтвердило працездатність архітектурних рішень і ефективність механізмів синхронізації даних. Система продемонструвала стабільну роботу при значному навантаженні, високу швидкість відгуку та цілісність даних користувачів.

Отримані результати свідчать про успішне досягнення мети дослідження. Розроблена платформа має практичну цінність і може бути використана для створення як комерційних, так і некомерційних Minecraft-мереж із підвищеними вимогами до продуктивності, масштабованості та безпеки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. **Gofiber**. An Express-inspired web framework written in Go. [Електронний ресурс] / Gofiber. – Режим доступу: https://gofiber.io/ (дата звернення: 10.05.2026).
2. **Paper**. The blazing fast Minecraft server. [Електронний ресурс] / PaperMC. – Режим доступу: https://papermc.io/software/paper/ (дата звернення: 10.05.2026).
3. **SpigotMC**. High Performance Minecraft Software. [Електронний ресурс] / SpigotMC. – Режим доступу: https://www.spigotmc.org/wiki/spigot/ (дата звернення: 12.05.2026).
4. **Cloudflare**. Build, deploy, and govern AI agents on the same network.. [Електронний ресурс] / Cloudflare. – Режим доступу: https://www.cloudflare.com/ (дата звернення: 12.05.2026).
5. **Cloudflare R2**. Scalable, durable, affordable object storage. [Електронний ресурс] / Cloudflare. – Режим доступу: https://www.cloudflare.com/products/r2/ (дата звернення: 12.05.2026).
6. **OVHcloud**. Cloud Computing & Web Hosting. [Електронний ресурс] / OVHcloud. – Режим доступу: https://www.ovhcloud.com/en/ (дата звернення: 12.05.2026).
7. **Caddy**. The Ultimate Server makes your sites more secure, more reliable, and more scalable than any other solution. [Електронний ресурс] / Caddy. – Режим доступу: https://caddyserver.com/ (дата звернення: 13.05.2026).
8. **Go**. The Go Programming Language. [Електронний ресурс] / Go. – Режим доступу: https://go.dev/ (дата звернення: 13.05.2026).
9. **Docker**. Accelerated Container Application Development. [Електронний ресурс] / Docker. – Режим доступу: https://www.docker.com/ (дата звернення: 13.05.2026).
10. **Protocol Buffers**. Google's data interchange format. [Електронний ресурс] / Protobuf. – Режим доступу: https://protobuf.dev/ (дата звернення: 13.05.2026).
11. **Poratiner**. Operational control for Kubernetes, Docker, and Podman; without the specialist overhead. [Електронний ресурс] / Portainer. – Режим доступу: https://www.portainer.io/ (дата звернення: 13.05.2026).

12. **Zot Registry**. OCI-native container image registry. [Електронний ресурс] / ZotRegistry. – Режим доступу: https://zotregistry.dev/v2.1.16/ (дата звернення: 13.05.2026).
13. **Mojang API**. Mojang API. [Електронний ресурс] / Minecraft. – Режим доступу: https://minecraft.wiki/w/Mojang_API (дата звернення: 14.05.2026).
14. **Yggdrasil**. Yggdrasil Server Technical Specification. [Електронний ресурс] / GitHub. – Режим доступу: https://yushijinhun.github.io/authlib-injector/en/yggdrasil-server-technical-specification.html (дата звернення: 14.05.2026).