Uniwersytet Warmińsko-Mazurski w Olsztynie
Wydział Matematyki i Informatyki

Kacper Renkel

Kierunek: Informatyka

Projekt i implementacja gry internetowej inspirowanej serią Plemiona

Praca inżynierska wykonana
w Instytucie Matematyki
pod kierunkiem
dr. Piotra Jastrzębskiego

Olsztyn, 2026 rok

University of Warmia and Mazury in Olsztyn
Faculty of Mathematics and Computer Science

Kacper Renkel

Field of Study: Computer Science

Design and implementation of an online game inspired by the Tribes series

Engineer's Thesis is performed
in the Institute of Mathematics
under supervision of
Piotr Jastrzębski, PhD

Olsztyn, 2026

Streszczenie

Tematem niniejszej pracy inżynierskiej jest projekt oraz implementacja wieloosobowej gry strategicznej czasu rzeczywistego (MMORTS), osadzonej w środowisku przeglądarki internetowej. Inspiracją dla powstania aplikacji były klasyczne tytuły tego gatunku, takie jak „Plemiona", które od lat cieszą się niesłabnącą popularnością, stanowiąc jednocześnie interesujące wyzwanie programistyczne. Gry tego typu charakteryzują się koniecznością obsługi wielu użytkowników jednocześnie, ciągłością rozgrywki niezależną od obecności gracza online oraz skomplikowaną logiką biznesową, opartą na zależnościach czasowych. Głównym celem pracy było stworzenie w pełni funkcjonalnego systemu, który odwzorowuje kluczowe mechaniki gatunku, a jednocześnie realizuje założenia nowoczesnej inżynierii oprogramowania w zakresie wydajności, skalowalności oraz bezpieczeństwa danych.

Aplikacja została zaprojektowana w architekturze wielowarstwowej, co pozwoliło na wyraźne odseparowanie logiki biznesowej, warstwy dostępu do danych oraz interfejsu użytkownika. Do realizacji warstwy serwerowej (backendu) wykorzystano język programowania Typescript wspierany przez framework Nest. Wybór ten podyktowany był potrzebą zapewnienia wysokiej wydajności przy obsłudze zapytań sieciowych oraz dostępnością rozbudowanych bibliotek wspomagających zarządzanie bezpieczeństwem i komunikacją z bazą danych. Komunikacja między klientem a serwerem odbywa się w oparciu o architekturę REST API, gdzie dane przesyłane są w lekkim formacie JSON oraz przy pomocy WebSocket, by dane były dostępne bez konieczności odświeżania strony. Takie podejście umożliwiło stworzenie uniwersalnego interfejsu programistycznego, który w przyszłości może posłużyć do stworzenia dedykowanej aplikacji mobilnej bez konieczności ingerencji w kod serwerowy.

Fundamentem działania gry jest zaprojektowana i wdrożona relacyjna baza danych, oparta na systemie MySQL. Schemat bazy danych został starannie znormalizowany, aby zapewnić integralność danych oraz zminimalizować redundancję. Kluczowe tabele przechowują informacje o kontach użytkowników, parametrach poszczególnych wiosek, stanie posiadania jednostek wojskowych, poziomie rozbudowy budynków.

Warstwa prezentacji (frontend) została zrealizowana przy użyciu frameworka Angular, co pozwoliło na stworzenie dynamicznego i responsywnego interfejsu użytkownika. Dzięki zastosowaniu technologii komponentowej, aplikacja odświeża jedynie te elementy widoku, które uległy zmianie, bez konieczności przeładowywania całej strony. Jest to kluczowe dla komfortu rozgrywki, zwłaszcza przy przeglądaniu interaktywnej mapy świata.

Kwestie bezpieczeństwa zostały potraktowane priorytetowo. Aplikacja implementuje system uwierzytelniania i autoryzacji oparty na tokenach JWT, co zabezpiecza sesję użytkownika przed przejęciem. Wszelkie dane wejściowe przesyłane przez graczy podlegają ścisłej walidacji po stronie serwera, co uniemożliwia manipulację stanem gry poprzez modyfikację kodu klienta lub wysyłanie sfałszowanych żądań API. Zastosowano również haszowanie haseł przy użyciu nowoczesnych algorytmów kryptograficznych, zapewniając poufność danych użytkowników.

Abstract

Design and implementation of an online game inspired by the Tribes series

The subject of this engineering thesis is the design and implementation of a multiplayer real-time strategy game (MMORTS), based in a web browser environment. The application was inspired by classic titles of the genre, such as "Tribal Wars," which have enjoyed enduring popularity for years while presenting an interesting programming challenge. Games of this type are characterized by the need to handle multiple concurrent users, gameplay continuity independent of the player's online status, and complex business logic based on temporal dependencies. The primary objective of the work was to create a fully functional system that replicates the genre's key mechanics while adhering to modern software engineering principles regarding performance, scalability, and data security.

The application was designed using a multi-tier architecture, allowing for a clear separation of business logic, the data access layer, and the user interface. The server-side layer (backend) was implemented using the TypeScript programming language, supported by the Nest framework. This choice was driven by the need to ensure high performance in handling network requests and the availability of extensive libraries supporting security management and database communication. Client-server communication relies on REST API architecture, utilizing the lightweight JSON format for data transfer, as well as WebSockets to ensure data availability without the need for page reloads. This approach enabled the creation of a universal application programming interface (API), which could facilitate the future development of a dedicated mobile application without requiring modifications to the server-side code.

The foundation of the game's operation is a designed and implemented relational database based on the MySQL system. The database schema was carefully normalized to ensure data integrity and minimize redundancy. Key tables store information regarding user accounts, village parameters, military unit inventories, and building upgrade levels. The presentation layer (frontend) was developed using the Angular framework, enabling the creation of a dynamic and responsive user interface. Thanks to the use of component-based technology, the application updates only those view elements that have changed, eliminating the need for full page reloads. This is crucial for the gameplay experience, particularly when navigating the interactive world map.

Security considerations were given high priority. The application implements an authentication and authorization system based on JWT tokens, protecting user sessions against hijacking. All input data submitted by players undergoes strict server-side validation, preventing game state manipulation via client code modification or forged API requests. Password hashing using modern cryptographic algorithms was also employed to ensure user data confidentiality.

Podziękowanie

Składam serdeczne podziękowania Panu dr. Piotrowi Jastrzębskiemu za objęcie opieką naukową niniejszej pracy. Dziękuję za poświęcony czas, cenne wskazówki merytoryczne oraz pomoc w ukierunkowaniu moich rozważań, które przyczyniły się do powstania tej pracy.

Spis treści

Wstęp

Głównym celem niniejszej pracy jest zaprojektowanie oraz implementacja gry internetowej typu MMORTS, inspirowanej mechaniką popularnej serii "Plemiona". Projekt ma na celu stworzenie funkcjonalnej aplikacji webowej, umożliwiającej rozgrywkę w czasie rzeczywistym w środowisku przeglądarki internetowej. Zakres pracy obejmuje zarówno część teoretyczną jak i praktyczną. Część teoretyczna koncentruje się na charakterystyce gier przeglądarkowych, analizie dostępnych rozwiązań. Natomiast część praktyczna, stanowiąca trzon pracy, obejmuje realizację kluczowych mechanik gry, takich jak: system zarządzania surowcami i rozbudowy wioski, system interakcji między graczami czy też system autoryzacji i zarządzania kontem użytkownika.

Wybór tematu niniejszej pracy inżynierskiej podyktowany był chęcią pogłębienia kompetencji w zakresie zaprojektowania złożonych systemów webowych. Gry typu MMORTS (ang. *Massively Multiplayer Online Real-Time Strategy*), a w szczególności seria "Plemiona", stanowią interesujący obiekt analizy z punktu widzenia inżynierii oprogramowania. W przeciwieństwie do prostych aplikacji typu CRUD, systemy tego rodzaju wymagają zaawansowanej logiki biznesowej, która musi funkcjonować w sposób ciągły. Głównym motywem podjęcia tej tematyki, była chęć zmierzenia się z konkretnymi wyzwaniami technologicznymi, takimi jak Wykorzystanie protokołu WebSocket do zapewnienia natychmiastowych powiadomień o zdarzeniach w grze czy również Zastosowanie nowoczesnego stosu technologicznego (NestJS, Angular, MySQL) pozwala na sprawdzenie w praktyce wzorców projektowych i metodologii tworzenia oprogramowania, które są obecnie standardem w branży IT. Dodatkowym uzasadnieniem jest fakt, że gry przeglądarkowe, mimo upływu lat, wciąż mają oddaną grupę odbiorców. Dodatkowo użycie nowoczesnych narzędzi takich jak TypeScript pozwala na pokazanie jak współczesne frameworki mogą usprawnić proces tworzenia systemów.

Rozdział 1

Analiza tematyki i przegląd technologii

1.1. Charakterystyka gatunku gier MMORTS (Massively Multiplayer Online Real-Time Strategy)

Gry z gatunku MMORTS (ang. *Massively Multiplayer Online Real-Time Strategy*) stanowią specyficzną hybrydę, łączącą w sobie cechy klasycznych strategii czasu rzeczywistego (RTS) z modelami rozgrywki masowej (MMO). W odróżnieniu od tradycyjnych gier RTS, takich jak *StarCraft* czy *Age of Empires*, gdzie rozgrywka toczy się w zamkniętych sesjach (meczach) o ograniczonym czasie trwania, gry MMORTS osadzone są w tzw. świecie trwałym (ang. *persistent world*). Oznacza to, że stan gry ewoluuje nieprzerwanie, 24 godziny na dobę, niezależnie od tego, czy dany użytkownik jest w tym momencie zalogowany do systemu, czy też nie[1]. Fundamentem tego gatunku jest czas jako kluczowy zasób strategiczny. W klasycznych grach RTS budowa jednostki lub struktury trwa zazwyczaj od kilku sekund do kilku minut. W grach przeglądarkowych typu MMORTS, takich jak *Plemiona* (ang. *Tribal Wars*) czy *Travian*, procesy te są znacznie wydłużone i mogą trwać od kilkunastu minut do wielu godzin, a nawet dni w zaawansowanych fazach rozgrywki. Taka konstrukcja mechaniki wymusza na graczu planowanie długoterminowe oraz regularne, choć niekoniecznie ciągłe, doglądanie stanu swojego konta[2].

Kluczowe cechy wyróżniające gatunek MMORTS to:

— **Ciągłość świata gry:** Serwer gry nieprzerwanie przetwarza logikę symulacji. Surowce są wydobywane, a wojska przemieszczają się po mapie nawet wtedy, gdy gracz wyłączy przeglądarkę. Stwarza to unikalne zagrożenie - gracz może zostać zaatakowany i stracić dorobek podczas swojej nieobecności, co jest silnym motywatorem do częstych powrotów do gry oraz łączenia się w grupy (sojusze/plemiona) w celu wzajemnej ochrony[3].

— **Skalowalność i masowość:** Architektura systemu musi obsługiwać tysiące graczy jednocześnie, którzy wchodzą ze sobą w interakcje na jednej, wspólnej mapie świata. Wymaga to od strony technicznej zastosowania wydajnych rozwiązań bazodanowych i optymalizacji zapytań, aby uniknąć opóźnień (lagów), które mogłyby wpłynąć na wynik starć militarnych, gdzie często decydują ułamki sekund.

— **Ekonomia i zarządzanie zasobami:** Podstawą rozwoju jest zazwyczaj model ekonomiczny oparty na wydobyciu kilku podstawowych surowców (np. drewno, glina, żelazo). Tempo rozwoju gracza jest ściśle skorelowane z wydajnością jego ekonomii, co nadaje grze charakter matematycznej optymalizacji.

— **Brak warunku zwycięstwa w krótkim terminie:** Gry te rzadko mają definitywny koniec w krótkim czasie. Rozgrywka na jednym serwerze (świecie) może trwać miesiącami lub latami, aż do momentu dominacji jednej grupy graczy, co zazwyczaj kończy się tzw. „zamknięciem świata" i startem nowej edycji.

1.2. Analiza gry "Plemiona" - dekonstrukcja mechanik i trendów (nawiązanie do pierwowzoru)

Gra przeglądarkowa „Plemiona" (znana międzynarodowo jako *Tribal Wars*), wyprodukowana przez niemieckie studio InnoGames, zadebiutowała w 2003 roku i od tego czasu stanowi kanoniczny przykład gatunku MMORTS. Mimo upływu ponad dwóch dekad i dynamicznego rozwoju grafiki w branży gier wideo, tytuł ten utrzymuje swoją popularność dzięki unikalnemu balansowi mechanik oraz silnemu naciskowi na aspekt socjologiczny rozgrywki[5].

Historia gry „Plemiona" sięga początków 2003 roku, kiedy to bracia Hendrik i Eike Klindworth, wspólnie z Michaelem Zillmerem, rozpoczęli prace nad projektem hobbystycznym znanym pierwotnie jako „Die Stämme". Dzięki innowacyjnemu na tamte czasy podejściu do czasu rzeczywistego w przeglądarce oraz niskim wymaganiom sprzętowym, tytuł błyskawicznie zyskał popularność, osiągając w ciągu roku liczbę tysięcy aktywnych użytkowników, co w 2007 roku doprowadziło do sformalizowania działalności i założenia firmy InnoGames[5].

Poniższa dekonstrukcja mechanik gry „Plemiona" posłużyła jako baza wymagań funkcjonalnych dla projektowanej w ramach niniejszej pracy aplikacji.

1.2.1. Ekonomia i model przyrostowy

Fundamentem gry jest prosty, lecz rygorystyczny model ekonomiczny oparty na trzech surowcach: drewno, glina i żelazo. Mechanika ta opiera się na kilku kluczowych założeniach:

— **Generowanie pasywne:** Surowce przyrastają w sposób ciągły (określona liczba jednostek na godzinę), zależny od poziomu rozbudowy kopalń. Wymaga to implementacji po stronie serwera mechanizmu obliczającego stan surowców w oparciu o różnicę czasu (Δt) między ostatnią aktualizacją a bieżącym żądaniem (tzw. *lazy evaluation*) lub cyklicznych zadań (CRON).

— **Wzrost wykładniczy kosztów:** Koszt rozbudowy budynków oraz czas trwania konstrukcji rosną zazwyczaj wykładniczo wraz z każdym kolejnym poziomem. Wymusza to na graczu inwestowanie w infrastrukturę ekonomiczną (magazyny, kopalnie) przed inwestycją w potencjał militarny.

— **Ograniczona pojemność:** Wprowadzenie limitu magazynowego (tzw. *cap*) jest mechanizmem zmuszającym gracza do regularnej aktywności. Przepełnienie magazynu oznacza bezpowrotną utratę wyprodukowanych zasobów, co jest karą za brak logowania.

1.2.2. Czasoprzestrzeń i nawigacja

W grze „Plemiona" świat przedstawiony jest jako dwuwymiarowa siatka współrzędnych kartezjańskich (x, y). Każda osada posiada unikalną pozycję, a odległość między nimi obliczana jest zazwyczaj w oparciu o metrykę euklidesową. Kluczową mechaniką jest tutaj ścisłe powiązanie odległości z czasem podróży. Czas dotarcia jednostki do celu jest funkcją dystansu oraz prędkości najwolniejszej jednostki w oddziale:

$$t_{podrozy} = \frac{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}{v_{jednostki}}$$

Ten deterministyczny charakter ruchu wojsk pozwala graczom na precyzyjne planowanie ataków (tzw. synchronizacja), gdzie o sukcesie decydują często milisekundy. Dla programisty oznacza to konieczność precyzyjnej obsługi znaczników czasu (timestamp) i uwzględnienia opóźnień sieciowych.

1.2.3. System walki i przejmowania kontroli

Walka w analizowanym tytule nie odbywa się w sposób interaktywny (jak w grach FPS czy klasycznych RTS). Jest to proces obliczeniowy wykonywany natychmiastowo w momencie dotarcia wojsk do celu. System generuje raport walki, porównując parametry ataku i obrony, uwzględniając modyfikatory takie jak:

— **Bonusy terenowe i budynki:** Np. wpływ poziomu muru obronnego na siłę defensywy.
— **Czynniki losowe:** Wprowadzenie zmiennej „Szczęście" (w zakresie np. -25% do +25%) zapobiega pełnej przewidywalności wyników i dodaje element ryzyka.
— **Morale:** Mechanizm balansujący rozgrywkę, chroniący słabszych graczy przed agresją znacznie silniejszych przeciwników poprzez sztuczne obniżenie siły ataku.

Unikalną cechą mechaniki „Plemion", która została zaimplementowana również w niniejszym projekcie, jest system przejmowania wiosek. Zamiast niszczenia bazy przeciwnika, gracz dąży do obniżenia jej parametru „poparcia" (lojalności) do zera za pomocą specjalnych jednostek (Szlachcic). Pozwala to na organiczny rozrost imperium gracza i przejmowanie dorobku innych użytkowników.

1.2.4. Wnioski dla projektu inżynierskiego

Analiza pierwowzoru wykazała, że siła gier tego typu nie leży w zaawansowanej warstwie wizualnej, lecz w spójności reguł matematycznych i niezawodności silnika gry. Współczesne trendy w grach przeglądarkowych (ewolucja od statycznego HTML do dynamicznych SPA) wskazują na konieczność zapewnienia płynnego interfejsu, który maskuje opóźnienia w komunikacji z serwerem. Projekt realizowany w ramach tej pracy czerpie z „Plemion" logikę biznesową, przenosząc ją jednak na nowoczesny stos technologiczny (Angular, NestJS), co pozwala na wyeliminowanie archaicznych rozwiązań, takich jak konieczność ręcznego odświeżania strony w celu zobaczenia nowych powiadomień.

1.3. Przegląd wybranych narzędzi i technologii

Wybór stosu technologicznego dla gry typu MMORTS musiał uwzględniać specyfikę gatunku, w którym kluczowe są: wysoka interaktywność, przetwarzanie zdarzeń w czasie rzeczywistym oraz spójność danych. Zdecydowano się na wykorzystanie ekosystemu opartego na języku TypeScript, co pozwoliło na zachowanie wysokiej typizacji zarówno po stronie serwerowej, jak i klienckiej.

1.3.1. Język programowania - TypeScript

Podstawowym językiem wykorzystanym w projekcie jest TypeScript. Jest to nadzbiór języka JavaScript, który wprowadza statyczne typowanie, klasy oraz interfejsy. Zastosowanie TypeScriptu w projekcie gry o złożonej logice biznesowej (zależności między budynkami, jednostkami i surowcami) znacząco redukuje ryzyko wystąpienia błędów na etapie wykonania (runtime) oraz ułatwia refaktoryzację kodu.

1.3.2. Framework serwerowy - NestJS

Do budowy warstwy backendowej wykorzystano framework NestJS (wersja 10). Jest to progresywny framework Node.js, który w swojej architekturze silnie inspiruje się Angular'em,

promując modularność i czystość kodu poprzez mechanizm wstrzykiwania zależności (Dependency Injection).
— **TypeORM:** Wykorzystany jako system mapowania obiektowo-relacyjnego (ORM), co umożliwiło intuicyjne zarządzanie bazą danych MySQL za pomocą klas TypeScript.
— **NestJS Schedule:** Biblioteka ta posłużyła do implementacji zadań cyklicznych (cron jobs), które są niezbędne w grze strategicznej do aktualizacji stanu surowców w czasie rzeczywistym oraz sprawdzania statusu zakończenia budowy czy rekrutacji jednostek.
— **Passport i JWT:** Do obsługi bezpiecznego uwierzytelniania wykorzystano bibliotekę `@nestjs/jwt`, co pozwoliło na bezstanową weryfikację użytkowników.

1.3.3. Warstwa kliencka - Angular

Front-end aplikacji został zrealizowany w frameworku Angular (wersja 17). Wybór ten był podyktowany potrzebą stworzenia rozbudowanego interfejsu Single Page Application (SPA), który zapewni płynność rozgrywki zbliżoną do aplikacji natywnych.
— **Angular Material:** Zastosowano bibliotekę komponentów graficznych `@angular/material`, aby zapewnić spójny i responsywny wygląd elementów interfejsu, takich jak okna modalne, tabele statystyk czy formularze.
— **RxJS:** Wykorzystanie programowania reaktywnego pozwoliło na efektywne zarządzanie strumieniami danych, szczególnie przy obsłudze asynchronicznych odpowiedzi z API oraz powiadomień z serwera.
— **ngx-translate:** Umożliwiło przygotowanie aplikacji do wielojęzyczności, co jest standardem w nowoczesnych produkcjach internetowych.

1.3.4. Komunikacja w czasie rzeczywistym - Socket.io

W grach typu MMORTS kluczowe jest, aby gracz otrzymywał informacje o zdarzeniach (np. powrocie wojsk czy zakończeniu budowy) natychmiastowo, bez konieczności odświeżania strony. Do tego celu wykorzystano protokół WebSocket oraz bibliotekę `Socket.io`. Dzięki niej serwer może inicjować komunikację z klientem (push notifications), co znacznie poprawia dynamikę rozgrywki.

1.3.5. Baza danych i bezpieczeństwo

Jako system zarządzania bazą danych wybrano MySQL, co zapewnia trwałość i integralność danych relacyjnych. Bezpieczeństwo danych wrażliwych zapewniono poprzez:
— **Bcrypt:** Wykorzystany do jednostronnego haszowania haseł użytkowników przed zapisaniem ich w bazie.
— **Class-validator:** Biblioteka stosowana w backendzie do ścisłej walidacji danych przychodzących od klienta (DTO - Data Transfer Objects), co zapobiega atakom typu SQL Injection oraz próbom przesyłania niepoprawnych parametrów gry.

1.3.6. Narzędzia wspomagające

W procesie wytwórczym wykorzystano również szereg narzędzi pomocniczych:
— **Swagger (@nestjs/swagger):** Do automatycznego generowania dokumentacji technicznej API, co ułatwiło testowanie punktów końcowych (endpoints).
— **Date-fns:** Zaawansowana biblioteka do manipulacji czasem, kluczowa przy obliczaniu odliczania do zakończenia akcji w grze.

— **Prettier i ESLint:** Narzędzia do automatycznego formatowania i analizy statycznej kodu, zapewniające wysoką jakość i spójność zapisu w całym zespole projektowym (lub w trakcie samodzielnych prac).

1.4. Środowisko programistyczne i narzędzia wspomagające

Proces wytwórczy oprogramowania wymagał dobrania odpowiednich narzędzi, które zapewniły efektywność pracy, kontrolę nad kodem źródłowym. Poniżej przedstawiono kluczowe elementy środowiska programistycznego wykorzystane podczas realizacji projektu.

1.4.1. Środowisko uruchomieniowe - Node.js

Podstawowym środowiskiem uruchomieniowym dla warstwy serwerowej oraz narzędzi deweloperskich warstwy klienckiej był Node.js. Dzięki silnikowi V8, Node.js umożliwia wykonywanie kodu JavaScript/TypeScript po stronie serwera, co pozwoliło na zachowanie wysokiej wydajności przy operacjach wejścia-wyjścia, niezbędnych w komunikacji sieciowej gry.

1.4.2. Zintegrowane środowisko programistyczne (IDE) - Visual Studio Code

Głównym narzędziem wykorzystanym do pisania kodu był Visual Studio Code. Wybór ten podyktowany był lekkością oraz doskonałym wsparciem dla języka TypeScript i frameworków Angular oraz NestJS. W celu podniesienia efektywności pracy wykorzystano szereg rozszerzeń, takich jak:
— **Angular Language Service:** Wspomagający nawigację po kodzie i autouzupełnianie w szablonach HTML.
— **ESLint i Prettier:** Zapewniające automatyczną kontrolę jakości kodu oraz spójność formatowania zgodnie z przyjętymi standardami.
— **GitLens:** Rozszerzenie ułatwiające analizę historii zmian w systemie kontroli wersji.

1.4.3. System kontroli wersji - Git

Do zarządzania kodem źródłowym wykorzystano system kontroli wersji Git. Pozwoliło to na bezpieczne wprowadzanie zmian, tworzenie gałęzi (branches) dla poszczególnych funkcjonalności (np. system walki, system budowy) oraz łatwe wycofywanie zmian w razie wystąpienia regresji. Repozytorium zostało umieszczone na platformie GitHub, co zapewniło dodatkową kopię zapasową oraz umożliwiło korzystanie z mechanizmów ciągłej integracji.

1.4.4. Zarządzanie bazą danych - MySQL Workbench i DBeaver

Do projektowania schematu relacyjnego oraz zarządzania danymi testowymi wykorzystano narzędzia MySQL Workbench oraz DBeaver. Pozwoliły one na wizualizację powiązań między tabelami (diagramy ERD) oraz szybkie wykonywanie zapytań SQL w celu weryfikacji poprawności operacji wykonywanych przez system TypeORM.

1.4.5. Testowanie i dokumentacja API - Postman

Podczas tworzenia warstwy backendowej, kluczowym narzędziem był Postman. Służył on do ręcznego testowania punktów końcowych REST API przed ich integracją z front-endem.

Dzięki możliwości tworzenia kolekcji żądań oraz definiowania zmiennych środowiskowych, proces weryfikacji logiki biznesowej (np. przesyłanie surowców, logowanie) przebiegał sprawnie i systematycznie.

1.4.6. Menedżer pakietów - npm

Do zarządzania zależnościami projektowymi, zarówno w części serwerowej, jak i klienckiej, wykorzystano menedżer pakietów npm (Node Package Manager). Pozwolił on na szybką instalację niezbędnych bibliotek, zarządzanie ich wersjami oraz definiowanie skryptów automatyzujących procesy budowania i uruchamiania aplikacji.

Rozdział 2

Projekt gry - Game Design Document (GDD)

2.1. Ogólna koncepcja gry

Zaprojektowana aplikacja jest strategiczną grą czasu rzeczywistego osadzoną w realiach średniowiecza, działającą w środowisku przeglądarki internetowej. Gracz wciela się w rolę zarządcy niewielkiej osady, a jego głównym celem jest przekształcenie jej w potężne imperium przez rozwój ekonomiczny oraz militarny. Kluczowym elementem rozgrywki jest zarządzanie ograniczonymi zasobami (czas, surowce, populacja) w taki sposób, aby zmaksymalizować efektywność rozwoju w stosunku do innych uczestników gry. System nie posiada zdefiniowanego, liniowego scenariusza.

2.1.1. Opis gatunku i grupy docelowej

Projektowana gra wpisuje się w ramy gatunku MMORTS, łączac klasyczne zarządzanie zasobami z modelem trwałego świata. Z perspektywy projektowej, kluczowym aspektem jest asynchroniczność interakcji, czyli gracz podejmuje decyzje w wybranym przez siebie momencie, jednak ich skutki materializują się w czasie rzeczywistym, narzuconym przez parametry serwera.

Grupę docelową aplikacji stanowią przede wszystkim osoby dorosłe oraz młodzież poszukujące gier strategicznych o niskim progu wejścia. Profil typowego odbiorcy obejmuje przede wszystkim graczy preferujących rozgrywkę długofalową, którzy cenią planowanie strategiczne rozłożone na tygodnie lub miesiące zamiast krótkich i intensywnych sesji. Istotnym filarem zaangażowania odbiorców jest także wysoka potrzeba interakcji społecznej, budowanie społeczności, co jest cechą charakterystyczną dla fanów pierwowzoru, jakim była seria „Plemiona".

2.1.2. Klasyfikacja wiekowa i uzasadnienie

Projektowana aplikacja została stworzona z myślą o użytkownikach powyżej 12 roku życia. Rekomendacja ta wynika zarówno z charakteru mechanik gry, jak i sposobu prezentacji treści. Choć gra osadzona jest w realiach średniowiecznych konfliktów zbrojnych, wszelkie przejawy rywalizacji militarnej mają charakter symboliczny. Kolejnym argumentem przemawiającym za taką grupą wiekową jest stopień złożoności warstwy ekonomicznej i strategicznej. Gra wymaga od użytkownika umiejętności logicznego myślenia, planowania długofalowego oraz zarządzania ograniczonymi zasobami, co może być zbyt wymagające dla bardzo młodych graczy. Jednocześnie system interakcji online sugeruje, że odbiorca powinien posiadać podstawową dojrzałość społeczną niezbędną do funkcjonowania w środowisku wieloosobowym. Całość projektu jest wolna od treści promujących zachowania patologiczne, używki czy hazard.

2.2. Szczegółowy opis mechaniki gry

2.2.1. System ekonomii i surowców

System ekonomiczny stanowi fundament funkcjonowania osady, determinując możliwości rozwoju militarnego i infrastrukturalnego gracza. W grze zaimplementowano trzy główne surowce: drewno, glinę oraz żelazo, które są przechowywane i zarządzane w ramach profilu użytkownika na konkretnym świecie - serwerze.

Mechanika pozyskania zasobów opiera się na modelu generowania pasywnego, realizowanego po stronie serwera w interwałach czasowych. Za proces ten odpowiada metoda *handleResourceGeneration* w serwisie *JobsService*, która co 10 sekund przelicza przyrost surowców dla wszystkich zalogowanych graczy. Takie podejście zapewnia wysoką dynamikę rozgrywki i pozwala na natychmiastowe aktualizowanie stanu posiadania użytkownika bez konieczności odświeżania strony.

Logika obliczania produkcji odpowiada metoda *calculateProduction* bazuje na sumowaniu wydajności poszczególnych budynków produkcyjnych przypisanych do wioski. Wydajność ta jest liniowo zależna od poziomu budynku (L) oraz jego bazowej stopy przyrostu (B_{rate}):

$$P_{total} = \sum(B_{rate} \times L)$$

W implementacji przyjęto następujące wartości bazowe: 10 dla tartaku (*Sawmill*), 7 dla cegielni (*Clay Pit*) oraz 5 dla huty (*Smithy*). System każdorazowo weryfikuje status budynku - jeśli obiekt jest w trakcie budowy lub ulepszania, jego produkcja nie jest doliczana do bilansu, co wymusza na graczu strategiczne planowanie przerw w dostawie surowców na rzecz przyszłych zysków.

Surowiec	**Generator (źródło)**
Drewno	Tartak
Glina	Glina
Żelazo	Kowal
Ludność	Zadania lub rozwijanie wioski

Tabela 2.1. Charakterystyka surowców w grze: źródła pozyskiwania i zastosowanie – opracowanie własne.

Integralną częścią ekonomii jest również system populacji, zarządzany przez budynki mieszkalne (*Residential House*). Każdy poziom tego budynku zwiększa maksymalny limit ludności.

2.2.2. Rozbudowa osady

Rozbudowa osady jest procesem wieloetapowym, łączącym zarządzanie zasobami, przestrzenią oraz czasem. Każda wioska jest reprezentowana dwuwymiarowa siatka współrzędnych (*row*, *col*), na której gracz może rozmieszczać budynki o różnym przeznaczeniu.

Model kosztów i progresji

Proces wznoszenia budowli jest obciążony kosztem surowcowym, zdefiniowanym w stałej *BUILDING_COSTS*. Podczas gdy budowa pierwszego poziomu wymaga stałej liczby zasobów, każde kolejne ulepszenie, realizowane przez metodę *startUpgradeForUser*, wiąże się z wykładniczym wzrostem ceny. Algorytm wykorzystuje mnożnik 1.2:

$$C_{upgrade} = C_{base} \times 1.2^{L}$$

Nazwa budynku	**Generowany efekt / Funkcja**
Tartak	Generuje drewno w czasie rzeczywistym. Produkcja rośnie wraz z poziomem.
Glina	Generuje glinę w czasie rzeczywistym. Produkcja rośnie wraz z poziomem.
Kowal	Generuje żelazo w czasie rzeczywistym. Produkcja rośnie wraz z poziomem.
Dom mieszkalny	Zwiększa limit populacji.
Ratusz	Centralny budynek wioski. Stanowi element strukturalny osady.
Spichlerz	Budynek magazynowy. W obecnej wersji stanowi element rozbudowy wioski.
Młyn	Stanowi element rozbudowy infrastruktury wioski.

Tabela 2.2. Zestawienie budynków dostępnych w grze oraz ich wpływ na rozgrywkę – opracowanie własne.

Gdzie L oznacza aktualny poziom budynku. Zastosowanie tej formuły sprawia, że rozwój na wysokich poziomach staje się wyzwaniem ekonomicznym, co jest kluczowym elementem balansu gry. Czas potrzebny na ukończenie inwestycji również skaluje się wraz z poziomem, co odzwierciedla rosnący stopień skomplikowania struktury.

Bezpieczeństwo transakcyjne i stany budowy

Ze względu na asynchroniczny charakter operacji, implementacja wznoszenia budowli opiera się na transakcjach bazodanowych (*QueryRunner*). Gwarantuje to atomowość operacji: w jednym cyklu system sprawdza dostępność pola na mapie, weryfikuje stan surowców gracza, odejmuje odpowiednie kwoty i zapisuje nową strukturę. W przypadku niepowodzenia na dowolnym etapie, np. przy próbie budowy na zajętym polu, system wykonuje *rollback*, zapobiegając błędom spójności danych. Budynki przechodzą przez kilka stanów cyklu życia:

— **W trakcie budowy/ulepszania:** Obiekt posiada ustawioną datę *constructionFinishedAt*. W tym stanie jego funkcje (np. produkcja surowców) są zawieszone.
— **Aktywny:** Po osiągnięciu zdefiniowanego czasu, serwerowy proces tła (*processFinishedConstructions*) finalizuje budowę, zerując znacznik czasu i aktualizując parametry budynku (zdrowie, poziom).
— **W naprawie:** Metoda *startRepairForUser* pozwala na przywrócenie punktów wytrzymałości (*health*) budynku, co jest niezbędne po ewentualnych uszkodzeniach wynikających z działań wojennych.

Personalizacja i komunikacja

Dla zwiększenia komfortu użytkowania, system umożliwia relokację istniejących budynków za pomocą funkcji *moveForUser*. Mechanika ta pozwala na zamianę miejsc (*swap*) między dwoma budynkami na siatce wioski bez konieczności ich wyburzania. Wszystkie kluczowe zdarzenia - zakończenie budowy, aktualizacja zasobów czy zmiana poziomu - są komunikowane klientowi w czasie rzeczywistym poprzez WebSockets. Dzięki temu interfejs użytkownika jest zawsze zsynchronizowany ze stanem serwera, co minimalizuje ryzyko wystąpienia konfliktów w logice gry po stronie klienta.

2.2.3. System jednostek wojskowych i walki

Możliwość formowania siły militarnej oraz wykorzystywania jej w celach ofensywnych i defensywnych stanowi fundamentalny element rozgrywki w grach typu MMORTS. W ramach realizowanego projektu zaprojektowano i zaimplementowano złożony moduł wosjkowy, który obejmuje dwa główne obszary funkcjonalne: system zarządzania armią (rekrutacja i ulepszanie jednostek) oraz silnik symulacji walki w czasie rzeczywistym. Rozwiązanie to odchodzi od statycznego modelu generowania raportów, typowego dla starszych gier przeglądarkowych, na rzecz dynamicznej wizualizacji potyczek.

Zarządzanie rekrutacją i integralność danych

Podstawowym mechanizmem rozwoju potencjału militarnego jest rekrutacja jednostek, która w warstwie logicznej polega na wymianie zgromadzonych surowców na obiekty reprezentujące oddziały wojskowe. Proces ten jest newralgiczny z punktu widzenia spójności danych, gdyż istnieje ryzyko wystąpienia zjawiska wyścigu. Sytuacja taka mogłaby zaistnieć, gdyby użytkownik wysłał niemal jednocześnie dwa żądania rekrutacji, co przy braku odpowiednich zabezpieczeń mogłoby doprowadzić do dwukrotnego utworzenia jednostek przy jednokrotnym zabraniu surowców. Aby wyeliminować to zagrożenie, w warstwie serwerowej zaimplementowano mechanizm transakcji bazodanowych połączonych z blokowaniem pesymistycznym. Każda operacja rekrutacji (obsługiwana przez metode *recruitUnits*) rozpoczyna się od otwarcia transakcji, wewnątrz której system pobiera aktualny stan wioski i nakłada blokadę na rekordy jednostek. Gwarantuje to, że żadna inna operacja nie zmodyfikuje tych danych do momentu zakończenia bieżącego przetwarzania. Całkowity koszt rekrutacji C_{total} dla danego typu jednostki obliczany jest zgodnie z poniższym wzorem

$$C_{total} = \sum_{r \in R} (C_{base}(u, r) \cdot N)$$

gdzie R oznacza zbiór typów surowców (drewno, glina, żelazo), $C_{base}(u, r)$ to bazowy koszt surowca r dla jednostki typu u, a N to żądana liczba jednostek. Dopiero po pomyślnym zweryfikowaniu dostępności zasobów i ich pobraniu z konta gracza, system aktualizuje stan liczebny armii. W przypadku jakiegokolwiek błędu logicznego następuje wycofanie transakcji, co przywraca stan bazy danych do postaci pierwotnej. Ponadto system umożliwia ulepszanie jednostek (metoda *upgradeUnit*), co zwiększa ich statystyki bojowe. Koszt takiej operacji rośnie liniowo wraz z poziomem, co ma na celu zbalansowanie rozgrywki i zapobieganie zbyt szybkiej dominacji. Koszt ulepszenia $C_{upgrade}$ na poziom L definiowany jest zależnością:

$$C_{upgrade} = C_{base} \cdot L \cdot 2$$

Zastosowanie mnożnika sprawia, że inwestycja w jakość wojska staje się w późniejszych etapach gry znaczącym obciążeniem ekonomicznym, wymuszającym na graczu podejmowanie strategicznych decyzji.

Architektura i matematyka symulacji bitewnej

W przeciwieństwie do modelu turowego lub natychmiastowego obliczania wyniku, opracowany system walki działa w oparciu o pętlę symulacyjną realizowaną w pamięci operacyjnej serwera. Wybór architektury *in-memory*, zaimplementowanej w klasie *BattlesService*, podyktowany był koniecznością częstego aktualizowania stanu bitwy (co 100 milisekund) bez nadmiernego obciążania dyskowej bazy danych operacjami zapisu.

Przestrzeń walki została odwzorowana jako dwuwymiarowy układ kartezjański, w którym jednostki atakującego i obrońcy posiadają określone współrzędne (x, y). Ruch wojsk realizowany jest poprzez przesuwanie ich pozycji w kierunku celu z prędkością v. Nowa pozycja w każdej klatce symulacji obliczana jest na podstawie znormalizowanego wektora kierunkowego pomnożonego przez prędkość jednostki. Wzór na przesunięcie wzdłuż osi X przyjmuje postać:

$$x_{new} = x_{old} + \frac{x_{target} - x_{old}}{d} \cdot v$$

gdzie d oznacza odległość euklidesową między aktualną pozycją a celem, wyrażoną wzorem:

$$d = \sqrt{(x_{target} - x_{old})^2 + (y_{target} - y_{old})^2}$$

Analogiczne obliczenia wykonywane są dla współrzędnej Y. Mechanizm ten pozwala na płynne animowanie przemieszczania się oddziałów w kierunku wrogich jednostek lub budynków strukturalnych przeciwnika.

W celu optymalizacji wydajności obliczeniowej zastosowano agregację jednostek. Zamiast symulować zachowanie każdego żołnierza z osobna, system operuje na zagregowanych obiektach reprezentujących całą armię. Parametry takie jak łączne punkty życia (HP_{total}) oraz siła ataku (DMG_{total}) są sumą wartości poszczególnych jednostek składowych. Podejście to znacząco redukuje złożoność obliczeniową symulacji, jednocześnie zachowując wystarczający poziom szczegółowości dla potrzeb strategicznych.

Logika konfrontacji i aktualizacja stanu

Logika sterująca zachowaniem armii oparta jest na maszynie stanów skończonych. Jednostki mogą znajdować się w stanie marszu, walki w zwarciu lub niszczenia infrastruktury. Przejście w stan walki następuje automatycznie, gdy odległość d między wrogimi armiami spadnie poniżej zdefiniowanego progu kolizji (w projekcie przyjęto wartość 50 jednostek odległości).

W fazie walki system cyklicznie zadaje obrażenia obu stronom konfliktu. Ilość zadawanych obrażeń jest skalowana w zależności od siły ataku, a utrata punktów życia przekłada się bezpośrednio na spadek liczebności oddziału. Aktualna liczba jednostek N_{curr} po otrzymaniu obrażeń wyznaczana jest z proporcji:

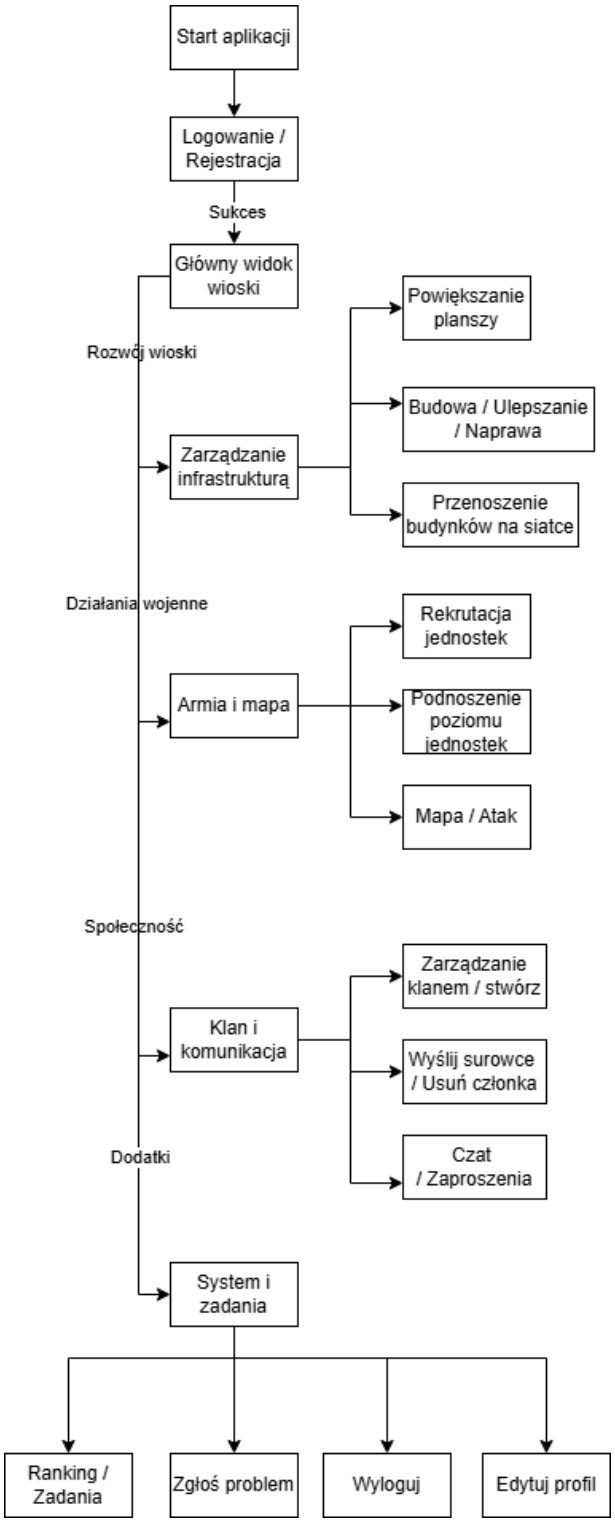
$$N_{curr} = \left\lfloor N_{init} \cdot \frac{HP_{curr}}{HP_{max}} \right\rfloor$$

Dzięki temu rozwiązaniu straty są rozkładane równomiernie, a siła bojowa armii maleje wraz z trwaniem potyczki. Po wyeliminowaniu obrońców, armia agresora automatycznie przechodzi w tryb niszczenia budynków, wyszukując najbliższy obiekt o dodatniej wartości punktów wytrzymałości. Kluczowym aspektem systemu jest synchronizacja stanu symulacji z interfejsem użytkownika. Wykorzystano w tym celu protokół WebSocket, który umożliwia dwukierunkową komunikację w czasie rzeczywistym. Serwer cyklicznie emituje zdarzenia zawierające aktualne koordynaty i stan zdrowia jednostek, co pozwala klientowi na płynne renderowanie przebiegu bitwy. Po zakończeniu starcia następuje finalizacja procesu: pętla symulacyjna zostaje zatrzymana, a ostateczne wyniki - w tym straty obu stron oraz zrabowane surowce - są trwale zapisywane w bazie danych MySQL, kończąc tym samym cykl życia instancji bitwy w pamięci operacyjnej.

2.3. Logika i przepływ gry

2.3.1. Diagram przepływu gry

Prezentowany schemat (Rys. 2.1) ilustruje przepływ sterowania oraz architekturę nawigacji w aplikacji. Układ ten opiera się na strukturze hierarchicznej.



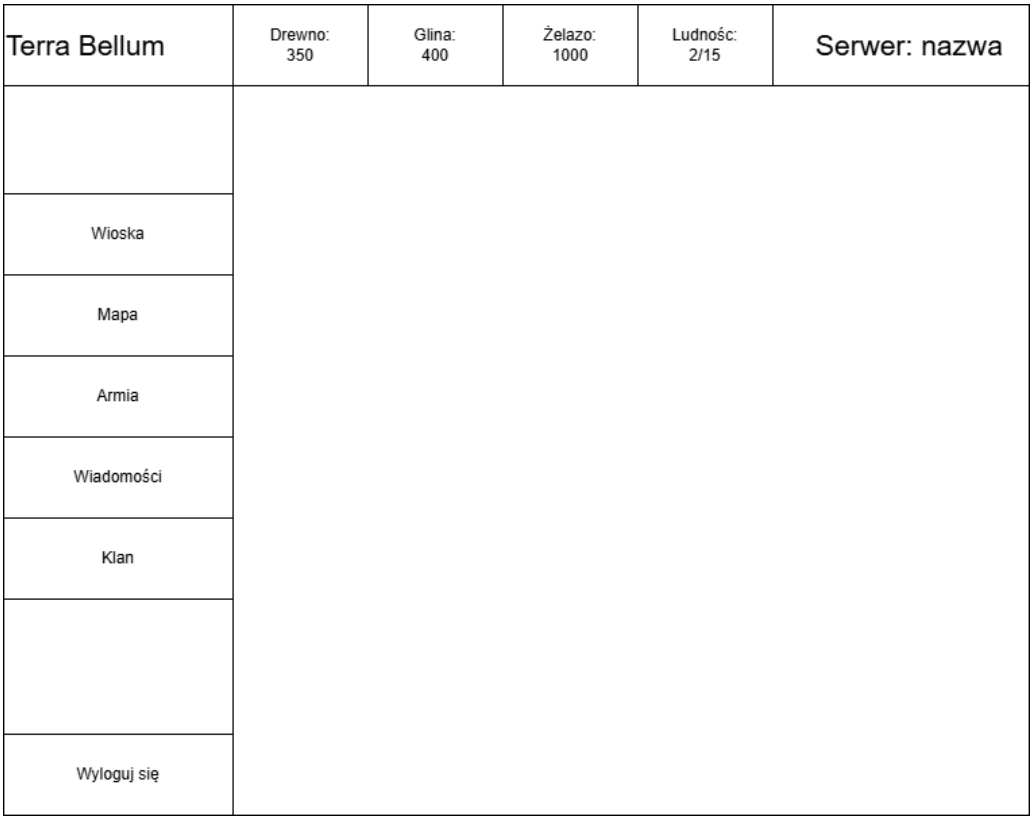
Rysunek 2.1. Diagram przepływu gry – opracowanie własne z wykorzystaniem draw.io.

2.4. Projekty graficznego interfejsu użytkownika

Projekt interfejsu użytkownika został opracowany, kładąc nacisk na przejrzystość. Ze względu na złożoność mechanik MMORTS, interfejs został podzielony na logiczne moduły, które zapewniają stały dostęp do informacji o stanie surowców oraz szybką nawigację pomiędzy kluczowymi widokami aplikacji.

2.4.1. Główny kokpit i system nawigacji

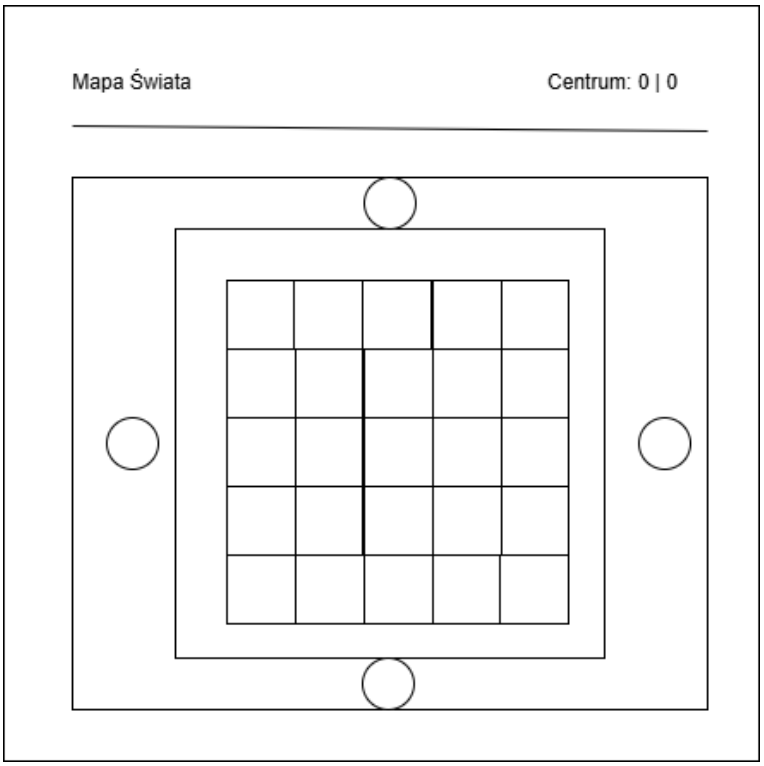
Główny układ aplikacji (Rys. 2.2) opiera się na strukturze ramowej, która pozostaje stała podczas poruszania się po różnych modułach gry. Górny pasek pełni funkcję informacyjną. Prezentuje w czasie rzeczywistym stan surowców oraz populację. Centralne umieszczenie pozwala na błyskawiczną ocenę zdolności ekonomicznych. Boczny panel nawigacyjny jest zlokalizowany po lewej stronie. Zawiera listę odnośników do poszczególnych funkcjonalności.



Rysunek 2.2. Główny kokpit – opracowanie własne z wykorzystaniem draw.io.

2.4.2. Mapa świata

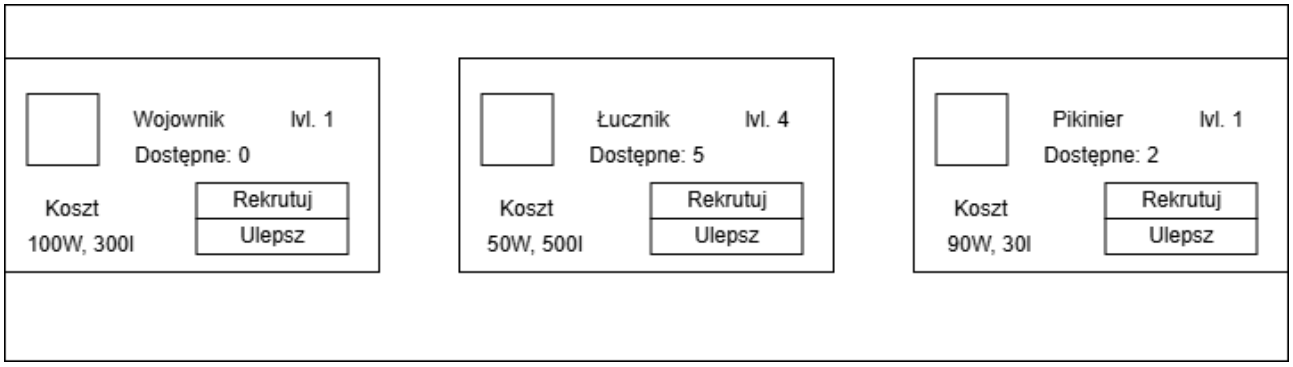
Widok mapy (Rys. 2.3) został zaprojektowany jako dwuwymiarowa siatka. Jest to kluczowy element interfejsu służący do interakcji z innymi graczami.



Rysunek 2.3. Mapa świata – opracowanie własne z wykorzystaniem draw.io.

2.4.3. Panel zarządzania jednostkami

Moduł militarny (Rys. 2.4) wykorzystuje układ kart, gdzie każda jednostka (Wojownik, Łucznik, Pikinier) reprezentowana jest przez osobny kontener. Główny nacisk padł na przejrzystość, każda karta ukazuje aktualny poziom jednostki, liczba posiadanych oddziałów.



Rysunek 2.4. Zarządzanie jednostkami – opracowanie własne z wykorzystaniem draw.io.

2.4.4. System komunikacji

Moduł wiadomości (Rys. 2.5) został zaprojektowany w układzie dwukolumnowym. Lewa kolumna zawiera listę aktywnych czatów wraz z wyszukiwarką użytkowników, co umożliwia szybkie filtrowanie kontaktów. Prawa to okno czatu z dynamicznie renderowanymi dymkami wiadomości.

Rysunek 2.5. System komunikacji – opracowanie własne z wykorzystaniem draw.io.

Rozdział 3

Implementacja projektu

3.1. Architektura systemu (Komunikacja Klient-Serwer)

Zaprojektowana aplikacja opiera się na nowoczesnej architekturze trójwarstwowej, w której wyraźnie odseparowano warstwę prezentacji, warstwę logiki biznesowej oraz warstwę trwałości danych. Takie podejście, znane jako architektura klient-serwer, jest standardem w tworzeniu skalowalnych systemów internetowych. W kontekście gry typu MMORTS, gdzie kluczowa jest integralność rozgrywki oraz bezpieczeństwo danych, zastosowano model, w którym serwer pełni rolę nadrzędnego autorytetu (ang. *Authoritative Server*). Oznacza to, że wszelkie kluczowe obliczenia, takie jak wynik bitwy, przyrost surowców czy weryfikacja możliwości budowy, odbywają się wyłącznie po stronie serwera, a klient aplikacji służy jedynie do wizualizacji stanu gry oraz przesyłania żądań użytkownika.

Komunikacja pomiędzy warstwą kliencką, zrealizowaną w frameworku Angular, a warstwą serwerową, opartą na NestJS, odbywa się dwutorowo, wykorzystując hybrydowe podejście do wymiany danych. Podstawowym kanałem komunikacyjnym jest protokół HTTP, realizujący architekturę REST API. Służy on do obsługi operacji inicjowanych przez użytkownika, które nie wymagają natychmiastowej reakcji zwrotnej w trybie ciągłym, takich jak logowanie do systemu, pobieranie statycznych danych konfiguracyjnych czy zlecanie rozbudowy infrastruktury. Każde żądanie przesłane w tym modelu jest bezstanowe i zawiera nagłówek autoryzacyjny z tokenem JWT, co pozwala serwerowi na identyfikację gracza bez konieczności utrzymywania sesji po stronie bazy danych. Dane przesyłane są w lekkim formacie JSON, a ich struktura jest ściśle definiowana poprzez obiekty transferu danych (DTO), co umożliwia automatyczną walidację poprawności typów i wartości po stronie backendu przed ich przetworzeniem.

Drugim, kluczowym dla dynamiki gry kanałem komunikacji, jest protokół WebSocket, zaimplementowany przy użyciu biblioteki Socket.io. Rozwiązanie to umożliwia nawiązanie trwałego, dwukierunkowego połączenia między przeglądarką a serwerem, co eliminuje narzut sieciowy związany z wielokrotnym nawiązywaniem połączeń HTTP. Zastosowanie WebSocketów jest niezbędne do obsługi zdarzeń asynchronicznych, które zachodzą niezależnie od akcji gracza, takich jak powiadomienie o ataku ze strony innego użytkownika, zakończenie procesu rekrutacji wojsk czy aktualizacja stanu bitwy w czasie rzeczywistym. Dzięki mechanizmowi server-push, serwer może natychmiastowo emitować zdarzenia (np. *resource_update*), co gwarantuje, że interfejs użytkownika zawsze odzwierciedla aktualny stan rozgrywki.

Architektura backendu została zaprojektowana w sposób modułowy, wykorzystując wzorzec wstrzykiwania zależności (Dependency Injection). Żądania trafiające do serwera są obsługiwane przez kontrolery, które pełnią rolę punktów wejścia i przekierowują ruch do odpowiednich serwisów realizujących logikę biznesową. Serwisy te komunikują się z bazą danych za pośrednictwem warstwy abstrakcji ORM (Object-Relational Mapping), co uniezależnia kod aplikacji od konkretnego silnika bazy danych i zwiększa bezpieczeństwo poprzez automatyczne parametryzowanie zapytań SQL. Taka separacja odpowiedzialności nie tylko ułatwia testowanie i konserwację kodu, ale również pozwala na potencjalne skalowanie systemu w przyszłości,

na przykład poprzez wydzielenie obsługi WebSocketów do osobnej mikro usługi w przypadku wzrostu obciążenia serwera.

3.2. Projekt i struktura bazy danych

Warstwa trwałości danych zaprojektowanego systemu została oparta na relacyjnym modelu bazy danych, co podyktowane było koniecznością zachowania ścisłej spójności transakcyjnej, niezbędnej w grach o charakterze ekonomiczno-strategicznym. Implementację fizyczną zrealizowano przy użyciu silnika MySQL, natomiast projekt logiczny schematu bazy danych został zdefiniowany w paradygmacie Code-First, wykorzystując mapowanie obiektowo-relacyjne (ORM). Architektura danych została podzielona na kilka kluczowych obszarów tematycznych: zarządzanie tożsamością i serwerami, strukturę przestrzenną i ekonomiczną wioski, system militarny, moduł zadań (questów) oraz warstwę społecznościową.

Centralnym elementem schematu jest tabela użytkowników, identyfikująca gracza w systemie globalnym. Ze względu na architekturę obsługującą wiele niezależnych światów gry, wprowadzono encję serwera, która definiuje parametry techniczne instancji, takie jak adres hosta, port czy status aktywności. Relacja między graczem a światem gry nie jest bezpośrednia, lecz realizowana poprzez tabelę zasobów. Tabela ta pełni funkcję łącznika, przypisując stan ekonomiczny (drewno, glina, żelazo, populacja) do pary kluczy obcych: identyfikatora użytkownika oraz identyfikatora serwera. Dzięki takiemu rozwiązaniu jeden profil użytkownika może posiadać niezależne postępy na wielu serwerach jednocześnie, zachowując separację danych rozgrywki.

Podstawową jednostką terytorialną w grze jest wioska, która stanowi węzeł łączący gracza z konkretnym serwerem. Relacje te zdefiniowano jako jeden-do-wielu, co oznacza, że jeden serwer zawiera wiele wiosek, a jeden użytkownik może zarządzać wieloma osadami. Wewnątrz struktury wioski zaimplementowano relację jeden-do-wielu względem budynków. Każdy rekord budynku przechowuje informacje o swoim typie, poziomie rozbudowy, punktach wytrzymałości oraz precyzyjnej lokalizacji na siatce współrzędnych osady. Zastosowano tu mechanizm kaskadowego usuwania danych, co gwarantuje, że w przypadku likwidacji wioski, wszelkie powiązane z nią struktury infrastrukturalne zostaną automatycznie usunięte, zapewniając integralność referencyjną.

System militarny został zoptymalizowany pod kątem wydajności poprzez agregację danych. Zamiast przechowywać każdą jednostkę wojskową jako osobny rekord, zastosowano tabelę oddziałów powiązaną z wioską relacją jeden-do-wielu. Kluczowym zabiegiem projektowym było nałożenie unikalnego ograniczenia na parę kolumn: identyfikator wioski oraz typ jednostki. W rezultacie, dla danej osady istnieje tylko jeden rekord reprezentujący dany rodzaj wojska (np. łuczników), przechowujący ich łączną liczebność oraz poziom zaawansowania technologicznego.

Rozbudowany schemat zaprojektowano również dla modułu zadań i progresji. Posiada on strukturę hierarchiczną, gdzie definicja zadania głównego jest powiązana relacją jeden-do-wielu ze szczegółowymi celami do wykonania. Śledzenie postępów gracza realizowane jest przez osobne tabele łączące użytkownika i serwer z definicjami zadań. Taka normalizacja bazy danych pozwala na oddzielenie statycznych danych konfiguracyjnych gry od dynamicznych danych o postępach poszczególnych graczy.

Warstwa społeczna systemu obejmuje relacje klanowe, zarządzanie znajomościami oraz komunikację. Klany modelowane są jako struktury przypisane do konkretnego serwera, posiadające jednego założyciela oraz wielu członków, co realizowane jest przez tabelę łączącą w relacji wiele-do-wielu. Rozszerzeniem interakcji między graczami jest tabela zaproszeń do znajomych, która nie tworzy natychmiastowego powiązania, lecz przechowuje stan relacji między inicjato-

rem a odbiorcą zaproszenia. System ten wykorzystuje statusy do określenia etapu znajomości (oczekująca, zaakceptowana, odrzucona). System komunikacji podzielono na wiadomości prywatne oraz czaty grupowe, a system raportowania nadużyć tworzy powiązania między użytkownikiem zgłaszającym, zgłaszanym a moderatorem, co pozwala na pełną audytowalność działań administracyjnych.

3.3. Omówienie kluczowych klas i struktur danych

Implementacja warstwy danych w projekcie została zrealizowana przy użyciu wzorca Active Record, wspieranego przez bibliotekę TypeORM. Dzięki temu, tabele widoczne na schemacie relacyjnym (Rys. 3.1) mają swoje bezpośrednie odzwierciedlenie w postaci klas TypeScript (tzw. encji), udekorowanych odpowiednimi metadanymi. Takie podejście pozwoliło na zachowanie silnego typowania oraz spójności logicznej pomiędzy strukturą bazy danych a kodem aplikacji.

3.3.1. Reprezentacja użytkownika i serwera

Centralnym punktem modelu jest klasa *User*, mapująca tabelę *users*. Przechowuje ona nie tylko dane uwierzytelniające (zahasłowane hasło, adres email, login), ale także metadane konta, takie jak rola w systemie (*role*), status aktywności czy znacznik usunięcia konta (*deleteAt*), co pozwala na implementację tzw. *soft delete*. Z uwagi na architekturę obsługującą wiele światów gry, klasa *User* nie przechowuje bezpośrednio stanu gry. Zamiast tego, powiązana jest relacjami jeden-do-wielu z encjami pośredniczącymi, takimi jak *Resource* czy *Village*, które są ściśle przypisane do konkretnej instancji klasy *Server*. Klasa *Server* definiuje parametry techniczne świata gry (hostname, port) oraz jego status (aktywny).

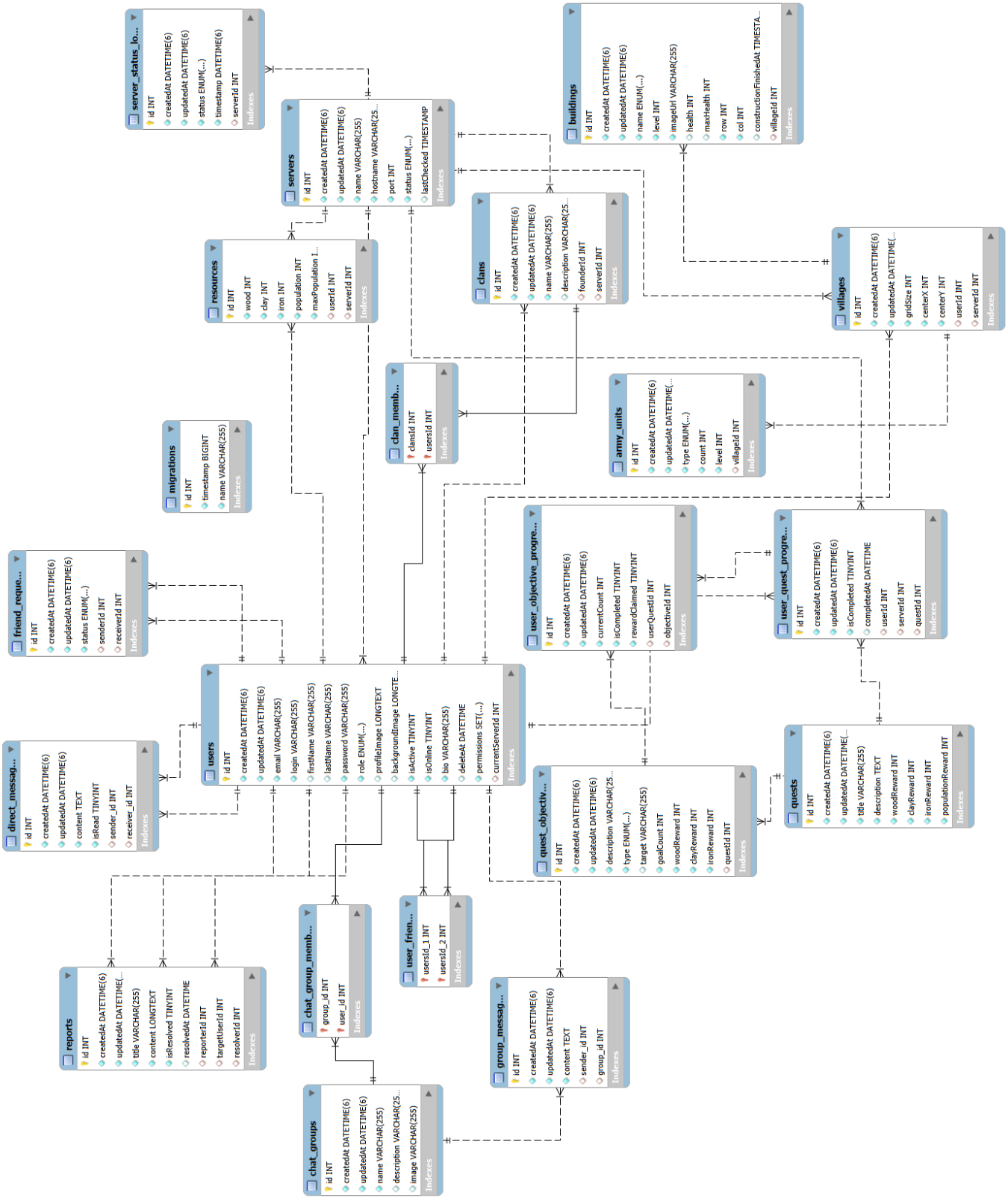
3.3.2. Struktura wioski i zabudowy

Klasa *Village* stanowi fundamentalny obiekt w przestrzeni gry. Odzwierciedla ona osadę gracza na mapie świata, przechowując współrzędne globalne (*centerX*, *centerY*) oraz odniesienie do właściciela (*userId*). Kluczowym atrybutem jest tutaj *gridSize*, definiujący rozmiar wewnętrznej siatki budowlanej wioski.

Powiązana z nią relacją kompozycji klasa *Building* odpowiada za poszczególne struktury wewnątrz osady, pełniąc kluczową rolę w mechanice rozwoju. Lokalizacja każdego obiektu na wirtualnej siatce wioski (x, y) jest ściśle definiowana przez atrybuty *row* oraz *col*, co umożliwia klientowi gry poprawne renderowanie rozmieszczenia infrastruktury. Rodzaj wznoszonej konstrukcji (np. Ratusz) określany jest poprzez pole *type* wykorzystujące typ wyliczeniowy. Z punktu widzenia logiki biznesowej, najbardziej istotnym elementem encji jest pole *constructionFinishedAt* typu *TIMESTAMP*. Przechowuje ono datę planowanego zakończenia budowy lub ulepszania; jeżeli wartość ta wskazuje na czas przyszły, serwer interpretuje budynek jako będący w trakcie konstrukcji, co skutkuje blokadą powiązanych procesów. Model dopełniają parametry *health* oraz *maxHealth*, które definiują aktualną oraz maksymalną wytrzymałość struktury, wykorzystywaną przez silnik walki do kalkulacji zniszczeń powstałych w wyniku działań oblężniczych.

3.3.3. Ekonomia i wojskowość

Stan ekonomiczny gracza na danym serwerze reprezentowany jest przez klasę *Resource*. Jest to encja łącząca użytkownika z serwerem, przechowująca bieżące wartości surowców (*wood*, *clay*, *iron*) oraz parametry populacyjne (*population*, *maxPopulation*). Oddzielenie zasobów od



Rysunek 3.1. Opracowanie własne z wykorzystaniem MySQL Workbench

samej wioski (w kontekście relacji z tabelą *resources*) pozwala na globalne zarządzanie bilansem surowcowym.

Klasa *ArmyUnit* odpowiada za stan militarny. Zgodnie z przyjętą strategią optymalizacji, klasa ta nie reprezentuje pojedynczego żołnierza, lecz oddział danego typu. Klucz unikalny złożony z identyfikatora wioski (*villageId*) oraz typu jednostki (*type*) gwarantuje, że w bazie istnieje tylko jeden rekord dla danego rodzaju wojska w konkretnej wiosce. Pole *count* przechowuje liczebność oddziału, a *level* stopień jego ulepszenia.

3.3.4. Aspekty społeczne i komunikacja

Dla obsługi interakcji między graczami zaprojektowano zestaw klas tworzących graf powiązań społecznych. Klasa *Clan* agreguje graczy w grupy sojusznicze, przechowując informacje o założycielu (*founderId*). Członkostwo realizowane jest przez encję *ClanMember*, łączącą klan z użytkownikiem. System komunikacji oparty jest na klasach *ChatGroup* oraz wiadomościach (*DirectMessage*, *GroupMessage*). Struktura tych klas, zawierająca pola *senderId*, *receiverId* oraz znaczniki czasowe, została zoptymalizowana pod kątem szybkiego pobierania historii rozmów w porządku chronologicznym.

3.3.5. System zadań (Quests)

Progresja gracza modelowana jest przez klasę *Quest*, definiującą nagrody surowcowe za wykonanie zadań. Postęp użytkownika śledzony jest przez klasę *UserQuestProgress*, która zawiera flagę *isCompleted* oraz datę zakończenia. Dzięki relacji z tabelą *quest_objectives*, system może weryfikować spełnienie konkretnych warunków (np. wybudowanie tartaku na 5. poziom) niezależnie dla każdego gracza. Całość struktury danych została zaprojektowana z uwzględnieniem indeksów, co zapewnia wysoką wydajność zapytań łączących (JOIN).

3.4. Zastosowane wzorce projektowe i zaawansowane mechanizmy architektoniczne

Projektowanie i implementacja złożonych systemów informatycznych, wymaga nie tylko znajomości składni języka programowania, ale przede wszystkim umiejętności budowania skalowalnej i łatwej w utrzymaniu architektury. W inżynierii oprogramowania fundamentalną rolę w tym procesie odgrywają wzorce projektowe. Są to sprawdzone, uniwersalne rozwiązania potwarzających się problemów architektonicznych.

W niniejszym projekcie, obejmującym zarówno warstwę serwerową (opartą na frameworku NestJS), jak i kliencką (zrealizowaną w Angularze), świadomie zaimplementowano szereg wzorców projektowych.

3.4.1. Wzorce strukturalne i behawioralne w warstwie serwerowej

Wykorzystany w projekcie framework NestJS narzuca, a jednocześnie ułatwia stosowanie dobrych praktyk programistycznych, czerpiąc inspiracje z architektury korporacyjnej. Poniżej omówiono kluczowe wzorce zastosowane w backendzie aplikacji.

Wstrzykiwanie Zależności (Dependency Injection)

Fundamentem architektury systemu jest wzorzec Wstrzykiwania Zależności (ang. *Dependency Injection*), będący realizacją szerszej zasady Odwrócenia Sterowania (ang. *Inversion*

of Control). W tradycyjnym podejściu komponenty same tworzą instancje obiektów, których potrzebują do działania, co prowadzi do silnego powiązania kodu. Wzrost zależności między klasami utrudnia testowanie i modyfikację systemu.

W realizowanym projekcie zastosowano kontener IoC dostarczany przez framework. Dzięki temu kontrolery obsługujące żądania HTTP oraz bramki WebSocket nie instancjonują samodzielnie serwisów logiki biznesowej, takich jak *BattlesService* czy *VillagesService*. Zamiast tego, zależności te są deklarowane w konstruktorach klas i "wstrzykiwane" przez framework w momencie uruchamiania aplikacji. Podejście to pozwoliło na zachowanie modularności systemu.

Wzorzec Singleton (Singleton Pattern)

Wzorzec Singleton to wzorzec kreacyjny, którego celem jest zagwarantowanie, że dana klasa posiada tylko jedną instancję w obrębie całej aplikacji, przy jednoczesnym zapewnieniu globalnego punktu dostępu do niej[4]. W architekturze NestJS większość dostawców usług (tzw. *Providers*) domyślnie funkcjonuje w oparciu o ten wzorzec.

W kontekście gry MMORTS zastosowanie Singletona ma kluczowe znaczenie dla zarządzania stanem połączeń w czasie rzeczywistym. Klasa *WsGateway*, odpowiedzialna za obsługę gniazd sieciowych (WebSocket) musi istnieć jako pojedyncza instancja współdzielona przez różne moduły aplikacji. Dzięki temu, niezależnie od tego, czy powiadomienie o ataku generowane jest przez moduł walki, czy moduł zadań, system korzysta z tej samej puli aktywnych połączeń, co zapewnia spójność komunikacji i optymalizuje zużycie pamięci operacyjnej serwera.

Zaawansowane mechanizmy: Dekoratory, Interceptory i Strażnicy

Oprócz klasycznych wzorców w projekcie wykorzystano zaawansowane techniki metaprogramowania i programowania zorientowanego aspektowo (AOP), które pozwalają na wyizolowanie logiki niebiznesowej z głównego nurtu aplikacji.

Autorskie Dekoratory (Custom Decorators)

Wzorzec Dekorator pozwala na dynamiczne dodawanie nowych zachowań do obiektów lub funkcji bez ingerencji w ich kod źródłowy[4]. W projekcie zdefiniowano i zaimplementowano kilka własnych dekoratorów w celu poprawy semantyki kodu i redukcji powtórzeń.

Pierwszym z nich jest dekorator parametrów, nazwany roboczo **@CurrentUser**. W standardowym podejściu, aby uzyskać dostęp do danych zalogowanego użytkownika, konieczne jest ręczne pobieranie obiektu żądania (Request) i ekstrahowanie z niego odpowiednich pól. Stworzony dekorator automatyzuje ten proces. Działa on na poziomie kontekstu wykonania (Execution Context), wyciągając obiekt użytkownika bezpośrednio z przetworzonego żądania HTTP i wstrzykując go jako gotowy argument do metody kontrolera. Dzięki temu kod metod biznesowych jest czystszy i skupiony wyłącznie na logice operacji.

Kolejną grupą są dekoratory metadanych, takie jak **Public** oraz **Message**. Nie zawierają one logiki wykonawczej, lecz służą do oznaczania (tagowania) metod kontrolerów.

— Dekorator **Public** służy do oznaczania punktów końcowych API, które mają być dostępne dla niezalogowanych użytkowników (np. rejestracja, logowanie). Ustawia on w metadanych klucz, który jest później odczytywany przez warstwę zabezpieczeń.

— Dekorator **Message** pozwala na zdefiniowanie czytelnego dla człowieka komunikatu (np. „Wioska została pomyślnie utworzona"), który zostanie dołączony do odpowiedzi serwera. Pozwala to na oddzielenie treści komunikatów od logiki ich przetwarzania.

Interceptory (Interceptors)

Interceptory w architekturze projektu pełnią funkcję zbliżoną do wzorca Pełnomocnik (Proxy) lub ogniwa w Łańcuchu Zobowiązań (Chain of Responsibility)[4]. Ich zadaniem jest przechwycenie wywołania metody, wykonanie operacji przed jej uruchomieniem lub - co było kluczowe w tym projekcie - transformacja wyniku zwróconego przez metodę.

Zaimplementowano globalny interceptor odpowiedzi (**MessageInterceptor**). Jego celem jest standaryzacja formatu danych wysyłanych do klienta. W złożonych systemach różne serwisy mogą zwracać dane w różnej postaci (tablice, obiekty, wartości proste). Interceptor przechwytuje każdy wynik zwracany przez kontroler i „opakowuje" go w jednolitą strukturę JSON. Struktura ta zawiera zawsze dwa klucze: pole z właściwymi danymi oraz pole z komunikatem statusu.

Mechanizm ten wykorzystuje bibliotekę RxJS i programowanie reaktywne (strumienie), aby połączyć wynik działania logiki biznesowej z komunikatem zdefiniowanym wcześniej za pomocą dekoratora **Message**. Dzięki temu warstwa frontendowa (klient Angular) zawsze otrzymuje przewidywalną strukturę odpowiedzi, co znacznie upraszcza obsługę błędów i wyświetlanie powiadomień użytkownikowi.

Strażnicy (Guards) i wzorzec Strategia

Strażnicy (Guards) odpowiadają za autoryzację, czyli decydowanie o tym, czy dane żądanie może zostać obsłużone przez system. Można ich traktować jako specyficzną implementację wzorca Strategia, gdzie logika weryfikacji uprawnień jest wydelegowana do osobnej klasy[4].

W projekcie stworzono strażnika **AuthenticatedGuard**. Jego działanie opiera się na ścisłej współpracy z wcześniej omówionymi dekoratorami metadanych. W momencie nadejścia żądania, strażnik wykorzystuje mechanizm refleksji (klasa Reflector), aby sprawdzić, czy docelowa metoda została oznaczona dekoratorem **Public**.

1. Jeżeli metoda jest oznaczona jako publiczna, strażnik natychmiast zezwala na dostęp, pomijając weryfikację tokena.
2. Jeżeli oznaczenia brak, strażnik sprawdza kontekst żądania w poszukiwaniu obiektu użytkownika (który jest tam umieszczany przez mechanizm uwierzytelniania JWT).
3. Jeżeli użytkownik nie istnieje lub nie posiada odpowiednich ról, strażnik blokuje wykonanie żądania, zwracając błąd autoryzacji (HTTP 401/403).

Takie podejście pozwala na deklaratywne zarządzanie bezpieczeństwem. Zamiast pisać instrukcje warunkowe *if-else*.

3.4.2. Wzorce projektowe w warstwie klienckiej (Frontend)

Warstwa prezentacji, zrealizowana w oparciu o framework Angular, również wykorzystuje szereg wzorców projektowych, niezbędnych do obsługi asynchronicznej natury aplikacji webowych.

Wzorzec Obserwator (Observer Pattern)

Jest to kluczowy wzorzec behawioralny wykorzystywany do obsługi zdarzeń. W przeciwieństwie do tradycyjnego modelu „odpytywania" (polling), gdzie klient cyklicznie pyta serwer o zmiany, zastosowano model reaktywny oparty na strumieniach danych (biblioteka RxJS).

Komponenty interfejsu, takie jak *GridComponent* subskrybują (ang. *subscribe*) odpowiednie serwisy. Serwisy te pełnią rolę „Obserwowanych" (Subjects). W momencie, gdy przez WebSocket nadejdzie informacja o zakończeniu budowy lub ataku, serwis emituje nową wartość do strumienia, a wszystkie subskrybujące go komponenty automatycznie aktualizują swój widok.

Dzięki temu interfejs użytkownika jest zawsze zsynchronizowany ze stanem serwera w czasie rzeczywistym, bez konieczności przeładowywania strony.

Wzorzec Kompozyt (Composite Pattern)

Interfejs użytkownika gry został zbudowany w oparciu o drzewiastą strukturę komponentów, co jest realizacją wzorca Kompozyt. Wzorzec ten pozwala traktować pojedyncze obiekty (np. przycisk, ikonę jednostki) oraz grupy obiektów (np. panel rekrutacji, widok całej wioski) w jednolity sposób[4].

Główny widok aplikacji składa się z mniejszych, niezależnych komponentów, które mogą zawierać w sobie kolejne podkomponenty. Taka hierarchiczna struktura ułatwia zarządzanie kodem (separacja odpowiedzialności widoku) oraz umożliwia wielokrotne wykorzystanie (reużywalność) tych samych elementów interfejsu w różnych miejscach aplikacji (np. komponent wyświetlający szczegóły jednostki jest używany zarówno w koszarach, jak i w raporcie z walki).

3.4.3. Podsumowanie rozwiązań architektonicznych

Analiza zastosowanych rozwiązań wykazuje, że projekt aplikacji nie jest jedynie zbiorem przypadkowych funkcjonalności, lecz przemyślanym systemem opartym na solidnych podstawach inżynierii oprogramowania. Wykorzystanie wzorców projektowych takich jak Wstrzykiwanie Zależności, Singleton czy Obserwator zapewniło systemowi skalowalność i wydajność.

Szczególną wartość dodaną stanowi implementacja własnych rozwiązań opartych na metaprogramowaniu - dekoratorów, interceptorów i strażników. Mechanizmy te pozwoliły na osiągnięcie wysokiego poziomu hermetyzacji, oddzielając logikę biznesową od logiki infrastrukturalnej (obsługa HTTP, bezpieczeństwo, formatowanie danych). Dzięki temu kod stał się bardziej czytelny i odporny na błędy, co jest kluczowym wyznacznikiem jakości w profesjonalnych projektach informatycznych.

Rozdział 4

Podsumowanie

4.1. Ocena realizacji celu

Główny cel pracy, zdefiniowany jako zaprojektowanie oraz implementacja wieloosobowej gry strategicznej czasu rzeczywistego (MMORTS) inspirowanej serią "Plemiona", został zrealizowany w pełnym zakresie. W wyniku podjętych prac inżynierskich powstała w pełni funkcjonalna aplikacja internetowa, która odwzorowuje kluczowe mechaniki gatunku, a jednocześnie spełnia współczesne standardy wytwarzania oprogramowania.

Weryfikacja przyjętych założeń projektowych pozwala na sformułowanie następujących wniosków dotyczących poszczególnych aspektów systemu:

Realizacja mechanik gry i logiki biznesowej
Z sukcesem zaimplementowano złożony model ekonomiczny oparty na ciągłym przyroście surowców, uwzględniający zależności czasowe. Mechanizm wznoszenia budynków oraz rekrutacji jednostek działa zgodnie z założeniami, poprawnie obsługując kolejkowanie zadań oraz blokady zasobów. Szczególnym osiągnięciem jest autorski silnik symulacji walki, który zamiast prostego porównania statystyk, realizuje dynamiczną konfrontację armii w pamięci operacyjnej serwera, uwzględniając parametry przestrzenne i czasowe.

Architektura i dobór technologii
Decyzja o wyborze języka TypeScript dla całego stosu technologicznego (full-stack) okazała się trafna. Zastosowanie frameworka NestJS po stronie serwera zapewniło wysoką skalowalność oraz ułatwiło zarządzanie złożoną logiką dzięki wstrzykiwaniu zależności i modułowości. Wyraźna separacja warstwy serwerowej od klienckiej (REST API) sprawia, że system jest elastyczny i gotowy na ewentualne podłączenie innych klientów, np. natywnej aplikacji mobilnej, bez konieczności modyfikacji backendu.

Komunikacja w czasie rzeczywistym i interfejs użytkownika
Implementacja protokołu WebSocket (Socket.io) pozwoliła na osiągnięcie płynności rozgrywki charakterystycznej dla aplikacji typu Single Page Application (SPA). Użytkownik otrzymuje powiadomienia o zakończeniu budowy, ataku czy nowej wiadomości natychmiastowo, bez konieczności przeładowywania strony. Warstwa prezentacji wykonana w Angularze poprawnie reaguje na strumienie danych, dynamicznie aktualizując widok mapy i statystyk.

Baza danych i spójność danych
Zaprojektowany schemat relacyjnej bazy danych MySQL, oparty na ścisłej normalizacji, skutecznie odwzorowuje skomplikowane relacje między graczami, wioskami i jednostkami. Zastosowanie mechanizmu transakcji oraz blokad (pessimistic locking) w krytycznych punktach systemu (np. podczas wydawania surowców) wyeliminowało ryzyko błędów spójności danych i prób oszustwa.

Podsumowując, stworzona aplikacja jest stabilnym, bezpiecznym i grywalnym systemem, który łączy klasyczne mechaniki gier przeglądarkowych z nowoczesną technologią webową. Wszystkie funkcjonalności wymagane do przeprowadzenia pełnoprawnej rozgrywki - od rejestracji, przez rozwój ekonomiczny, aż po działania militarne - działają poprawnie i stabilnie.

4.2. Wnioski końcowe

Realizacja niniejszej pracy inżynierskiej pozwoliła na praktyczne zweryfikowanie wiedzy z zakresu inżynierii oprogramowania oraz zmierzenie się z wyzwaniami charakterystycznymi dla tworzenia zaawansowanych systemów. Proces projektowania i implementacji gry typu MMORTS dowiódł, że tego rodzaju aplikacje stanowią znacznie większe wyzwanie architektoniczne niż standardowe systemy informacyjne typu CRUD (Create, Read, Update, Delete). Konieczność zarządzania czasem, obsługi zdarzeń w tle oraz utrzymania ciągłości symulacji wymagała zastosowania niestandardowych rozwiązań programistycznych.

Na podstawie przeprowadzonych prac można sformułować następujące wnioski:

Po pierwsze, **jednolite środowisko technologiczne oparte na języku TypeScript okazało się kluczowym czynnikiem sukcesu**. Współdzielenie modeli danych (interfejsów) między warstwą serwerową (NestJS) a kliencką (Angular) znacząco przyspieszyło proces developmentu i niemal całkowicie wyeliminowało błędy wynikające z niekompatybilności typów danych przesyłanych przez sieć. Silne typowanie statyczne pozwoliło na wczesne wykrywanie błędów na etapie kompilacji, co jest nieocenione przy tak złożonej logice biznesowej.

Po drugie, **obsługa współbieżności w grach wieloosobowych wymaga rygorystycznego podejścia do zarządzania danymi**. Wnioskiem z fazy implementacji jest to, że w środowisku, gdzie tysiące graczy może w tej samej milisekundzie wejść w interakcję z tym samym zasobem, poleganie wyłącznie na logice kodu aplikacji jest niewystarczające. Zastosowanie mechanizmów silnika bazy danych, takich jak transakcje oraz blokady pesymistyczne, jest absolutnie niezbędne dla zachowania uczciwości rozgrywki i uniknięcia błędów spójności (np. duplikacji surowców czy jednostek).

Po trzecie, **odejście od tradycyjnego modelu żądanie-odpowiedź na rzecz komunikacji strumieniowej definiuje współczesny standard gier przeglądarkowych**. Integracja protokołu WebSocket za pomocą biblioteki Socket.io udowodniła, że opóźnienia sieciowe można skutecznie maskować, a architektura sterowana zdarzeniami (Event-Driven Architecture) pozwala na stworzenie wrażenia płynności porównywalnego z grami natywnymi, uruchamianymi bezpośrednio na komputerze gracza.

Podsumowując, projekt potwierdził, że współczesne technologie webowe są w pełni dojrzałe do obsługi skomplikowanych gier strategicznych. Proces tworzenia aplikacji stanowił cenną lekcję z zakresu projektowania architektury oprogramowania, optymalizacji baz danych oraz zabezpieczania systemów przed nieautoryzowaną manipulacją. Stworzony system nie tylko zrealizował założenia projektowe, ale stanowi również solidny fundament edukacyjny i techniczny dla przyszłych przedsięwzięć komercyjnych.

4.3. Możliwości dalszego rozwoju projektu

Zaprojektowana aplikacja, dzięki zastosowaniu modularnej architektury opartej na wstrzykiwaniu zależności (NestJS) oraz separacji warstw (Client-Server), charakteryzuje się wysokim potencjałem rozwojowym. Obecna wersja systemu stanowi solidny rdzeń, który może być skalowany i rozbudowywany w wielu płaszczyznach bez konieczności przepisywania istniejącego kodu, a jedynie poprzez dodawanie nowych modułów i serwisów. Poniżej przedstawiono szczegółową mapę drogową dalszego rozwoju projektu.

4.3.1. Rozwój mechanik rozgrywki i logiki biznesowej

Warstwa rozgrywki posiada szerokie możliwości ekspansji, które zwiększyłyby głębię strategiczną oraz retencję graczy.

Zaawansowany system ekonomii i logistyki (Marketplace)
Naturalnym krokiem w ewolucji gry jest implementacja globalnego rynku surowców. Obecny model izolowanej produkcji można rozszerzyć o mechanizmy popytu i podaży, umożliwiając graczom wystawianie ofert handlowych. Implementacja wymagałaby wprowadzenia nowej klasy jednostek - kupców, których ruch po mapie byłby symulowany analogicznie do wojsk. Rozwój ten mógłby obejmować również system karawan sojuszniczych, co wymusiłoby na graczach ochronę szlaków handlowych, dodając nowy wymiar strategiczny.

Rozbudowa modułu militarnego o elementy RPG i szpiegostwo
System walki można wzbogacić o jednostki specjalne typu „Bohater", posiadające własne drzewka rozwoju, ekwipunek i umiejętności pasywne wpływające na statystyki armii. Dodatkowo, kluczowym elementem w grach strategicznych jest system wywiadowczy. Wprowadzenie jednostek zwiadowczych pozwoliłoby graczom na poznanie stanu armii i budynków przeciwnika przed atakiem (obecnie jest to nie ukryte). Od strony technicznej wymagałoby to modyfikacji silnika walki oraz rozszerzenia modelu bazy danych o atrybuty przedmiotów i doświadczenia bohaterów.

4.3.2. Ewolucja architektury i infrastruktury technicznej

Aby przygotować system na obsługę dziesiątek tysięcy jednoczesnych połączeń, niezbędne są dalsze prace optymalizacyjne w warstwie backendu. **Migracja do architektury mikroserwisowej**
Wraz ze wzrostem złożoności projektu, monolityczna struktura backendu może stać się ograniczeniem. Logiczny podział systemu na niezależne mikroserwisy (np. osobny serwis do obsługi walki, osobny do czatu, osobny do logowania), zwiększyłby niezawodność systemu. Awaria modułu czatu nie wpływałaby wówczas na możliwość prowadzenia rozgrywki.

Szarding bazy danych i wieloświatowość
Aby obsłużyć nieograniczoną liczbę graczy, system bazy danych powinien zostać przygotowany do horyzontalnego podziału danych. Oznacza to fizyczne rozdzielenie danych różnych światów gry na osobne instancje bazodanowe, przy zachowaniu wspólnego serwera uwierzytelniania.

4.3.3. Rozwój interfejsu i dostępności (Frontend)

Warstwa prezentacji posiada potencjał do transformacji w kierunku pełnej wieloplatformowości.

Dedykowane aplikacje mobilne
Dzięki oparciu komunikacji na REST API, stworzenie natywnych aplikacji mobilnych (iOS/Android) przy użyciu technologii takich jak React Native lub Flutter jest logicznym następstwem rozwoju. Aplikacje te mogłyby wykorzystywać natywne powiadomienia *Push*, informując gracza o ataku.

4.3.4. Narzędzia administracyjne i analityka

Ostatnim, lecz kluczowym obszarem rozwoju, jest zaplecze administracyjne.

Analityka Big Data
Implementacja systemu zbierania telemetrii pozwoliłaby na analizę zachowań graczy. Dane te

są bezcenne przy balansowaniu rozgrywki (np. wykrywanie, że dana jednostka jest zbyt silna) oraz optymalizacji procesów biznesowych.

Podsumowując, zrealizowany projekt inżynierski nie jest zamkniętą całością, lecz otwartą platformą technologiczną. Zastosowane wzorce projektowe i nowoczesny stos technologiczny sprawiają, że system jest gotowy na implementację powyższych funkcjonalności, co czyni go atrakcyjnym punktem wyjścia do prac badawczych lub komercyjnych w obszarze gier MMO.

Bibliografia

[1] Bartle R., *Designing Virtual Worlds*, New Riders Publishing, 2004.

[2] Wolf M. J. P., *Encyclopedia of Video Games: The Culture, Technology, and Art of Gaming*, Greenwood, 2012.

[3] Rollings A., Morris D., *Game Architecture and Design*, New Riders, 2003.

[4] Shvets A., Refactoring.Guru, `https://refactoring.guru/pl/design-patterns` [dostęp: 14.01.2026].

[5] InnoGames GmbH, *History of Tribal Wars*, `https://www.innogames.com/company/history/` [dostęp: 14.01.2026].